ABSTRACT

Title of Dissertation: Modeling and Solving Variants of the Vehicle Routing Problem: Algorithms, Test Problems, and Computational Results

Feiyue Li, Doctor of Philosophy, 2005

Dissertation Directed by: Professor Bruce Golden
Department of Decision and Information Technologies
Robert H. Smith School of Business

In the standard version of the capacitated vehicle routing problem (VRP), a sequence of deliveries is generated for each vehicle in a homogeneous fleet based at a single depot so that all customers are serviced and the total distance traveled by the fleet is minimized. Each vehicle has a fixed capacity and must leave from and return to the depot. Each vehicle might have a route-length restriction that limits the maximum distance it can travel. Each customer has a known demand and is serviced by exactly one visit of a single vehicle.

For more than 45 years, the standard VRP has attracted an enormous amount of attention in the operations research literature. There are many practical applications of vehicle routing in the distribution of products such as soft drinks, newspapers, groceries, and milk and in street sweeping, solid waste collection, and mail delivery.

In this dissertation, we model and solve variants of the standard VRP. First,

we focus on very large VRPs. We develop new, benchmark instances via a problem generator with as many as 1,200 customers along with estimated solutions. We also develop a simple, flexible, fast, and powerful heuristic solution procedure based on the record-to-record travel algorithm and apply our heuristic to the new problems and generate high-quality solutions very quickly.

Next, we turn our focus to five interesting variants of the VRP that have received little attention in the literature but have practical application in the real world: (1) the time dependent traveling salesman problem (TDTSP), (2) the noisy traveling salesman problem (NTSP), (3) the heterogeneous vehicle routing problem (HVRP), (4) the open vehicle routing problem (OVRP), and (5) the landfill routing problem (LRP). For each variant, we develop an effective solution procedure and report computational results. In particular, we solve the TDTSP, HVRP, OVRP, and LRP with our record-to-record travel-based heuristic and generate high-quality results. For the NTSP, we develop a new procedure based on quad trees that outperforms existing solution methods. Finally, for the HVRP and the OVRP, we generate new test problems and solve each new problem using our record-to-record travel-based heuristic.

Modeling and Solving Variants of the Vehicle Routing Problem: Algorithms, Test Problems, and Computational Results

by

Feiyue Li

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2005

Advisory Committee:

Professor Bruce Golden, Chairman/Advisor
Professor Edward Wasil
Proferssor Elise Miller-Hooks
Professor Paul Smith
Professor Philip Evers

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The standard version of the capacitated vehicle routing problem (VRP) is easy to state and very difficult to solve: Generate a sequence of deliveries for each vehicle in a homogeneous fleet based at a single depot so that all customers are serviced and the total distance traveled by the fleet is minimized. Each vehicle has a fixed capacity and must leave from and return to the depot. Each vehicle might have a route-length restriction that limits the maximum distance it can travel. Each customer has a known demand and is serviced by exactly one visit of a single vehicle.

The standard vehicle routing problem was introduced in the operations research and management science literature about 45 years ago. Since then, the vehicle routing problem has attracted an enormous amount of research attention. In the late 1990s, large vehicle routing problem instances with nearly 500 customers were generated and solved using metaheuristics. In Chapter 2, we focus on very large vehicle routing problems. Our contributions are threefold. First, we present problem instances with as many as 1,200 customers along with estimated solutions. Second, we introduce the variable-length neighbor list as a tool to reduce the number of unproductive computations and develop a solution procedure based on the record-to-record travel algorithm. Third, we apply record-to-record travel with a variable-length neighbor list to 32 problem

instances and obtain high-quality solutions very quickly.

In the standard version of the traveling salesman problem (TSP), we are given a set of customers located in and around a city and the distances between each pair of customers, and need to find the shortest tour that visits each customer exactly once. Suppose that some of the customers are located in the center of the city. Within a window of time, the center city becomes congested so that the time to travel between customers takes longer. Clearly, we would like to construct a tour that avoids visiting customers when the center of the city is congested. This variant of the TSP is known as the time dependent TSP (TDTSP). In Chapter 3, we review the literature on the TDTSP, develop two solution algorithms, and report computational experience with our algorithms.

Consider a truck that visits n households each day. The specific households (and their locations) vary slightly from one day to the next. In the noisy traveling salesman problem (NTSP), we develop a rough (skeleton) route which can then be adapted and modified to accommodate the actual node locations that need to be visited from day to day. In Chapter 4, we conduct extensive computational experiments on problems with $n = 100, 200$, and 300 nodes in order to compare several heuristics for solving the noisy traveling salesman problem including a new method based on quad trees. We find that the quad tree approach generates high-quality results quickly.

In the heterogeneous fleet vehicle routing problem (HVRP), several different types of vehicles can be used to service the customers. The types of vehicles differ with respect to capacity, fixed cost, and variable cost. We assume that the number of vehicles of each type is fixed and equal to a constant. We must decide how to make the best use of the fixed fleet of heterogeneous vehicles.

In Chapter 5, we review methods for solving the HVRP, develop a variant of

our record-to-record travel algorithm for the standard vehicle routing problem that takes a heterogeneous fleet into account, and report computational results on eight benchmark problems. Finally, we generate a new set of five test problems which have 200 to 360 customers and we solve each new problem using our record-to-record travel algorithm.

In the open vehicle routing problem (OVRP), a vehicle does not return to the depot after servicing the last customer on a route. The description of this variant of the standard vehicle routing problem appeared in the literature over 20 years ago, but has just recently attracted the attention of researchers and practitioners. In the last five years, tabu search, deterministic annealing, and large neighborhood search have been applied to the OVRP with some success. The OVRP is the focus of Chapter 6.

In Chapter 7, we study a real-life problem – the landfill routing problem (LRP). In the LRP, there are several landfills scattered in a geographical area. A homogeneous fleet of vehicles is used to collect trash from customers. If a vehicle is nearly full, it goes to a landfill and dumps all of its contents. The vehicle continues its work before a time limit is reached. We develop a savings heuristic to generate a feasible initial solution to the LRP. Then we apply our record-to-record travel algorithm to improve the initial solution.

To summarize, in the chapters that follow, we develop a simple, flexible, fast, and powerful heuristic solution procedure that is based on the record-to-record travel algorithm. Our heuristic contains a small number of parameters (values are easy to set), can easily accommodate side constraints, and can be applied to a wide range of routing problems.

We apply our heuristic to very large-scale vehicle routing problems with as many as 1,200 customers and find that, for the new benchmark problems that we

construct, it generates very good solutions. We also apply our heuristic to four variants of the standard vehicle routing problem with hundreds of customers, and find that, for many new, benchmark problems that we construct, it generates new, best-known solutions. We develop a new quad tree procedure that outperforms existing methods for solving the noisy traveling salesman problem.

Finally, we provide a problem generator that can be used by researchers to generate new instances of large-scale, open, and heterogeneous vehicle routing problems. Researchers can then apply their solution procedures to these new instances and compare results.

Chapter 2

The Large-Scale Vehicle Routing Problem

2.1 Introduction

The standard version of the capacitated vehicle routing problem (VRP) is easy to state and very difficult to solve: Generate a sequence of deliveries for each vehicle in a homogeneous fleet based at a single depot so that all customers are serviced and the total distance traveled by the fleet is minimized. Each vehicle has a fixed capacity and must leave from and return to the depot. Each vehicle might have a route-length restriction that limits the maximum distance it can travel. Each customer has a known demand and is serviced by exactly one visit of a single vehicle.

In the last five years, there has been a great deal of computational effort devoted to solving the 20 large-scale vehicle routing problems (denoted by LSVRPs) developed by Golden et al. [21]. These benchmark problems have 200 to 483 customers. Eight problems have route-length restrictions. Each problem has a geometric symmetry that allows a user to estimate a high-quality solution (eight problems have customers located in concentric circles around the depot, four problems have customers located in concentric squares with the depot located in one corner, four problems have customers located in concentric squares

around the depot, and four problems have customers located in a six-pointed star around the depot).

Researchers have applied general-purpose metaheuristics including deterministic annealing and tabu search to the 20 LSVRPs and have generated high-quality solutions. In Section 2.2, we review six algorithms that have been used by researchers to solve the large-scale problems, develop an improved version of our record-to-record travel algorithm that uses a variable-length neighbor list, and report computational results for all seven procedures.

In Section 2.3, we develop a new set of 12 very large-scale vehicle routing problems (denoted by VLSVRPs). The problems have 560 to 1,200 customers, route-length restrictions, and exhibit geometric symmetry. We report computational experience with our variable-length neighbor list record-to-record travel algorithm and compare results to the visually estimated solutions. In Section 2.4, we summarize our findings and suggest areas for future work.

2.2 Review of solution approaches to the LSVRP

In the last five years, a variety of algorithms have been developed to solve the 20 LSVRPs. Researchers have used deterministic variants of simulated annealing (record-to-record travel, backtracking adaptive threshold accepting, and list-based threshold accepting) and variants of tabu search (network flow-based tabu search, adaptive memory-based tabu search, and granular tabu search). We summarize these six algorithms in Table 2.1.

We set out to develop an improved version of the record-to-record travel algorithm (denoted by RTR) described in Golden et al. [21] that would be accurate, fast, simple, and flexible (this was motivated by the work of Cordeau et

Table 2.1: Six algorithms for solving the large-scale vehicle routing problem.

Authors	Algorithm	Comments
Golden et al. [21]	Record-to-record travel	An initial solution is generated by the Clarke and Wright algorithm [11]. Feasible one-point moves are made using record-to-record travel (uphill moves allowed). Points are exchanged on different routes (two-point exchange) while feasibility is maintained (uphill moves allowed). Routes are cleaned up (only downhill moves allowed). A local re-initialization allows individual routes to be resequenced and the process of one-point moves, two-point exchanges, and clean-up is repeated. At the end, global re-initialization perturbs the best solution and the process of one-point moves, two-point exchanges, and clean-up is repeated.
Golden et al. [21]	Network flow-based tabu search	The authors used the algorithm of Xu and Kelly. [48] where insertion and swap moves are controlled by a network flow model. Infeasible solutions that violate capacity are allowed
Tarantilis and Kiranoudis [43]	Adaptive memory-based tabu search	The key idea is to extract a sequence of points (called bones) from a set of solutions and generate a route using adaptive memory. If a large number of routes in the set of solutions contains a specific bone, then the authors argue that this bone should be included in a route that appears in a high-quality solution. The BoneRoute algorithm has two phases. In Phase I, a set of initial solutions is generated using weighted savings. The solutions are improved using a standard tabu search algorithm. In Phase II, promising bones are extracted, a solution is generated and improved using tabu search, and the set of solutions is updated.
Toth and Vigo [47]	Granular tabu search	The authors define a granular neighborhood for the VRP by considering short edges (these are edges whose lengths are less than a threshold value) and by typically not considering long edges. Granular neighborhoods are similar in concept to neighbor list strategies. A granular tabu search algorithm is developed. The algorithm starts with a Clarke and Wright solution (infeasible solutions are allowed) and uses granularity-based intensification and diversification during the search.
Tarantilis [39]	Backtracking adaptive threshold accepting List-based threshold accepting	Threshold accepting is a deterministic variant of simulated annealing in which a threshold value T is specified as the upper bound on the amount of objective function increase allowed (uphill moves can be made). In the backtracking algorithm, T is allowed to increase during the search. In the list-based algorithm, a list of values for T is used during the search.

al. [12]). The details of our improved algorithm are given in Table 2.2. Our algorithm (denoted by VRTR) uses a variable-length neighbor list. The idea is to consider only a fixed number of neighbors for each node when making one-point, two-point, and two-opt moves. In Figure 2.1, we show one-point moves. The black node is inserted into another place on the same route or on a different route. In Figure 2.2, we show two-point moves. Two black nodes are exchanged on the same route or on different routes. In Figure 2.3, we show two-opt moves. A two-opt move consists of deleting two edges and then connecting the two paths in a different way. In the top portion of Figure 2.3, edges ij and $k\ell$ are deleted and replaced by $i\ell$ and kj and this provides a better solution. In the bottom portion of Figure 2.3, $i\ell$ and jk are replaced by ij and $k\ell$.

We start with a traditional fixed-length neighbor list with $k = 40$ nearest neighbors (fixed-length neighbor lists are not new; this type of implementation was used in solving the traveling salesman problem (Coy et al. [13]). For each node i, we remove all edges (for nodes in the neighbor list) with length greater than $\alpha \times L$, where L is the maximum length among all edges in i's neighbor list. When α is equal to 1, we consider $k = 40$ edges. When α is less than 1, we consider fewer edges. In general, we expect that as α decreases, so does running time and accuracy suffers. Our variable-length neighbor list is similar to the granular neighborhood developed by Toth and Vigo [47] for the VRP.

We run our record-to-record travel algorithm three times with different starting solutions that have been generated by the modified Clarke and Wright algorithm with parameter λ. The three values for λ that we use in our algorithm (see Step 0 in Table 2.2) were determined by a search over values ranging from 0.6 to 2.0 that we conducted in preliminary computational experiments. The values 0.6, 1.4, and 1.6 generated reasonably good results.

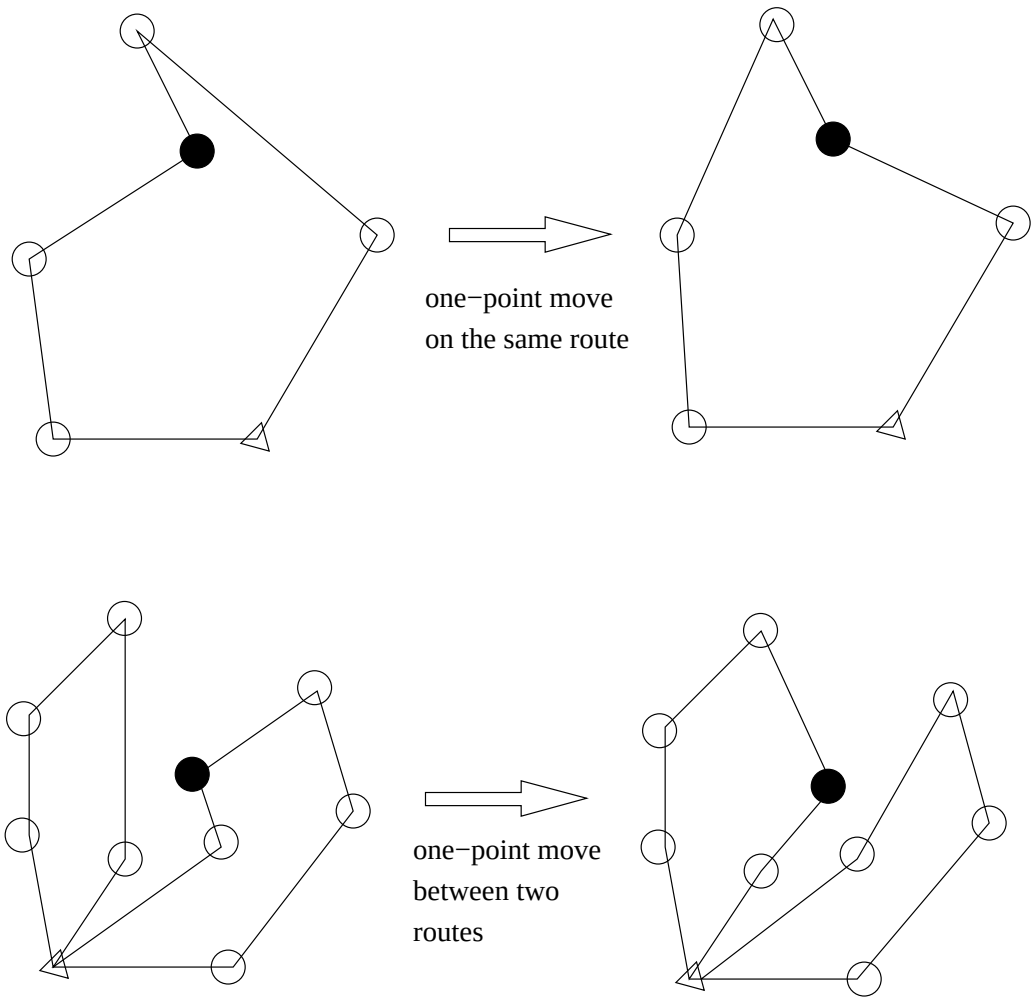


Figure 2.1: One-point move within and between routes.

There are two key differences between VRTR and RTR. First, VRTR considers two-opt moves between and within routes; RTR considers two-opt moves only within routes. Second, VRTR uses a variable-length neighbor list that should help focus the algorithm on promising moves and speed up the search procedure; RTR does not use a neighbor list.

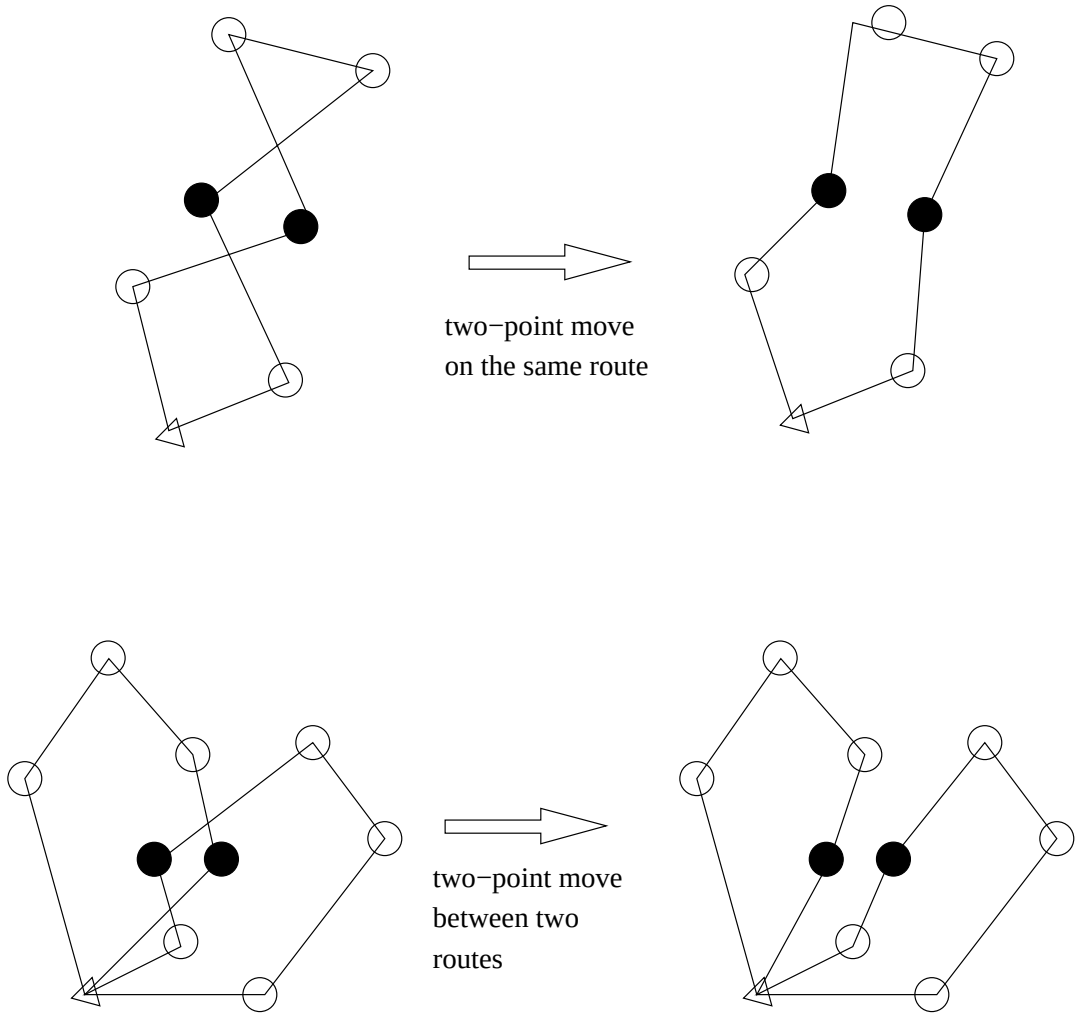


Figure 2.2: Two-point move within and between routes.

Table 2.2: Variable-length neighbor list record-to-record travel algorithm.

Step 0. Initialization.
Parameters are I, K and λ.
Set $I = 30, K = 5$, and $\lambda \in \{0.6, 1.4, 1.6\}$.
Step 1. Starting solution.
Generate an initial feasible solution using the modified Clarke and Wright algorithm with parameter λ.
Set Record = objective function value of the current solution.
Set Deviation = $0.01 \times$ Record.
Step 2. Improve the current solution.
For $i = 1$ to I (I loop)

Table 2.2: (continued)

Do One-Point move with record-to-record travel, Two-Point move with record-to-record travel between routes, and Two-opt move with record-to-record travel. Feasibility must be maintained.
If no feasible record-to-record travel move is made, go to Step 3.
If a new record is produced, update Record and Deviation.
End I loop

Step 3. For the current solution, apply One-Point move (within and between routes), Two-Point move (between routes), and Two-opt move (within and between routes). Only downhill moves are allowed.
If a new record is produced, update Record and Deviation.

Step 4. Repeat for K consecutive iterations.
If no new record is produced, go to Step 5.
Otherwise go to Step 2.

Step 5. Perturb the solution.
Compare the solution generated after perturbation to the best solution generated so far and keep the best solution.

Step 6. Keep the best solution generated so far.
Go to Step 1 and select a new value for λ.

One-point move with record-to-record travel

For each node i in the current solution (I loop)
For each edge j in the current solution whose one end is in node i's neighbor list (J loop)
Calculate the savings of inserting node i between edge j if such a move is feasible.
If the savings is greater than or equal to zero, then make the move and continue with the I loop.
Store the value of the largest savings so far.
If the tour length - largest savings from J loop $\leq$ Record + Deviation, then make the move and continue with the I loop.
End J loop
End I loop

Two-point move with record-to-record travel

For each node i in the current solution (I loop)
For each node j in the neighbor list of node i (J loop)
Calculate the savings of exchanging i and j.
If the savings is greater than or equal to zero, then make the exchange and go to the I loop.
Store the value of the largest savings so far.
If the tour length - largest savings from the J loop $\leq$ Record + Deviation, then make the exchange and continue with the I loop.
End J loop

Table 2.2: (continued)

End I loop

Two-opt move with record-to-record travel
For each edge i in the current solution (I loop)
For each node j from the neighbor list of node i (J loop)
Calculate the savings of the two-opt move with i and j if the move is feasible, that is, both capacity and distance constraints are satisfied.
If the savings is greater than or equal to zero, then make the move and go to I loop.
Store the value of the largest saving so far.
If the tour length - largest savings from the J loop $\leq$ Record + Deviation, then make the move and continue with the I loop.
End J loop
End I loop

Perturb a feasible solution
For each node i, define $r(i) = d(i)/s(i)$, where $d(i)$ is the demand of customer i and $s(i)$ =dist (prior(i),i) + dist(i, next(i)) - dist(prior(i),next(i)), where dist(i,j) is the distance from node i to node j, prior(i) is the node serviced before node i, and next(i) is the node serviced after node i.
Sort the $r(i)$ values in ascending order and select the first M nodes where $M = \min(20,\frac{n}{10})$; where n is the number of nodes. Try to insert these nodes, one by one, into a new location on a tour using least-cost insertion while maintaining feasibility.

In Table 2.3, we present the estimated solution values and solution values generated by seven algorithms on the 20 LSVRPs (we maintain the same ordering of the problems as given in Golden et al. [21]). The first eight problems (problems 1 to 8) have route-length restrictions.

By exploiting the geometric symmetry of a problem, a high-quality estimated solution can be produced visually. In the column labeled ESTG, we give the estimated solutions produced by Golden et al. [21]. Tarantilis and Kiranoudis [43] improved the estimated solutions for the first eight problems and these are given in the column labeled ESTTK.

The results for six algorithms (RTR, NFTS, BR, GTS, BATA, and LBTA)

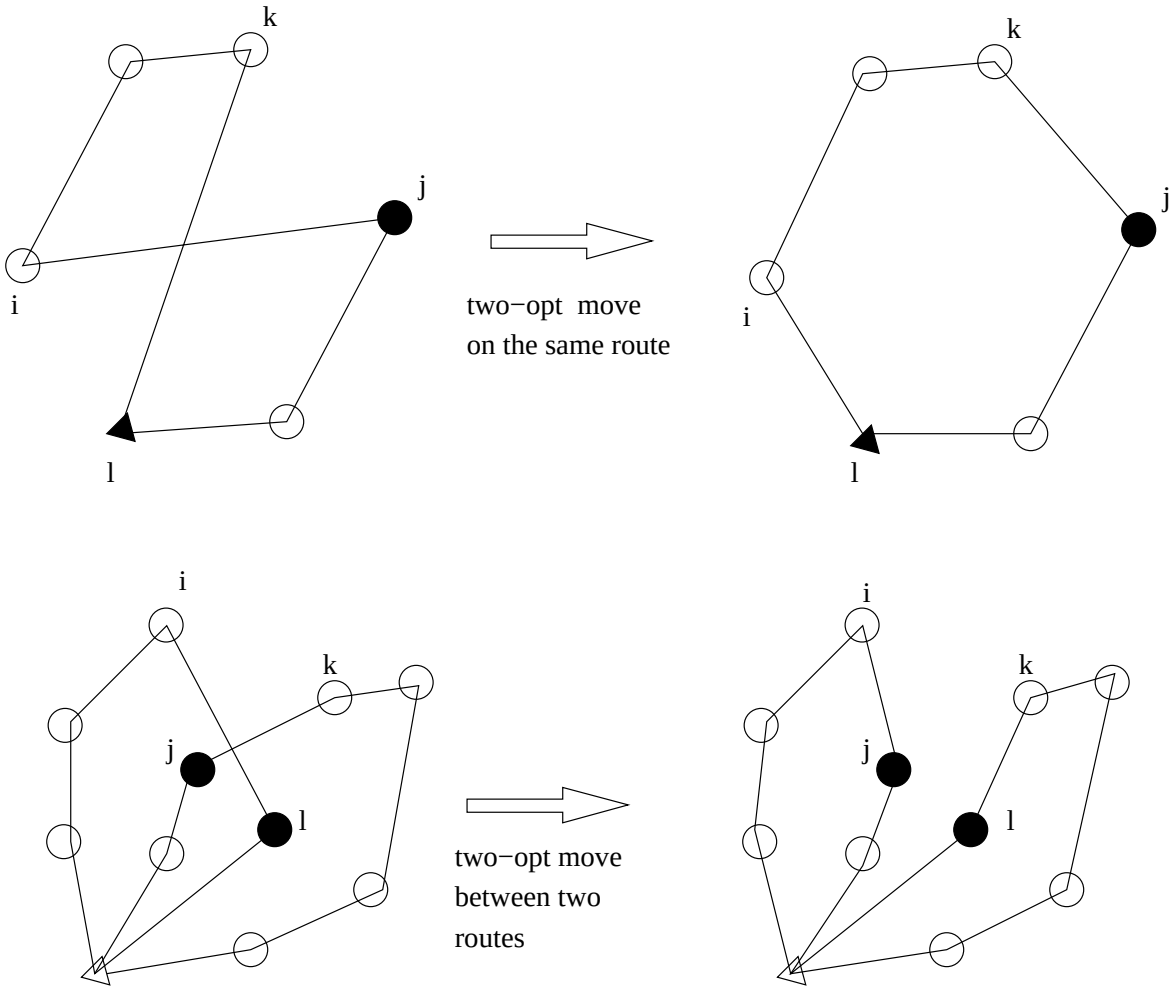


Figure 2.3: Two-opt move within and between routes.

are taken from the literature and represent solutions generated using a single set of parameter values. For example, there are seven parameters whose values need to be set in the BoneRoute algorithm (BR) and the authors use one standard setting for each parameter to produce the results shown in Table 2.3.

We experimented with a range of values from 0.40 to 1 for the α parameter in VRTR. In Table 2.3, we show results for the default value of 1 (use all edges in the neighbor list with $k = 40$; we point out that, when $\alpha = 0.40$, we use about 50-60% of the edges and when $\alpha = 0.60$, we use about 80-90% of the edges).

In Table 2.3, among the seven algorithms, VRTR performs best – it generates the best solution to nine problems. GTS is second best – it generates the best solution to two problems. The estimated solutions to problems 2 through 8 given by Tarantilis and Kiranoudis [43] are very good – no algorithm produced a better solution to these seven problems.

In Table 2.4, we compare the results of the seven algorithms on the 20 benchmark problems (we do not include the estimated solution values). VRTR performs the bset – it generates the best solution to 12 problems. BR generates the best solution to three problems, followed by LBTA and GTS – both generate the best solution to two problems.

In Table 2.5, we provide additional computational results on the 20 LSVRPs. In the columns marked Best Known and Source, we provide the best-known solution to each problem and the source of the solution. Six of the best-known solutions are visually estimated solutions (ESTTK), four are generated by VRTR with three different values (0.50, 0.60, 0.65) for the α parameter, and 10 are generated by RTR (the VRTR and RTR solutions were generated during the overall computational testing with a variety of settings for the parameters). We compute the percent above the best-known solution for each solution generated by two visual procedures (ESTG and ESTTK) and seven algorithms (RTR, NFTS, BR, GTS, BATA, LBTA, and VRTR with $\alpha = 1$, 0.60, and 0.40), and for the best solutions found by granular tabu search (these are given in the column marked BGTS; recall that GTS gives results using the same set of parameter values for each problem, while the parameter settings in BGTS can vary).

In Table 2.5, over all 20 problems, we see that VRTR with the default value of $\alpha = 1$ generates very high-quality solutions (on average within 0.70% of the best-known solution) in a small amount of computing time (on average 1.13

Table 2.3: Estimated solution values and solution values generated by seven algorithms on 20 LSVRPs.

Problem	n	ESTG	ESTTK	RTR	NFTS	BR	GTS	BATA	LBTA	VRTR
1	240	5859.62	5676.97	5834.60		5676.97	5736.15	5683.63	5680.16	**5666.42**
2	320	8566.04	**8447.92**	9002.26	8570.28	8512.64	8553.03	8528.80	8512.64	8469.32
3	400	11649.06	**11036.23**	11879.95	11880.37	11199.72	11402.75	11199.72	11190.38	11145.80
4	480	15108.68	**13624.53**	14639.32	15250.78	13637.53	14910.62	13661.16	13706.78	13758.08
5	200	8631.64	**6460.98**	6702.73	7361.29	**6460.98**	6697.53	6466.68	**6460.98**	6478.09
6	280	9843.01	**8412.88**	9016.93	9088.66	8429.28	8936.32	8429.28	8427.72	8539.61
7	360	11047.69	**10195.56**	11213.31	11411.85	10216.50	10547.44	10297.27	10274.19	10289.72
8	440	12250.06	**11828.78**	12514.20	12825.00	11936.16	12036.24	11953.93	11968.93	11920.52
9	255	678.82		**587.09**	589.10		593.35	596.92	595.35	588.25
10	323	865.50		749.15	**746.56**		751.66	765.03	764.88	749.49
11	399	1074.80		934.33	932.68		936.04	945.20	945.09	**925.91**
12	483	1306.73		1137.18	1140.72		1147.14	1143.39	1143.74	**1128.03**
13	252	956.04		881.04	881.07		868.80	872.66	871.97	**865.20**
14	320	1220.27		1103.69	1118.09		**1096.18**	1102.40	1102.66	1097.78
15	396	1516.48		1364.23	1377.79		1369.44	1384.04	1385.59	**1361.41**
16	480	1844.67		1657.93	1656.66		1652.32	1679.50	1677.25	**1635.58**
17	240	796.13		720.44			**711.07**	718.16	717.40	711.74
18	300	1133.25		1029.21			1016.83	1030.54	1032.07	**1010.32**
19	360	1554.64		1403.05			1400.96	1408.62	1406.47	**1382.59**
20	420	2081.38		1875.17			1915.83	1872.23	1872.87	**1850.92**

ESTG, estimated solution from Golden et al. [21].
ESTTK, estimated solution from Tarantilis and Kiranoudis [43].
RTR, record-to-record travel solution from Golden et al. [21].
NFTS, network flow-based tabu search solution of Xu and Kelly from Golden et al. [21].
BR, BoneRoute solution from Tarantilis and Kiranoudis [43].
GTS, granular tabu search solution from Toth and Vigo [47].
BATA, backtracking adaptive threshold accepting solution from Tarantilis [39].
LBTA, list-based threshold accepting solution from Tarantilis [39].
VRTR, variable-length neighbor list record-to-record travel solution (α=1).
Bold, best solution.

Table 2.4: Solution values generated by seven algorithms on 20 LSVRPs.

Problem	n	RTR	NFTS	BR	GTS	BATA	LBTA	VRTR
1	240	5834.60		5676.97	5736.15	5683.63	5680.16	**5666.42**
2	320	9002.26	8570.28	8512.64	8553.03	8528.80	8512.64	**8469.32**
3	400	11879.95	11880.37	11199.72	11402.75	11199.72	11190.38	**11145.80**
4	480	14639.32	15250.78	**13637.53**	14910.62	13661.16	13706.78	13758.08
5	200	6702.73	7361.29	**6460.98**	6697.53	6466.68	**6460.98**	6478.09
6	280	9016.93	9088.66	8429.28	8936.32	8429.28	**8427.72**	8539.61
7	360	11213.31	11411.85	**10216.50**	10547.44	10297.27	10274.19	10289.72
8	440	12514.20	12825.00	11936.16	12036.24	11953.93	11968.93	**11920.52**
9	255	**587.09**	589.10		593.35	596.92	595.35	588.25
10	323	749.15	**746.56**		751.66	765.03	764.88	749.49
11	399	934.33	932.68		936.04	945.20	945.09	**925.91**
12	483	1137.18	1140.72		1147.14	1143.39	1143.74	**1128.03**
13	252	881.04	881.07		868.80	872.66	871.97	**865.20**
14	320	1103.69	1118.09		**1096.18**	1102.40	1102.66	1097.78
15	396	1364.23	1377.79		1369.44	1384.04	1385.59	**1361.41**
16	480	1657.93	1656.66		1652.32	1679.50	1677.25	**1635.58**
17	240	720.44			**711.07**	718.16	717.40	711.74
18	300	1029.21			1016.83	1030.54	1032.07	**1010.32**
19	360	1403.05			1400.96	1408.62	1406.47	**1382.59**
20	420	1875.17			1915.83	1872.23	1872.87	**1850.92**

RTR, record-to-record travel solution from Golden et al. [21].
NFTS, network flow-based tabu search solution of Xu and Kelly from Golden et al. [21].
BR, BoneRoute solution from Tarantilis and Kiranoudis [43].
GTS, granular tabu search solution from Toth and Vigo [47].
BATA, backtracking adaptive threshold accepting solution from Tarantilis [39].
LBTA, list-based threshold accepting solution from Tarantilis [39].
VRTR, variable-length neighbor list record-to-record travel solution (α=1).
Bold, best solution.

minutes per problem) and clearly outperforms RTR, GTS, BGTS, BATA, and LBTA. VRTR with smaller values of the α parameter (0.60 and 0.40) also produces very high-quality solutions (on average within 0.69% and 0.77% of the best-known solution, respectively) very quickly (average of 0.97 minutes and 0.68 minutes per problem, respectively).

In Figure 2.4 to Figure 2.19, we provide the "old" estimated solutions and our new improved estimated solutions for problems 1 to 8. The "old" estimated solutions are taken from Chao [9]. The new estimated solutions were developed when we tried to reduce the length of each route of the "old" estimated solutions. In Figure 2.5, we were unable to generate an estimated solution that has the same tour length as ESTTK.

In Figure 2.20 to Figure 2.39, we provide the best solution generated by VRTR to each of the 20 problems given in Table 2.3.

2.3 Very large-scale vehicle routing: New problems and results

We have generated a new set of 12 very large-scale vehicle routing problems with route-length restrictions. These problems have 560 to 1,200 customers. The problem generator is given in Section 2.5. Each problem exhibits a geometric symmetry that allows us to visually estimate a high-quality solution.

In Table 2.6, we present the estimated solution values and the solution values generated by VRTR for three values of the α parameter (1, 0.60, 0.40). In addition, we show the percent above the best-known solution, and the average computing time. Over all 12 problems, we see that VRTR with the default value of $\alpha = 1$ generates very high-quality solutions (on average within 1.10% of the best-known solution) in a small amount of computing time (on average 3.16

Table 2.5: Computational results on 20 LSVRPs: Percent above best-known solution and computing time.

Problem	n	Best known	Source	ESTG	ESTTK	RTR	NTFS	BR	GTS	BGTS	BATA	LBTA	VRTR		
													$\alpha = 1$	$\alpha = 0.6$	$\alpha = 0.4$
1	240	5639.36	VRTR, α=.65	3.91	0.67	3.46		0.67	1.72	1.72	0.79	0.72	0.48	0.65	0.60
2	320	8447.92	ESTTK	1.40	0.00	6.56	1.45	0.77	1.24	1.24	0.96	0.77	0.25	0.39	0.39
3	400	11036.23	ESTTK	5.55	0.00	7.65	7.65	1.48	3.32	3.32	1.48	1.40	0.99	0.14	1.42
4	480	13624.53	ESTTK	10.89	0.00	7.45	11.94	0.10	9.44	5.22	0.27	0.60	0.98	1.28	1.03
5	200	6460.98	ESTTK	33.60	0.00	3.74	13.93	0.00	3.66	3.40	0.09	0.00	0.26	0.26	0.35
6	280	8412.88	ESTTK	17.00	0.00	7.18	8.03	0.19	6.54	3.56	0.19	0.18	1.51	1.48	1.50
7	360	10195.56	ESTTK	8.36	0.00	9.98	11.93	0.21	3.45	3.14	1.00	0.77	0.92	0.98	0.52
8	440	11696.55	ORTR	4.73	1.13	6.99	9.65	2.05	2.90	2.90	2.20	2.33	1.91	2.13	0.65
9	255	585.54	ORTR	15.93		0.26	0.61		1.33	1.33	1.94	1.68	0.46	0.80	0.55
10	323	743.17	ORTR	16.46		0.80	0.46		1.14	1.14	2.94	2.92	0.85	0.78	0.78
11	399	923.11	ORTR	16.43		1.22	1.04		1.40	1.40	2.39	2.38	0.30	0.31	1.11
12	483	1116.22	VRTR, α=.65	17.07		1.88	2.19		2.77	1.78	2.43	2.47	1.06	0.86	1.07
13	252	862.38	ORTR	10.86		2.16	2.17		0.74	0.73	1.19	1.11	0.33	0.98	1.41
14	320	1083.65	EST	12.61		1.85	3.18		1.16	1.16	1.73	1.75	1.30	0.98	1.11
15	396	1351.35	ORTR	12.22		0.95	1.96		1.34	0.89	2.42	2.53	0.74	0.24	0.24
16	480	1632.47	ORTR	13.00		1.56	1.48		1.22	1.10	2.88	2.74	0.19	0.51	0.66
17	240	708.63	ORTR	12.35		1.67			0.34	0.18	1.34	1.24	0.44	0.44	0.59
18	300	1007.86	ORTR	12.44		2.12			0.89	0.69	2.25	2.40	0.24	0.24	0.24
19	360	1378.20	ORTR	12.80		1.80			1.65	0.79	2.21	2.05	0.32	0.32	0.41
20	420	1840.66	VRTR, α=.60	13.08		1.87			4.08	4.08	1.72	1.75	0.56	0.00	0.80
Average percent above best-known solution				12.53	0.22	3.56	5.18	0.68	2.52	1.99	1.62	1.59	0.70	0.69	0.77
Average computing time (min)						37.15	1825.59	42.05	17.55		18.41	17.81	1.13	0.97	0.68

EST, estimated solution by Li, Golden, and Wasil.
ESTG, estimated solution from Golden et al. [21].
ESTTK, estimated solution from Tarantilis and Kiranoudis [43].
RTR, record-to-record travel solution from Golden et al. [21]; Pentium 100 MHz CPU.
NFTS, network flow-based tabu search solution of Xu and Kelly from Golden et al.[21]; DEC Alpha Workstation.
BR, BoneRoute solution from Tarantilis and Kirnoudis [43]; Pentium 400 MHz CPU.
GTS, granular tabu search solution from Toth and Vigo [47]; Pentium 200 MHz CPU.
BGTS, best solution found by granular tabu search over all testing from Toth and Vigo [6].
BATA, backtracking adaptive threshold accepting solution from Tarantilis [39]; Pentium 233 MHz CPU.
LBTA, list-based threshold accepting solution from Tarantilis [39]; Pentium 233 MHz CPU.
VRTR, variable-length neighbor list record-to-record travel solution; Athlon 1 GHz CPU.
ORTR, record-to-record travel solution from other experiments by Li, Golden, and Wasil.

minutes per problem). VRTR with a value of $\alpha = 0.60$ also produces very high-quality solutions (on average within 1.20% of the best-known solution) very quickly (on average 2.97 minutes per problem). VRTR with a value of $\alpha = 0.40$ is very fast (on average 2.08 minutes per problem), but not highly accurate (on average within 2.28% of the best-known solution) especially on the three largest problems.

In Figure 2.40 to Figure 2.75, we provide the estimated solution, one route, and the best solution produced by VRTR to each of the 12 problems given in Table 2.6.

2.4 Conclusions

In this chapter, we reviewed procedures for solving large-scale vehicle routing problems and developed a record-to-record travel algorithm that uses a variable-length neighbor list. Our algorithm was very fast and highly accurate in solving 20 LSVRPs. We generated a new set of 12 very large-scale VRPs with 560 to 1,200 customers that are some of the largest test instances in the literature. A solution to each problem can be estimated visually. We applied our record-to-record travel algorithm to these 12 VLSVRPs and found that it produced solutions quickly and accurately. Finally, we applied VRTR without any additional fine-tuning using the three values of the α parameter (1, 0.60, 0.40) to the seven, small-scale benchmark VRPs of Christofides et al. [10]. These seven problems do not have service times for customers. In Table 2.7, we present the solution values generated by VRTR and the granular tabu search procedure (GTS) of Toth and Vigo [47]. In addition, we show the average percent above the best-known solution, and the average computing time. Over all seven small-scale

Table 2.6: Computational results on 12 VLSVRPs: Solution value, percent above best-known solution, and computing time.

				Solution Value				Percent above best-known solution			
					VRTR				VRTR		
Problem	n	Best Known	Source	EST	$\alpha = 1$	$\alpha = 0.6$	$\alpha = 0.4$	EST	$\alpha = 1$	$\alpha = 0.6$	$\alpha = 0.4$
21	560	**16212.83**	EST	16212.83	16602.99	16627.22	16739.25	0.00	2.41	2.56	3.25
22	600	**14641.64**	ORTR	14652.28	14651.27	14655.60	14669.39	0.07	0.07	0.10	0.19
23	640	**18801.13**	EST	18801.13	19005.37	19094.98	18838.62	0.00	1.09	1.56	0.20
24	720	**21389.43**	EST	21389.43	21784.43	21616.25	21932.57	0.00	1.85	1.06	2.54
25	760	**17053.26**	EST	17053.26	17151.43	17163.31	17146.41	0.00	0.58	0.65	0.55
26	800	**23977.74**	EST	23977.74	24189.66	24200.27	24009.74	0.00	0.88	0.93	0.13
27	840	**17651.60**	ORTR	18253.55	17823.40	17936.25	17901.56	3.41	0.97	1.61	1.42
28	880	**26566.04**	EST	26566.04	26606.11	26784.38	26787.38	0.00	0.15	0.82	0.83
29	960	**29154.34**	EST	29154.34	29181.21	29183.58	29401.90	0.00	0.09	0.10	0.85
30	1040	**31742.64**	EST	31742.64	31976.73	31961.58	33246.60	0.00	0.74	0.69	4.74
31	1120	**34330.94**	EST	34330.94	35369.17	35355.50	36339.65	0.00	3.02	2.98	5.85
32	1200	**36919.24**	EST	36919.24	37421.44	37410.84	39415.85	0.00	1.36	1.33	6.76
Average percent above best-known solution								0.29	1.10	1.20	2.28
Average computing time (min)									3.16	2.94	2.08

EST, estimated solution by Li, Golden, and Wasil.
VRTR, variable-length neighbor list record-to-record travel solution; Athlon 1 GHz CPU.
ORTR, record-to-record travel solution from other experiments by Li, Golden, and Wasil.

Table 2.7: Solutions generated by VRTR and GTS on seven small-scale VRPs.

			VRTR			
Problem	n	Best Known	$\alpha = 1$	$\alpha = 0.6$	$\alpha = 0.4$	GTS
1	50	**524.61**	**524.61**	**524.61**	**524.61**	**524.61**
2	75	**835.26**	846.64	838.60	836.18	838.60
3	100	**826.14**	**826.14**	**826.14**	827.39	828.56
4	150	**1028.42**	1036.00	1044.36	1045.36	1033.21
5	199	**1291.45**	1318.70	1322.56	1303.47	1318.25
11	120	**1042.11**	1043.50	**1042.11**	**1042.11**	1042.87
12	100	**819.56**	**819.56**	**819.56**	**819.56**	**819.56**
Average percent above best-known solution			0.62	0.62	0.41	0.47
Average computing time (min)			0.41	0.35	0.32	3.10

Best known, best-known solutions from Golden et al. [21] and Gendreau et al. [18].
VRTR, variable-length neighbor list record-to-record travel solutions; Athlon 1 GHz CPU.
GTS, granular tabu search solution from Toth and Vigo [46]; Pentium 200 MHz CPU.
Bold, best-known solution.

problems, we see that VRTR with the default value of $\alpha = 1$ generates very high-quality solutions (on average within 0.62% of the best-known solution) in a small amount of computing time (on average 0.41 minutes per problem). All three versions of VRTR are quick and accurate, and very competitive with GTS.

2.5 VLSVRP problem generator

$(x(i), y(i))$ is the coordinate of customer i, where $i = 0$ is the depot
$q(i)$ is the demand of customer i
A and B are parameters that determine the number of customers n, where $n = A \times B$
All data recorded to four decimal places

```
begin
      ω = 0
      x(ω) = 0, y(ω) = 0, q(ω) = 0
      for k := 1 to B do
      begin
   γ = 30k
   for i := 1 to A do
   begin
         ω = ω + 1
         x(ω) = γ cos [2(i − 1)π/A]
         y(ω) = γ sin [2(i − 1)π/A]
         if mod(i, 4) = 2 or 3
         then q(ω) = 30
         else q(ω) = 10
         end
   end
      end
end
```

Problem	A	B	n	Vehicle capacity	Maximum route-length
21	40	14	560	1200	1800
22	60	10	600	900	1000
23	40	16	640	1400	2200
24	40	18	720	1500	2400
25	76	10	760	900	900
26	40	20	800	1700	2500
27	84	10	840	900	900
28	40	22	880	1800	2800
29	40	24	960	2000	3000
30	40	26	1040	2100	3200
31	40	28	1120	2300	3500
32	40	30	1200	2500	3700

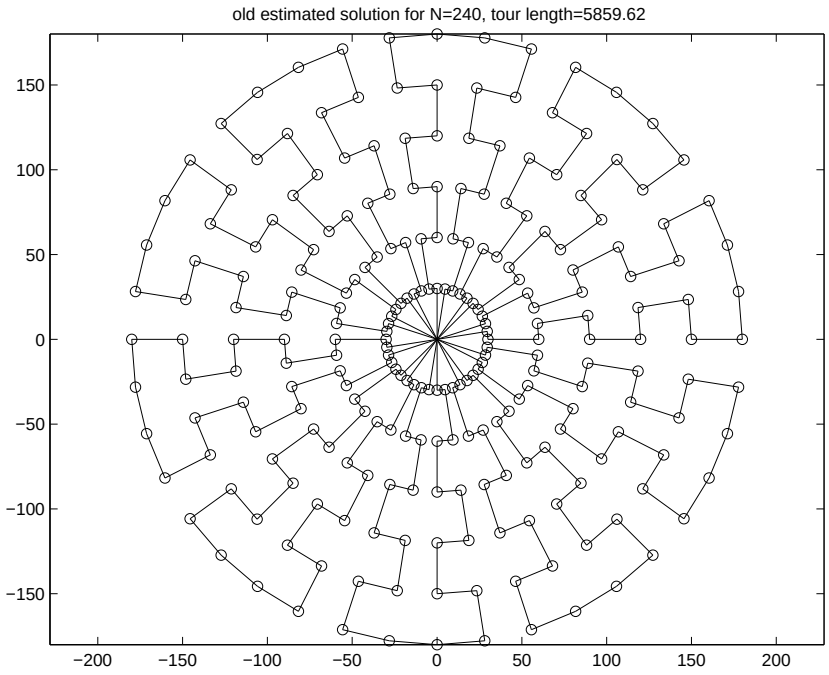


Figure 2.4: Old estimated solution for 240-node problem.

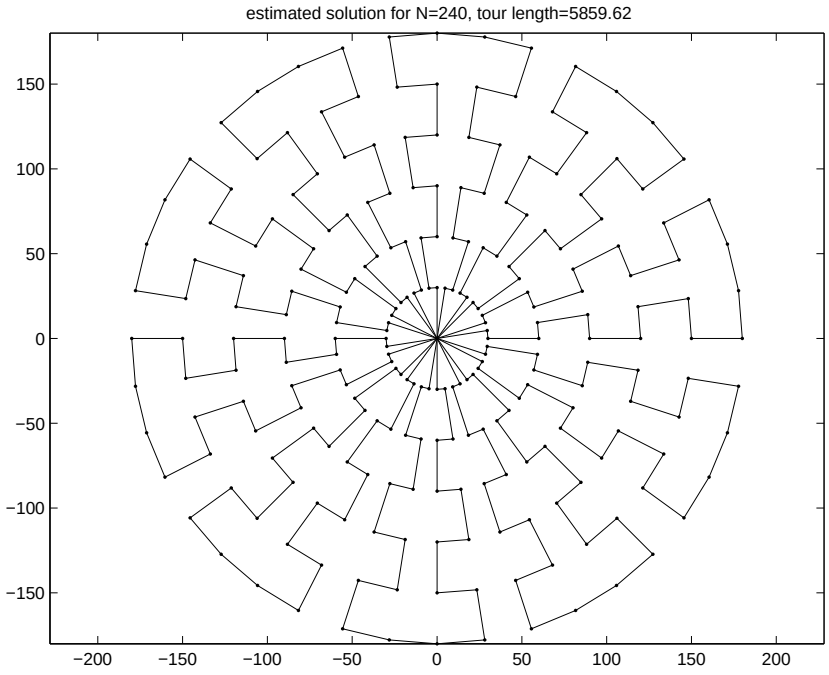


Figure 2.5: New estimated solution for 240-node problem.

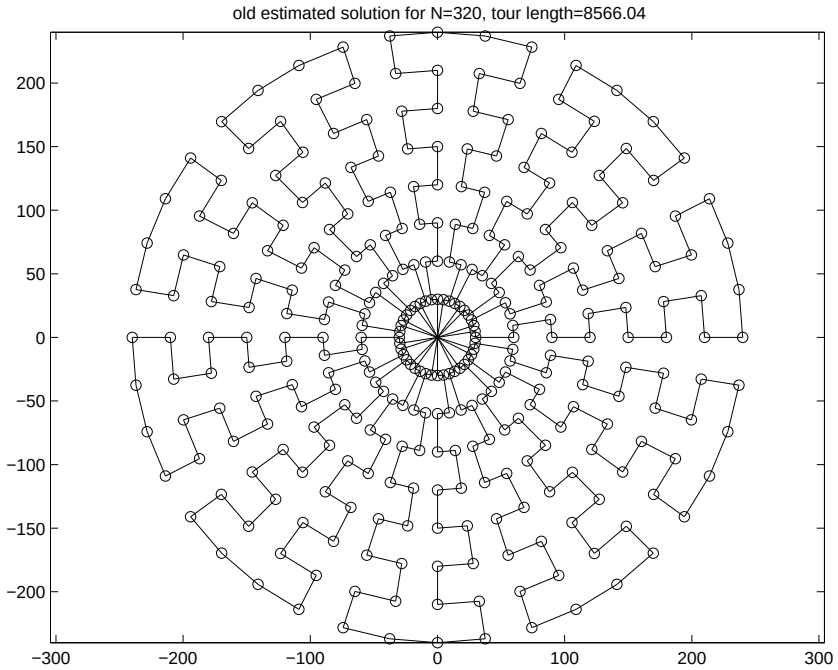


Figure 2.6: Old estimated solution for 320-node problem.

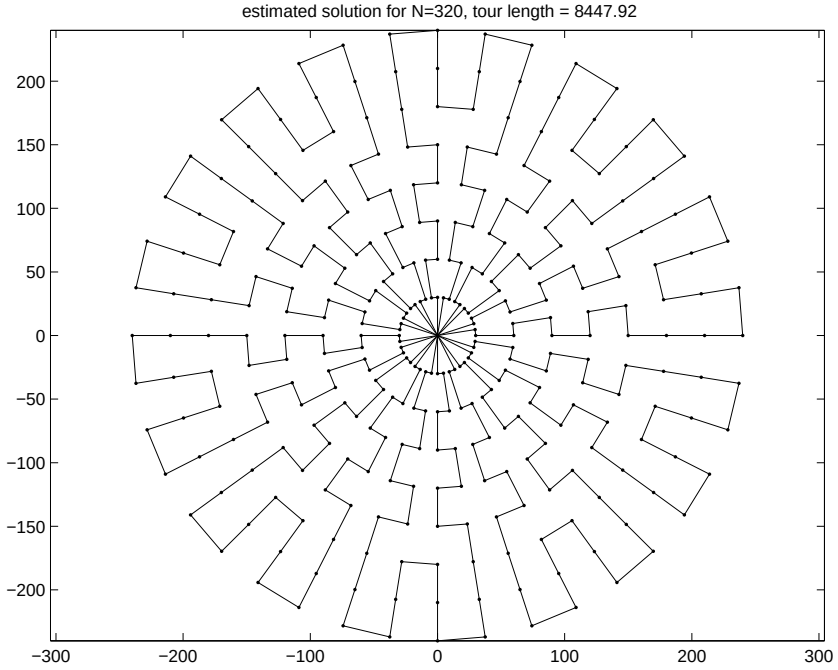


Figure 2.7: New estimated solution for 320-node problem.

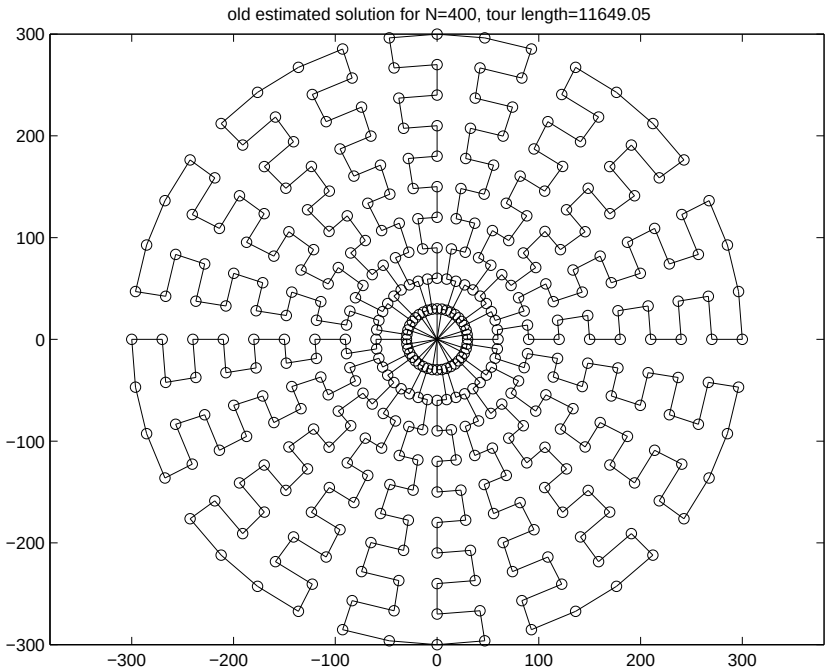


Figure 2.8: Old estimated solution for 400-node problem.

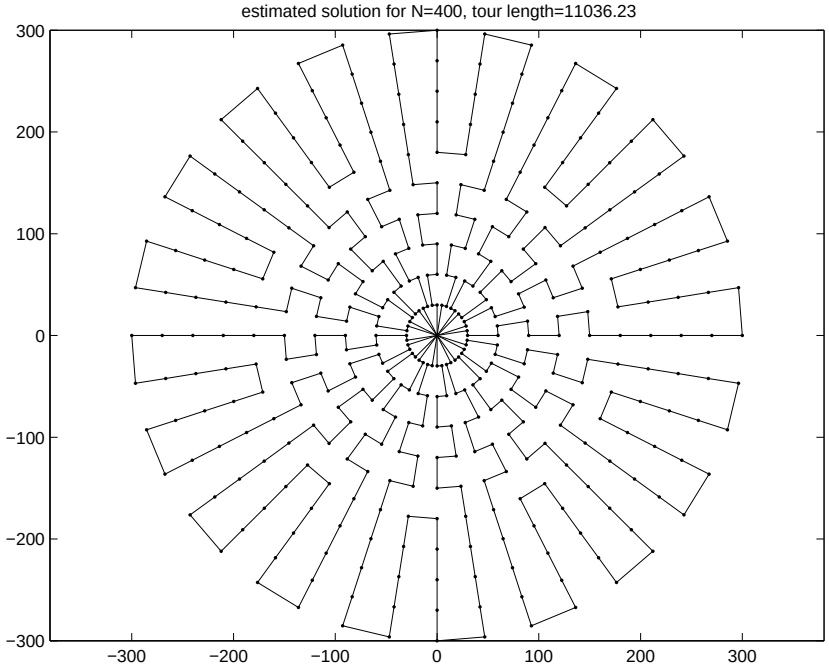


Figure 2.9: New estimated solution for 400-node problem.

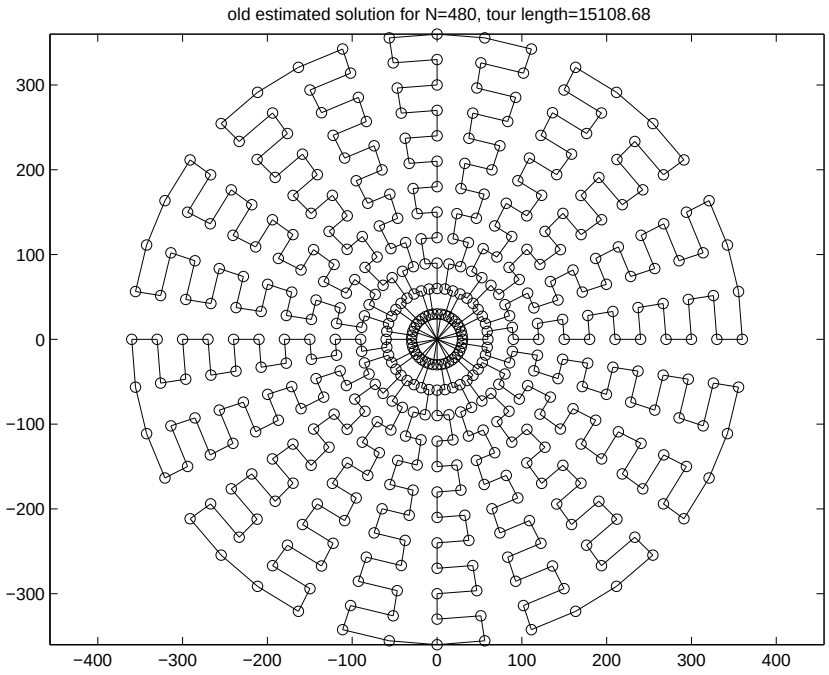


Figure 2.10: Old estimated solution for 480-node problem.

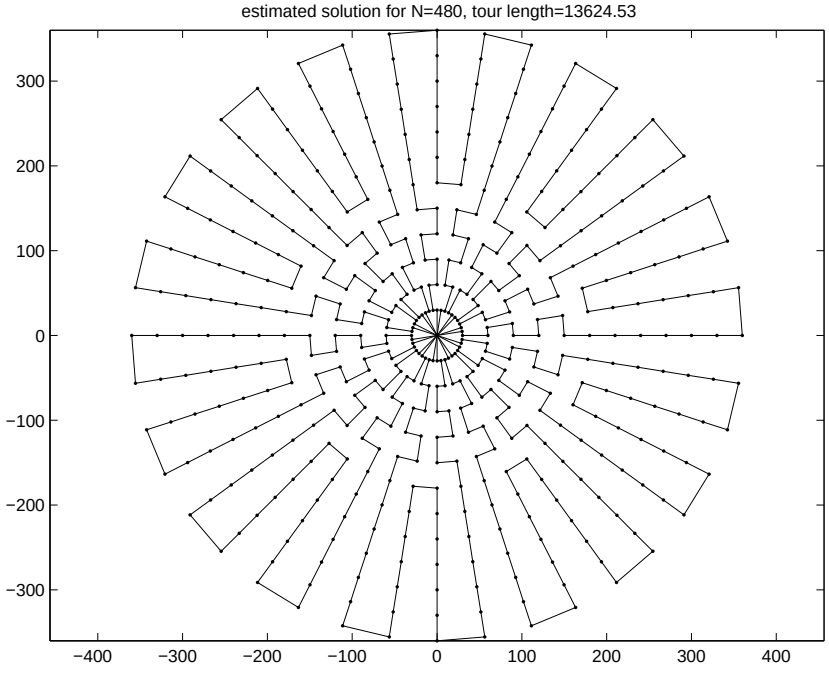


Figure 2.11: New estimated solution for 480-node problem.

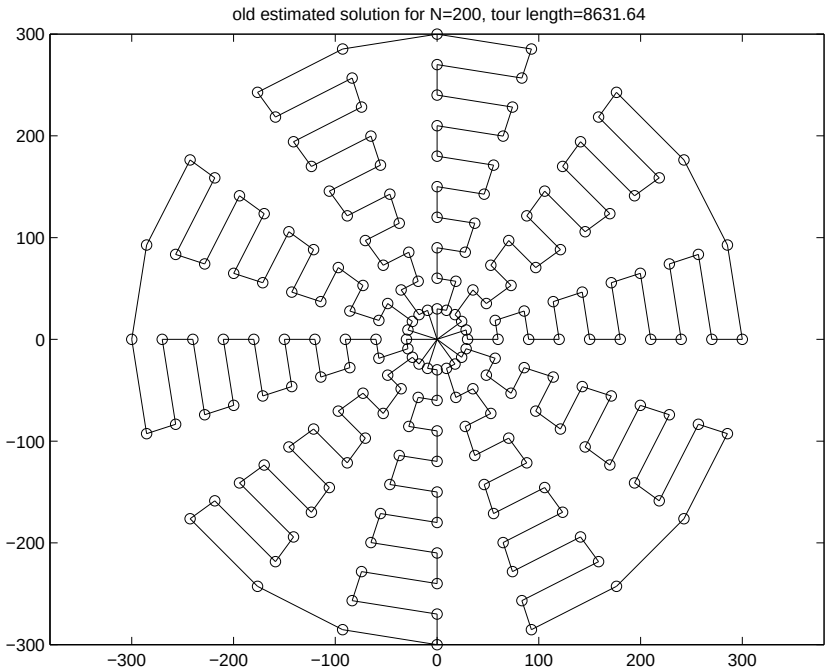


Figure 2.12: Old estimated solution for 200-node problem.

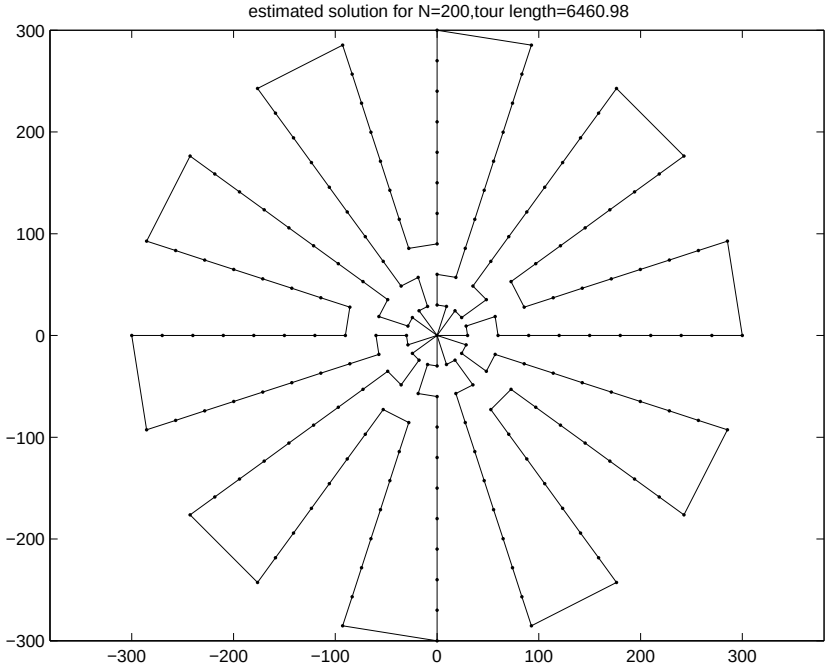


Figure 2.13: New estimated solution for 200-node problem.

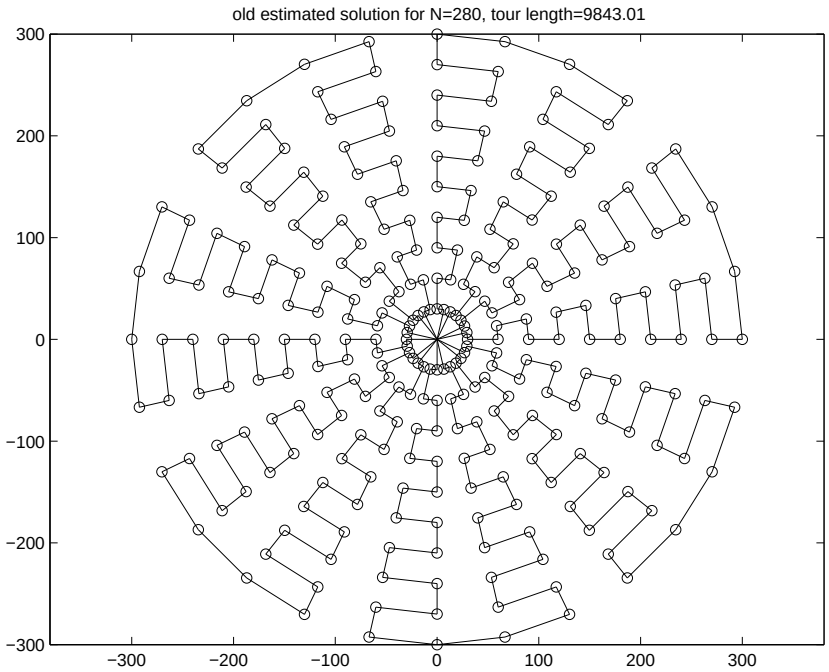


Figure 2.14: Old estimated solution for 280-node problem.

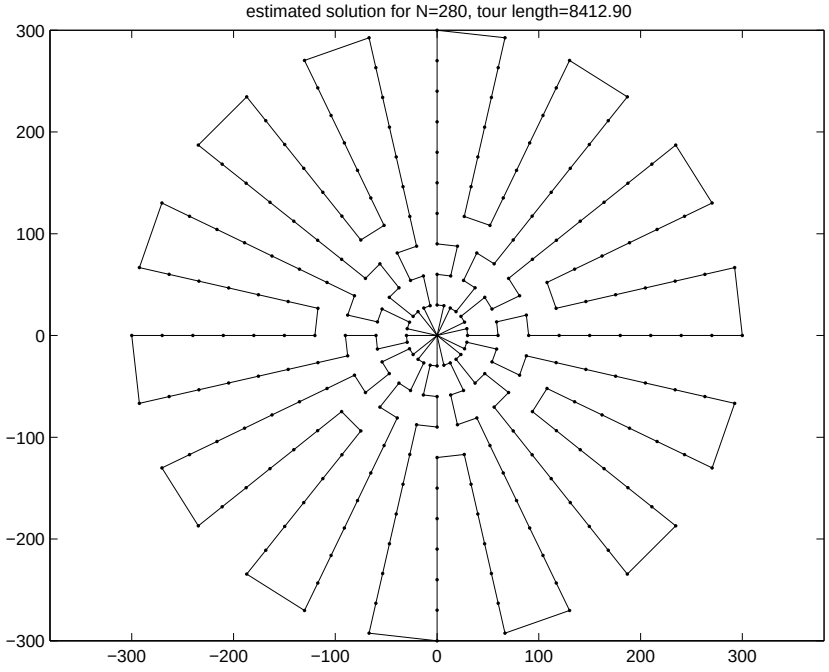


Figure 2.15: New estimated solution for 280-node problem.

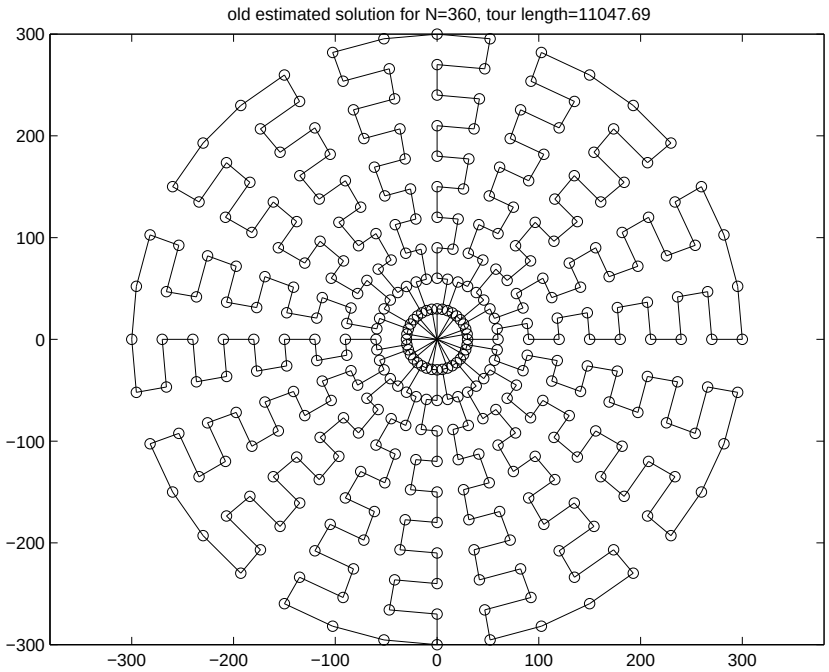


Figure 2.16: Old estimated solution for 360-node problem.

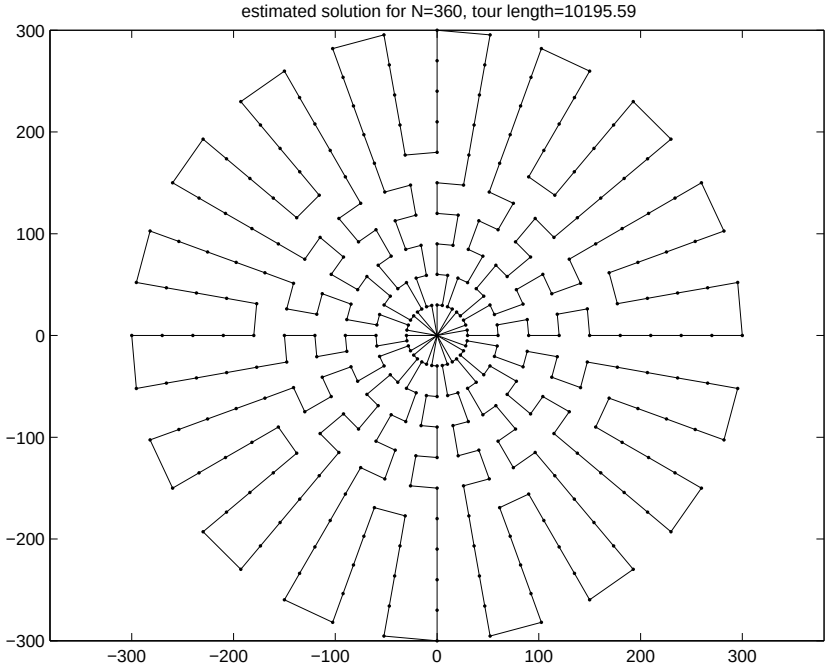


Figure 2.17: New estimated solution for 360-node problem.

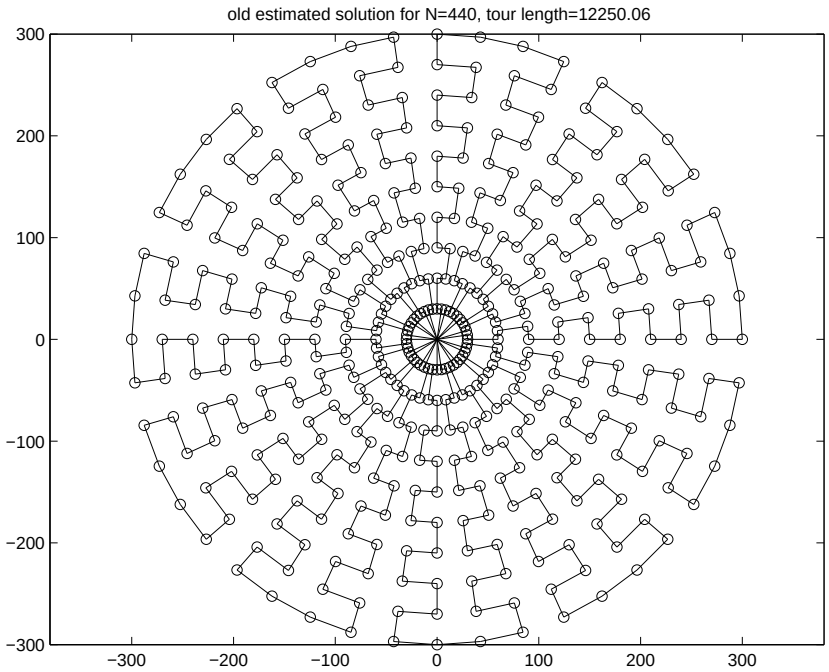


Figure 2.18: Old estimated solution for 440-node problem.

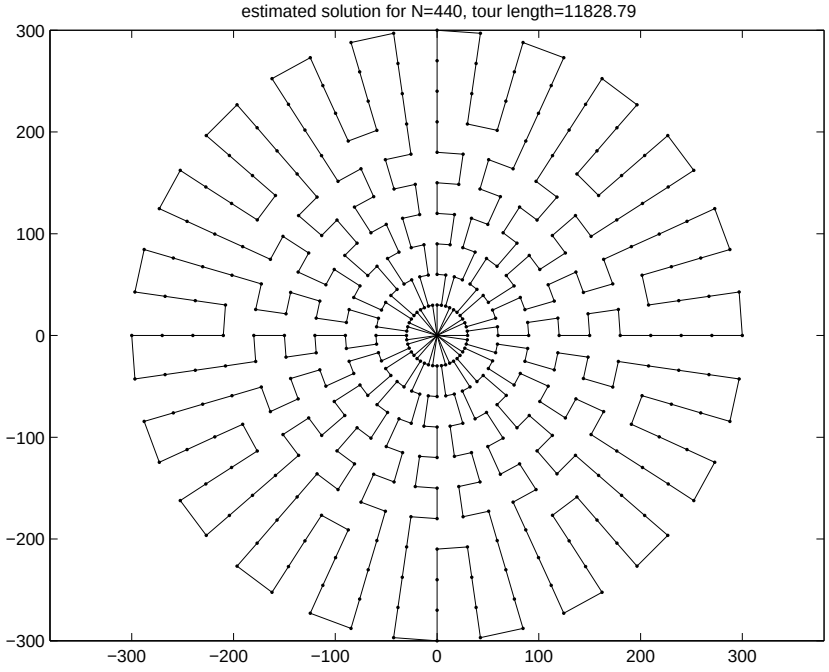


Figure 2.19: New estimated solution for 440-node problem.

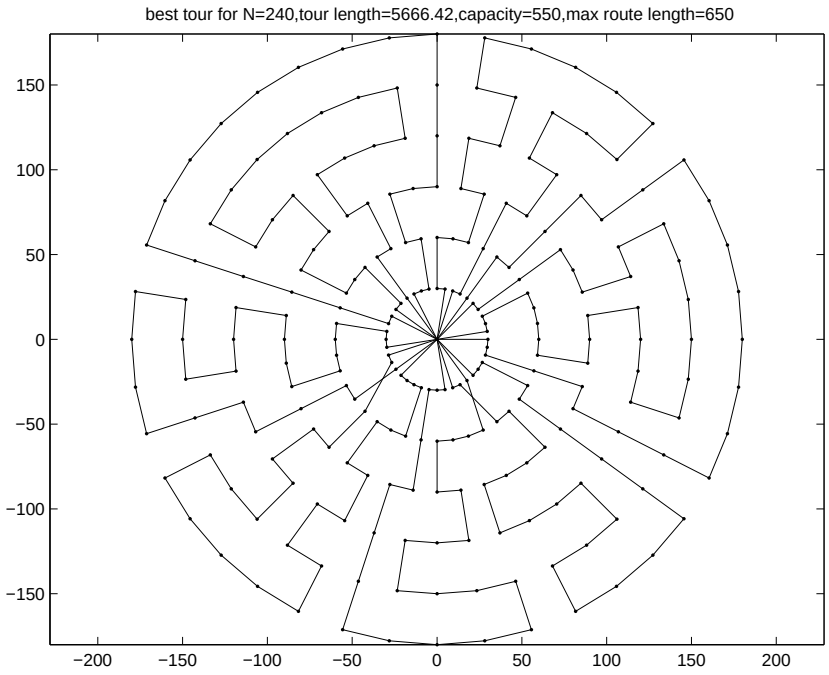


Figure 2.20: 240-node problem best VRTR tour.

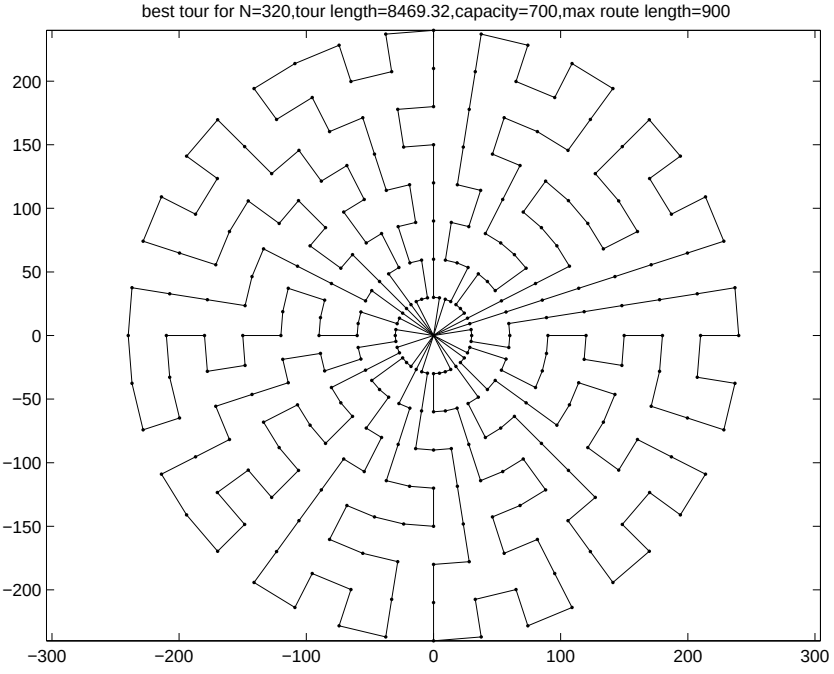


Figure 2.21: 320-node problem best VRTR tour.

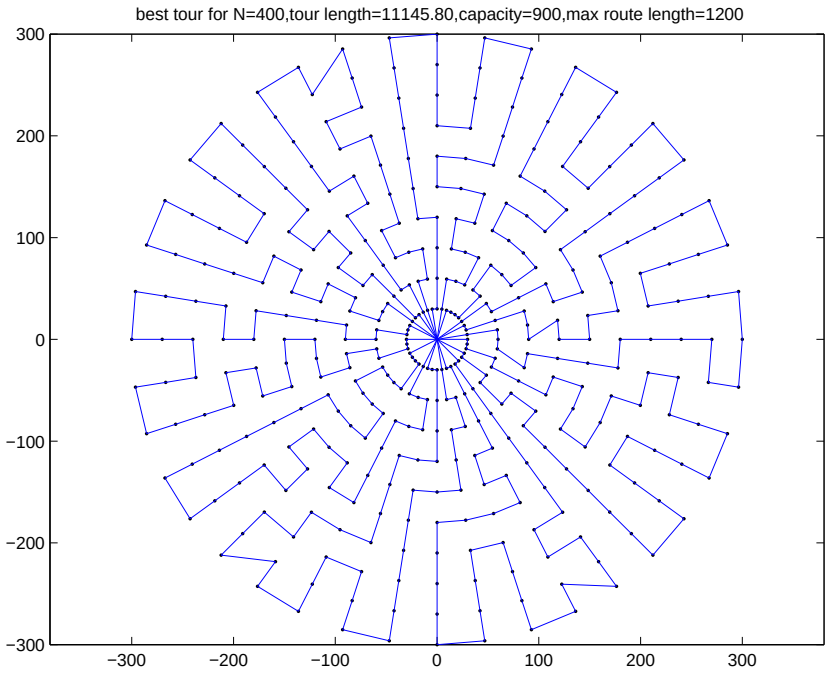


Figure 2.22: 400-node problem best VRTR tour.

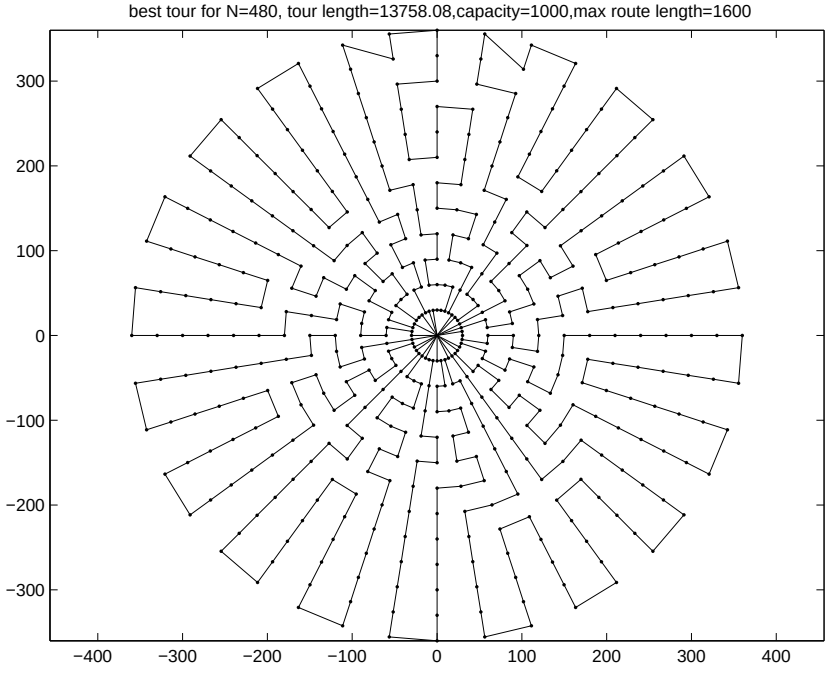


Figure 2.23: 480-node problem best VRTR tour.

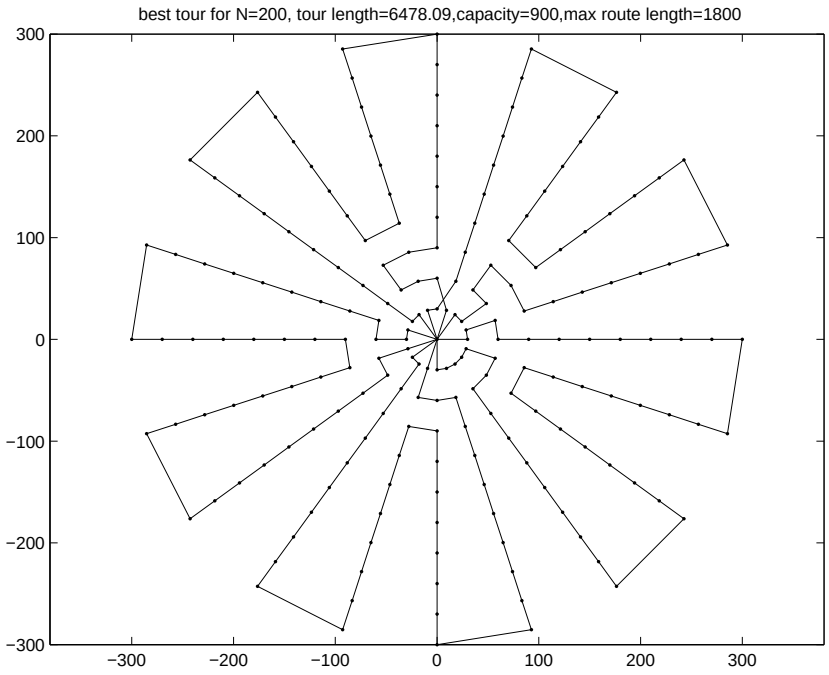


Figure 2.24: 200-node problem best VRTR tour.

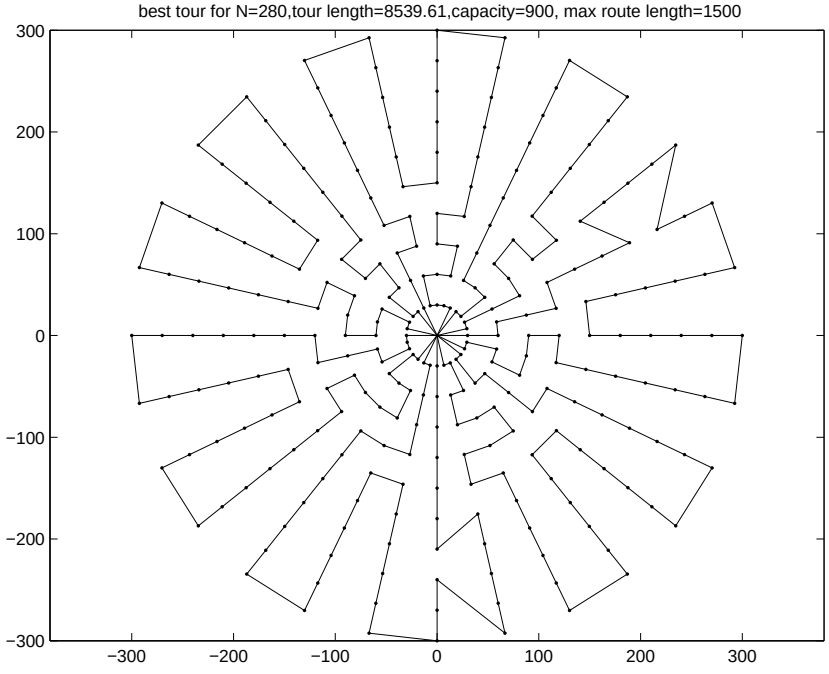


Figure 2.25: 280-node problem best VRTR tour.

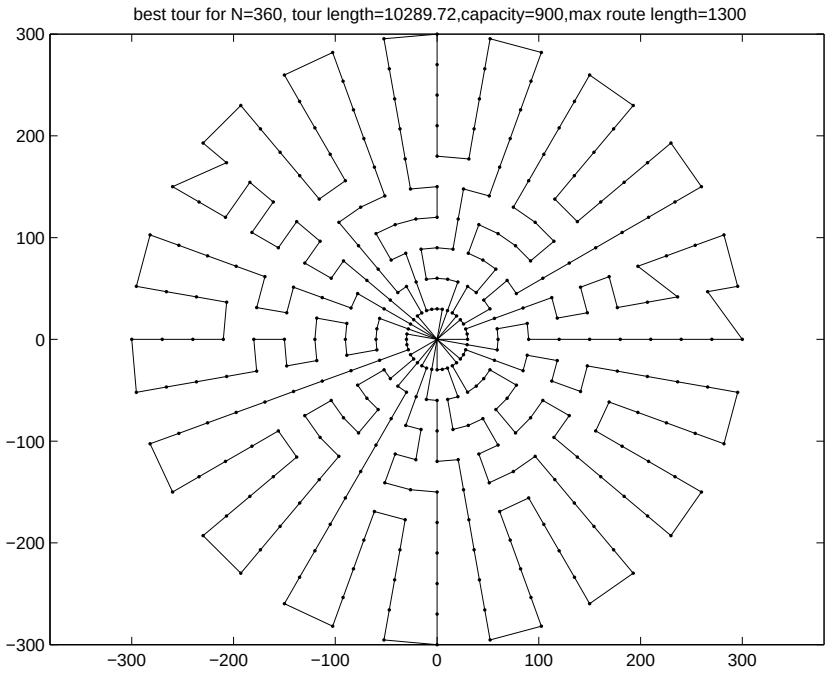


Figure 2.26: 360-node problem best VRTR tour.

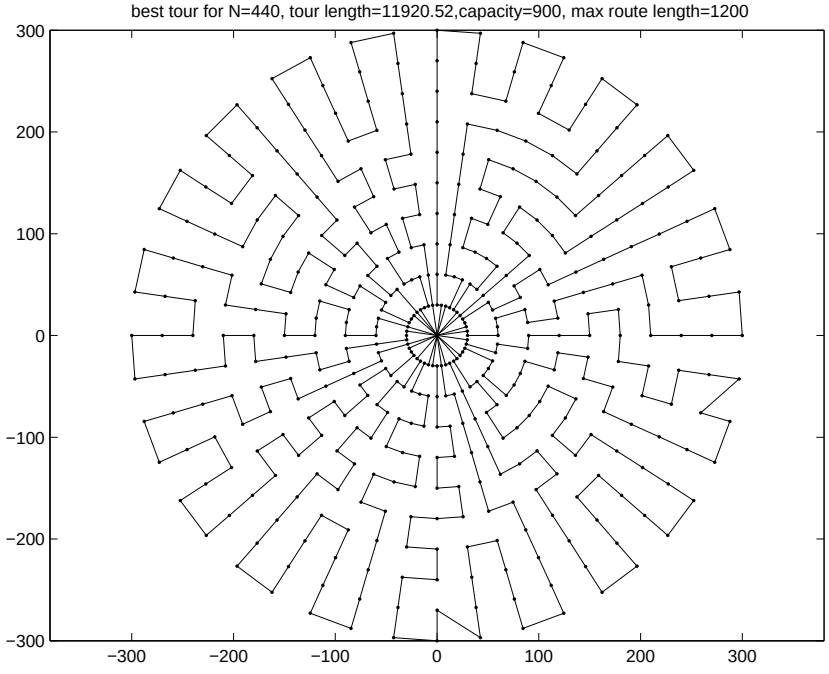


Figure 2.27: 440-node problem best VRTR tour.

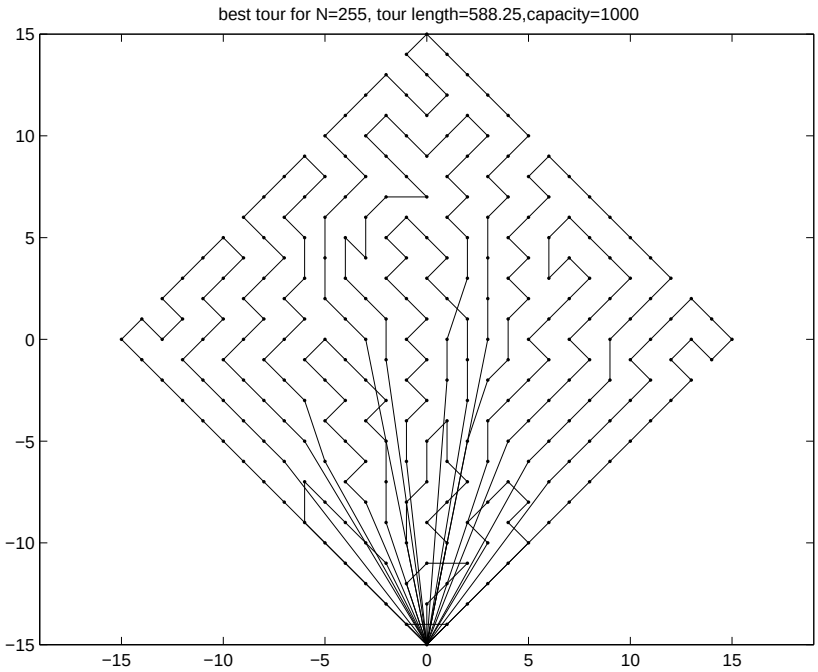


Figure 2.28: 255-node problem best VRTR tour.

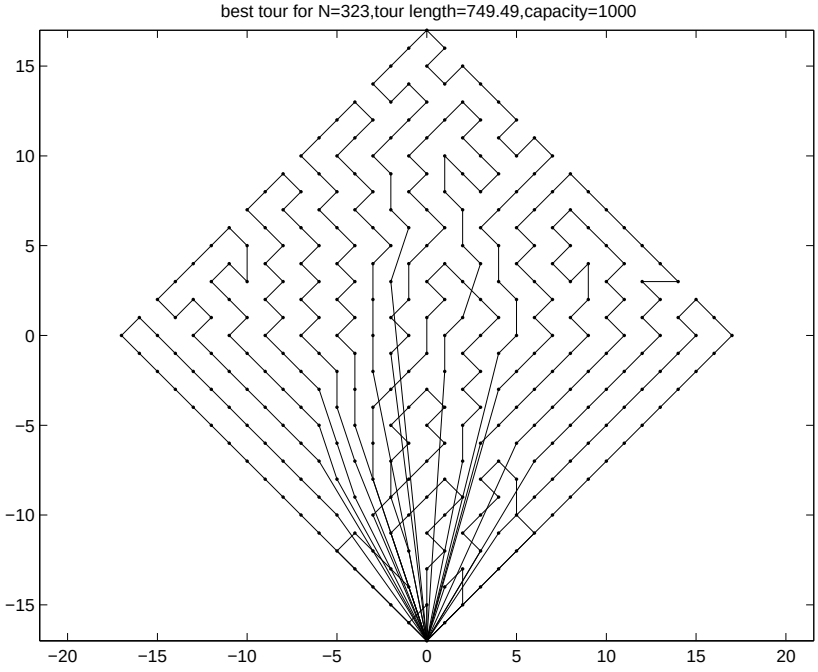


Figure 2.29: 323-node problem best VRTR tour.

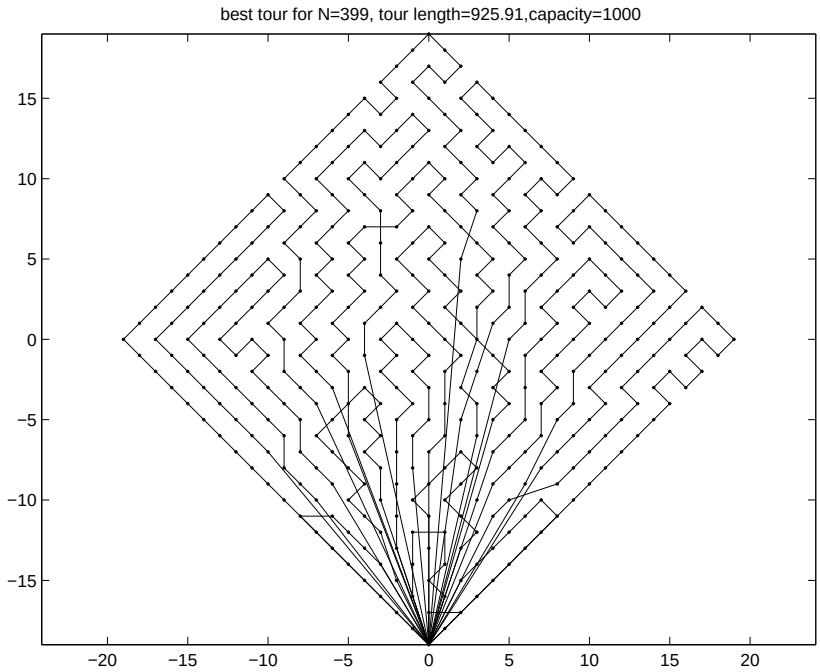


Figure 2.30: 399-node problem best VRTR tour.

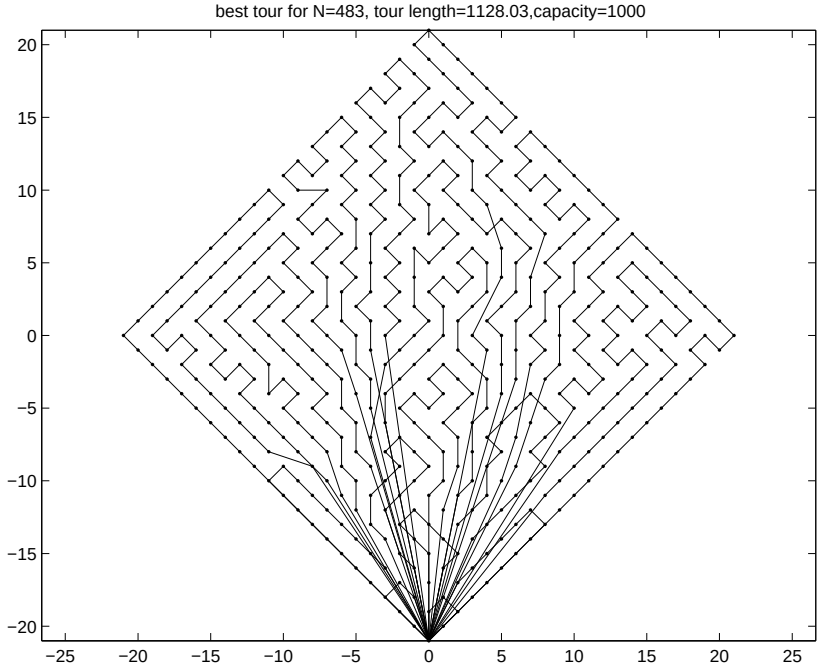


Figure 2.31: 483-node problem best VRTR tour.

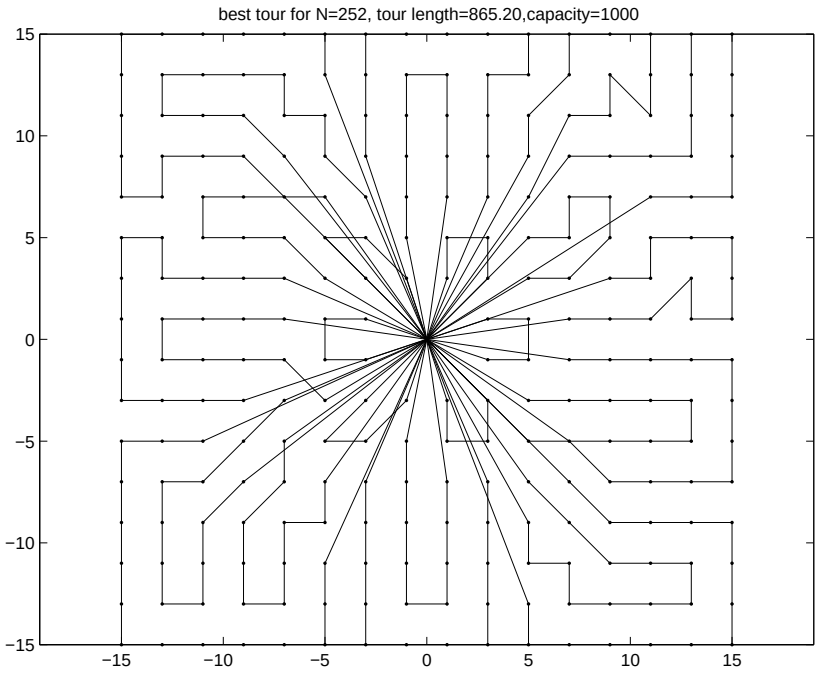


Figure 2.32: 252-node problem best VRTR tour.

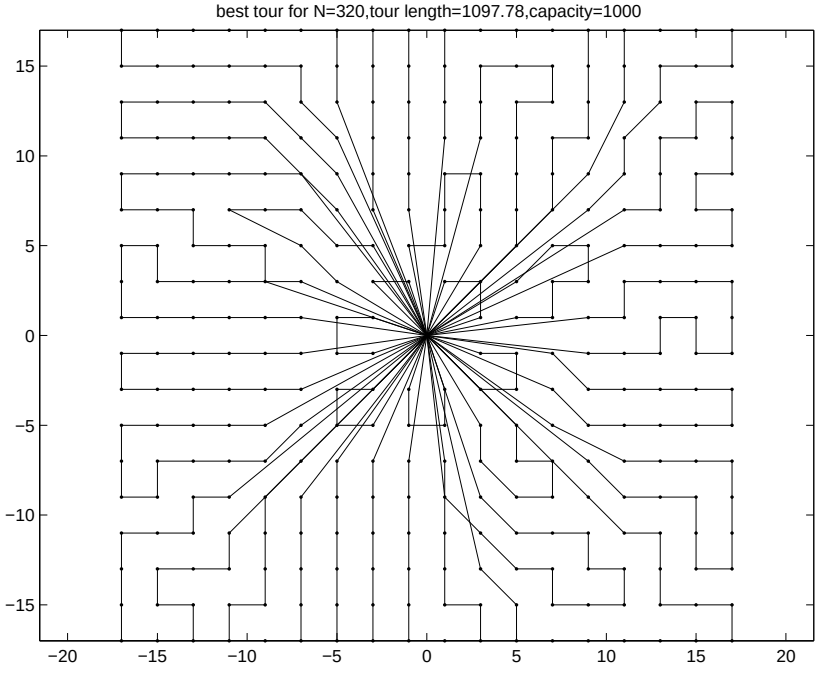


Figure 2.33: 320-node problem best VRTR tour.

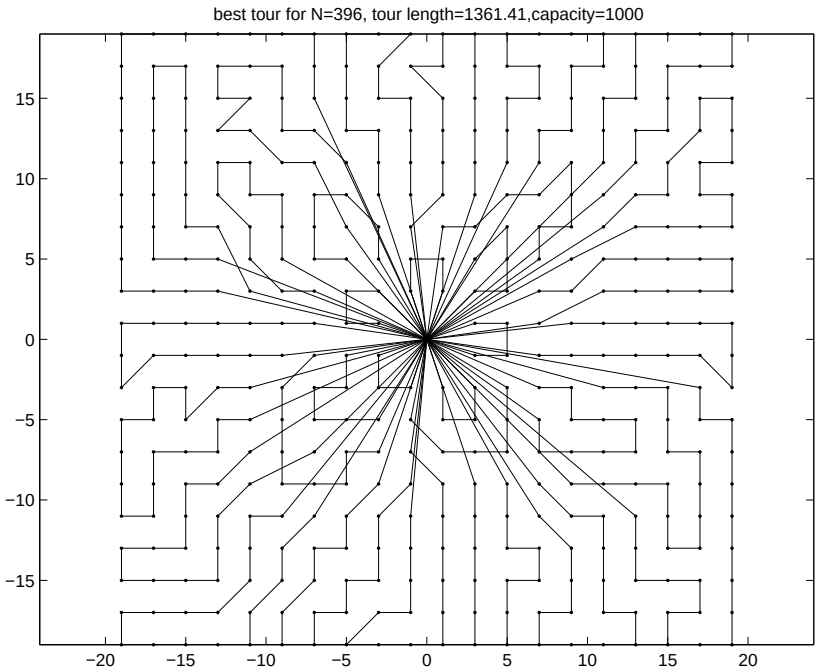


Figure 2.34: 396-node problem best VRTR tour.

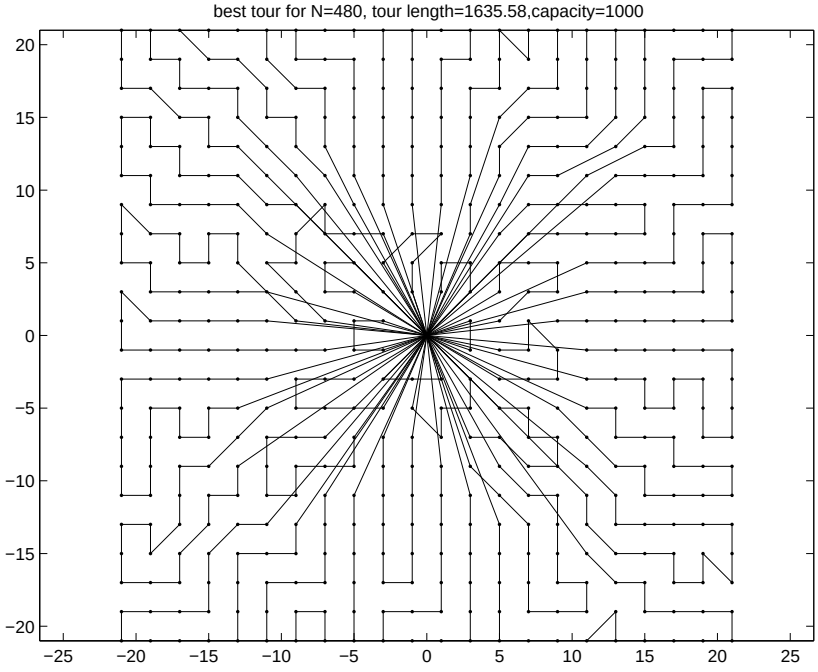


Figure 2.35: 480-node problem best VRTR tour.

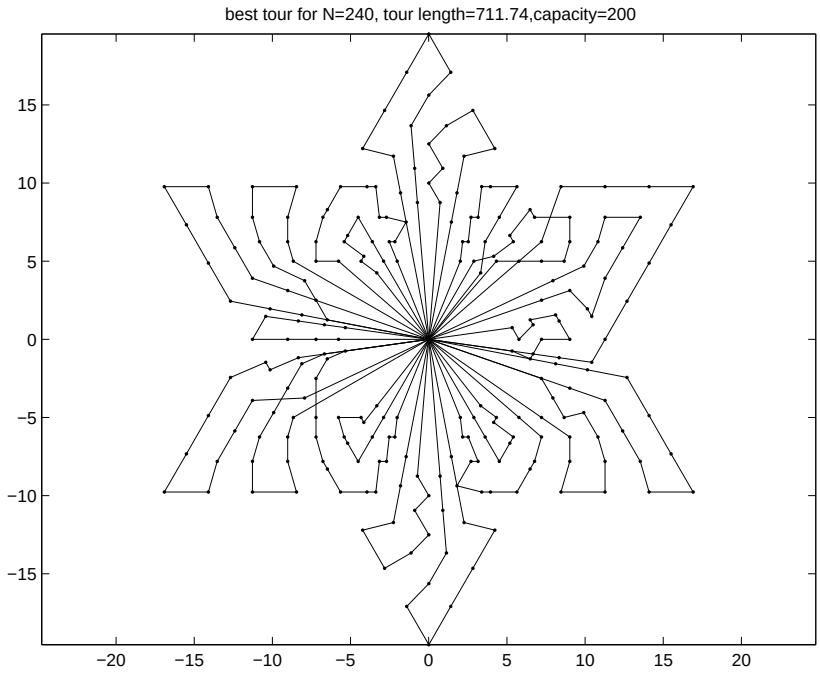


Figure 2.36: 240-node problem best VRTR tour.

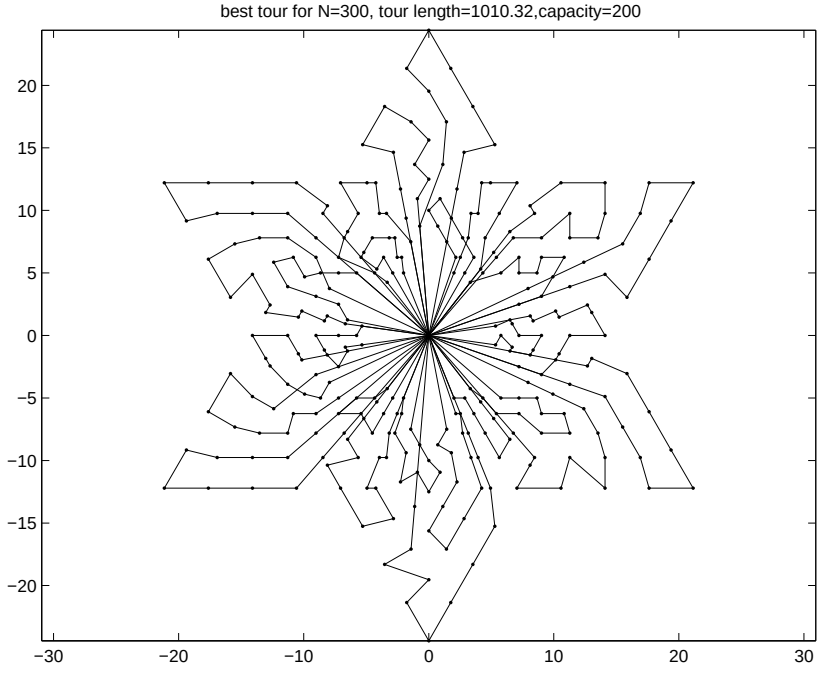


Figure 2.37: 300-node problem best VRTR tour.

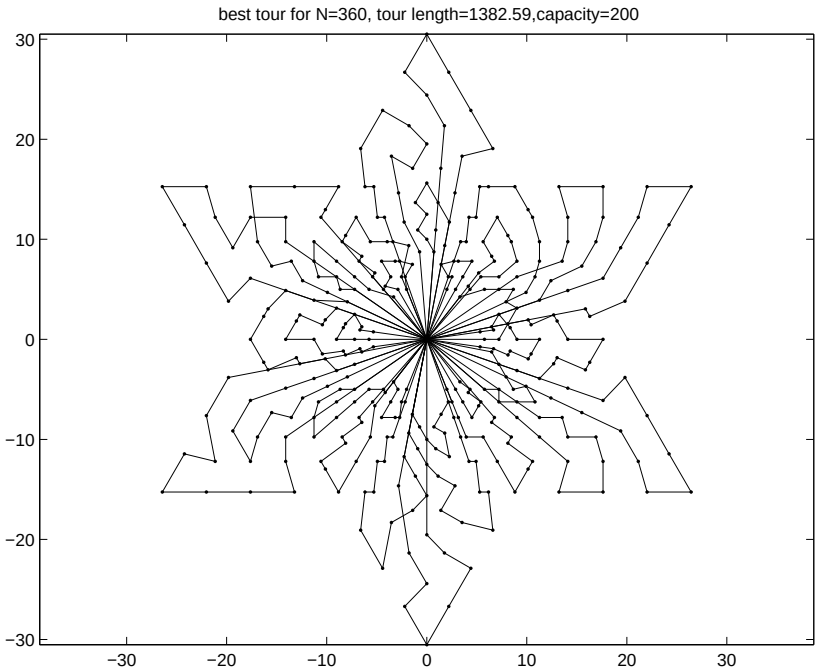


Figure 2.38: 360-node problem best VRTR tour.

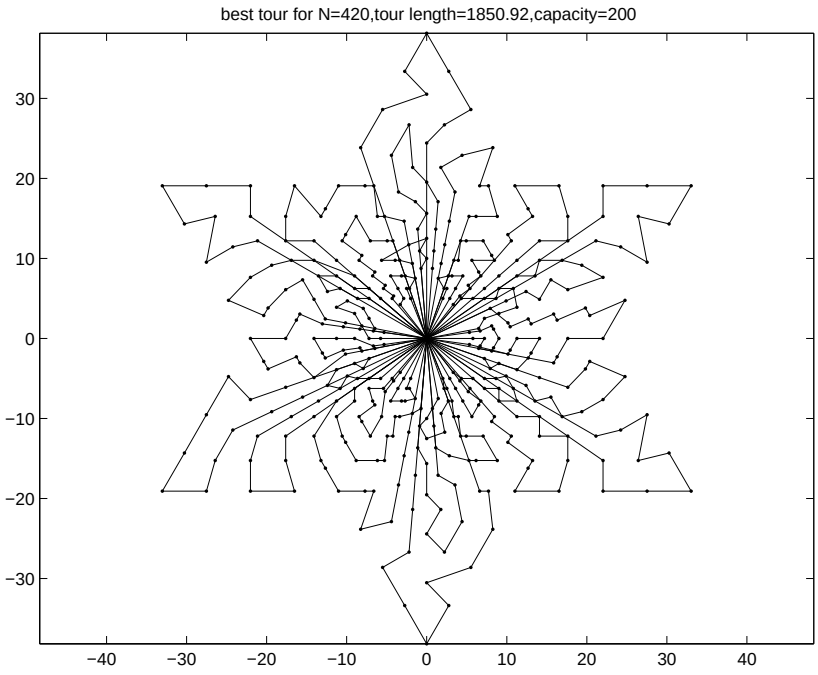


Figure 2.39: 420-node problem best VRTR tour.

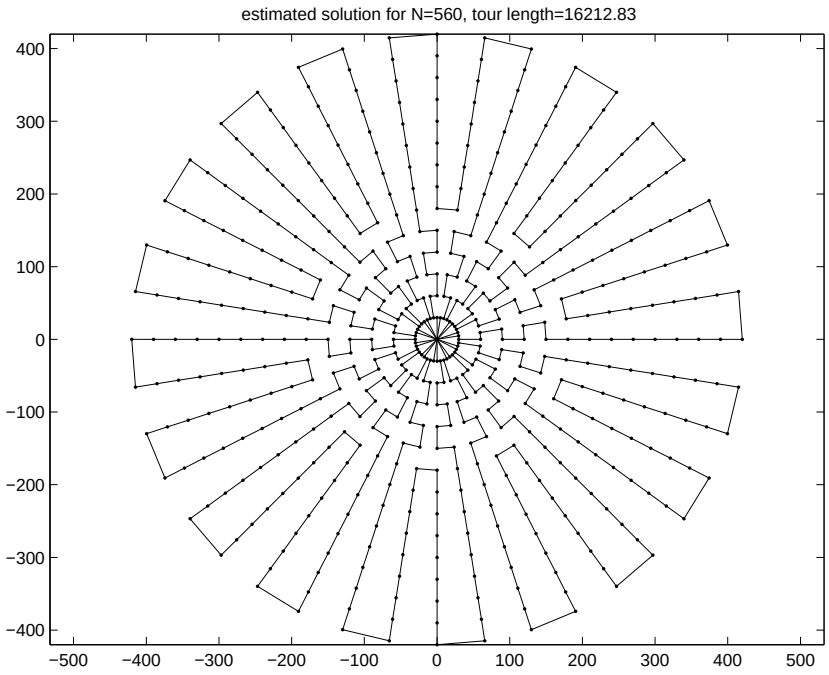


Figure 2.40: 560-node problem estimated tour.

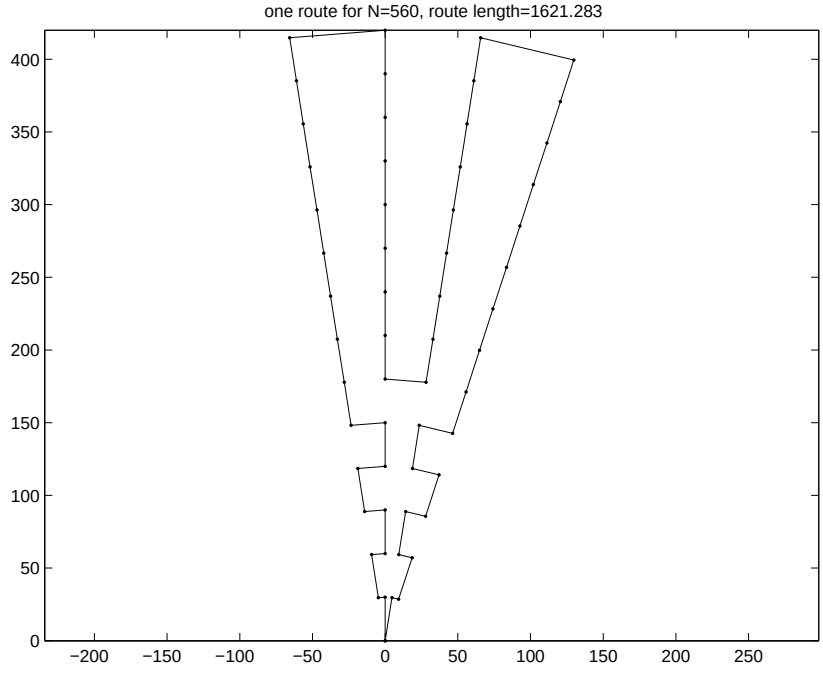


Figure 2.41: One route of 560-node problem estimated solution.

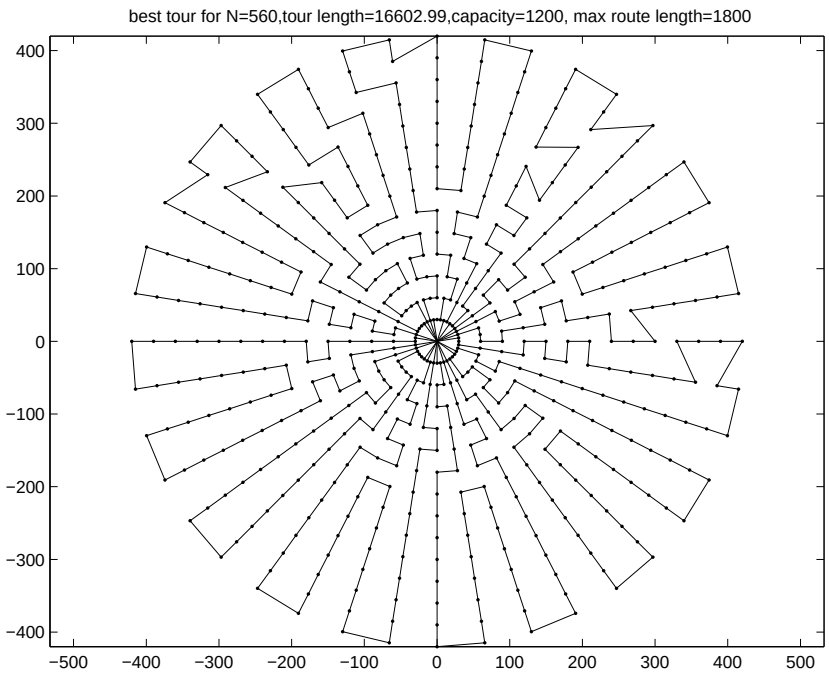


Figure 2.42: 560-node problem best VRTR tour.

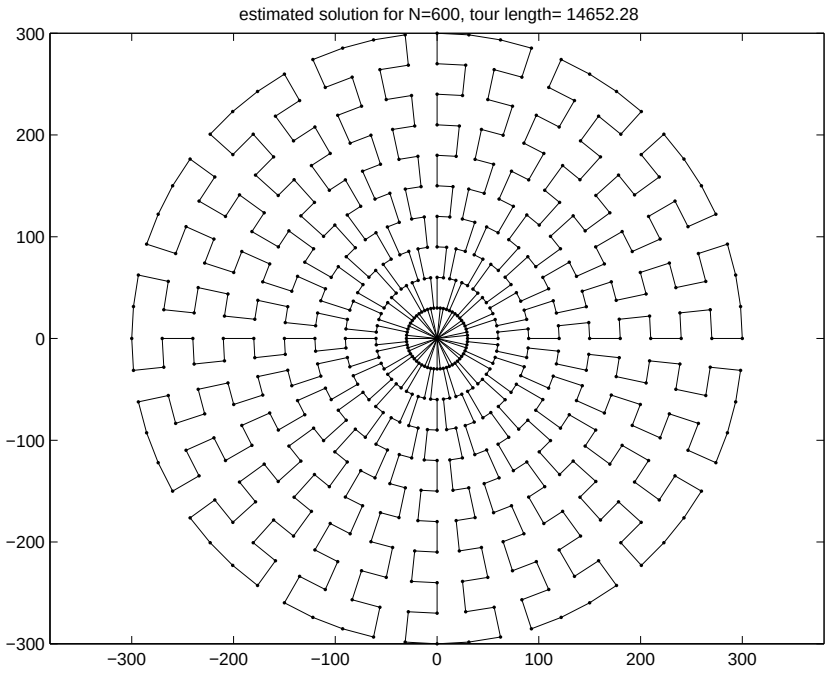


Figure 2.43: 600-node problem estimated tour.

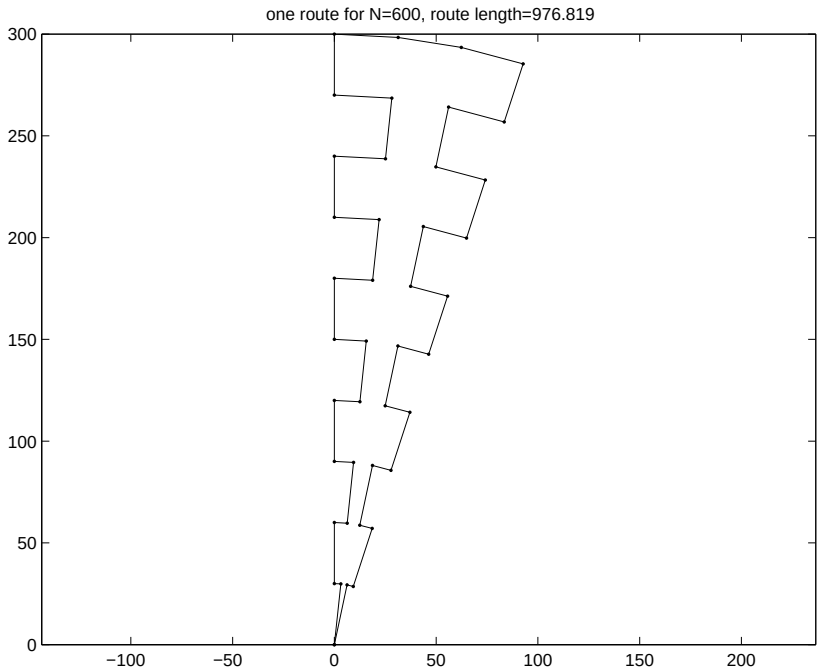


Figure 2.44: One route of 600-node problem estimated solution.

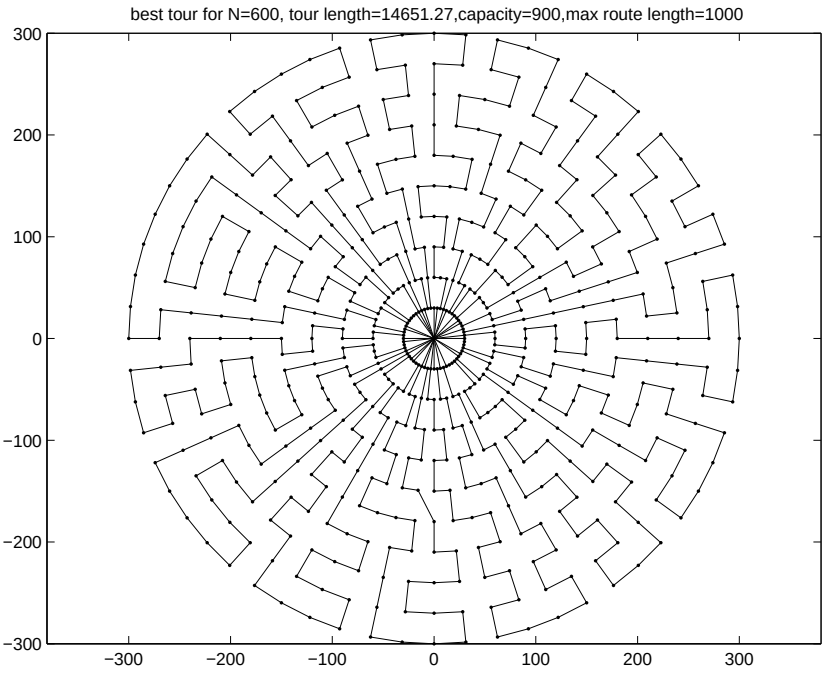


Figure 2.45: 600-node problem best VRTR tour.

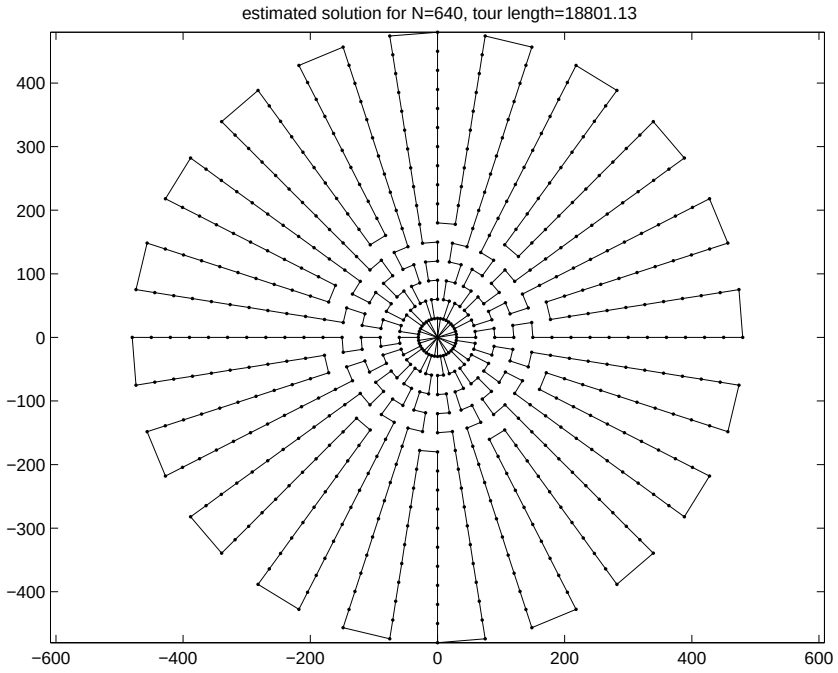


Figure 2.46: 640-node problem estimated tour.

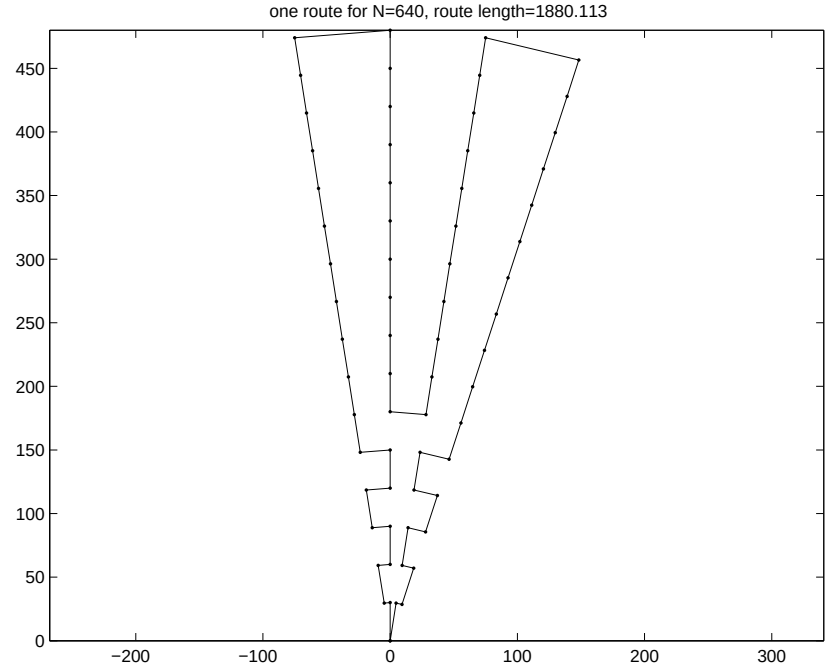


Figure 2.47: One route of 640-node problem estimated solution.

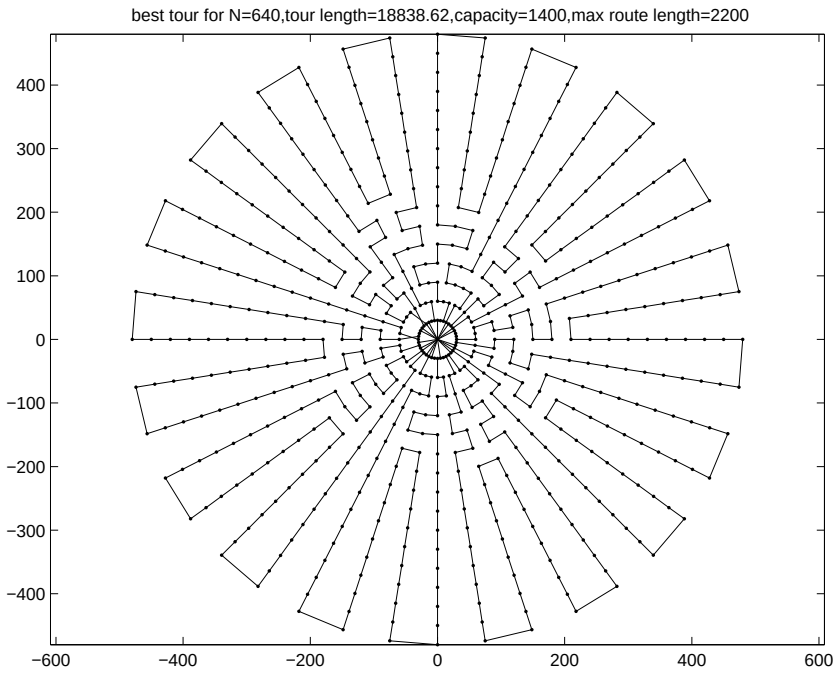


Figure 2.48: 640-node problem best VRTR tour.

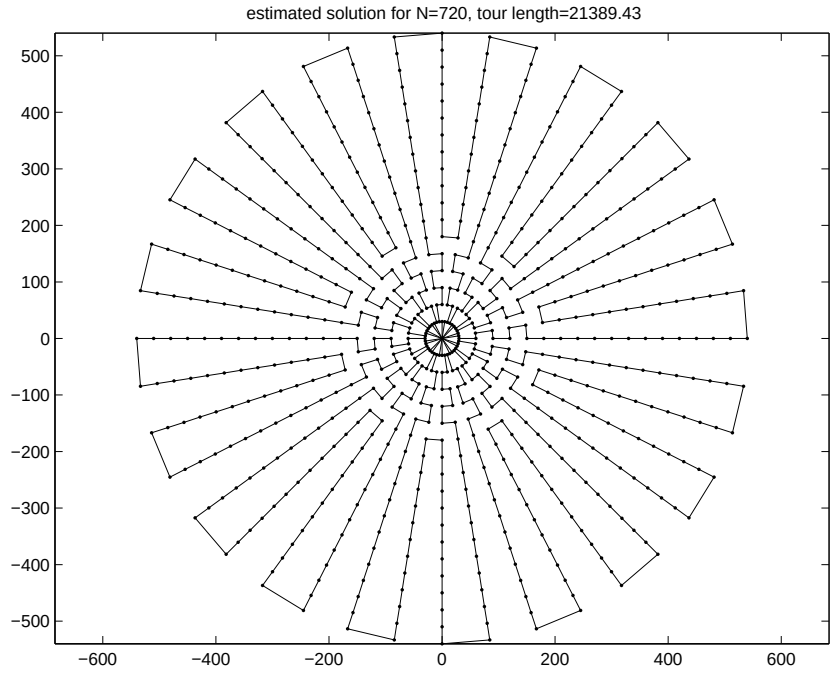


Figure 2.49: 720-node problem estimated tour.

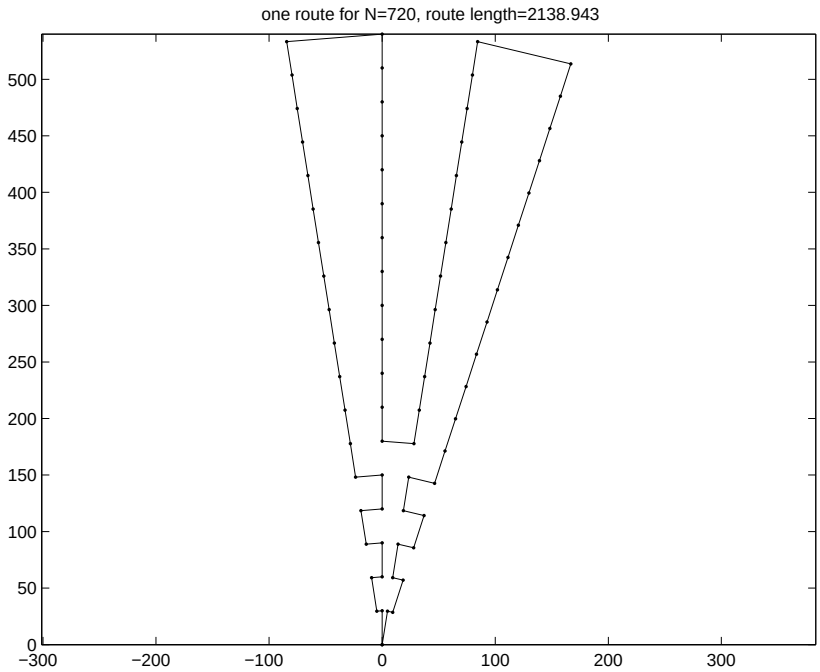


Figure 2.50: One route of 720-node problem estimated solution.

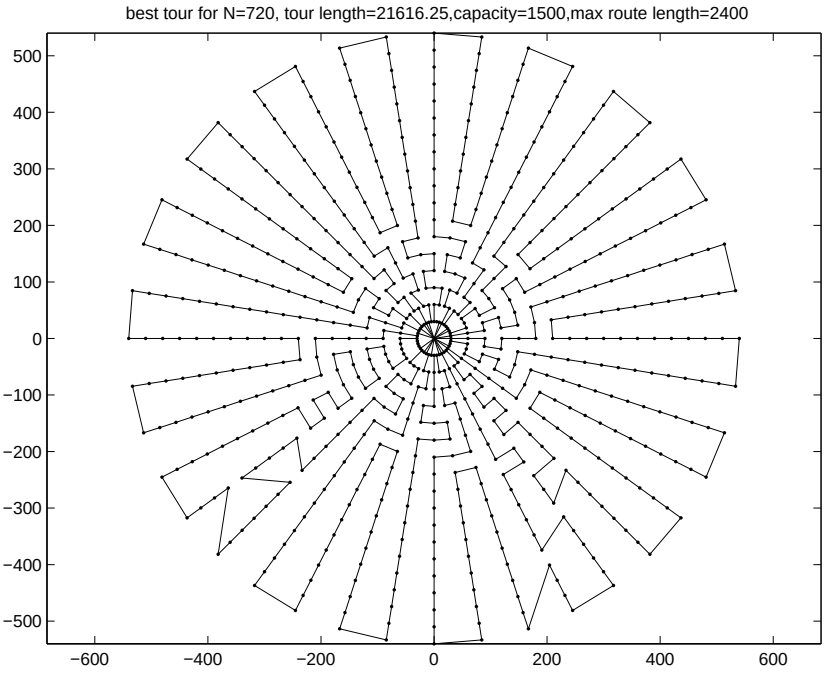


Figure 2.51: 720-node problem best VRTR tour.

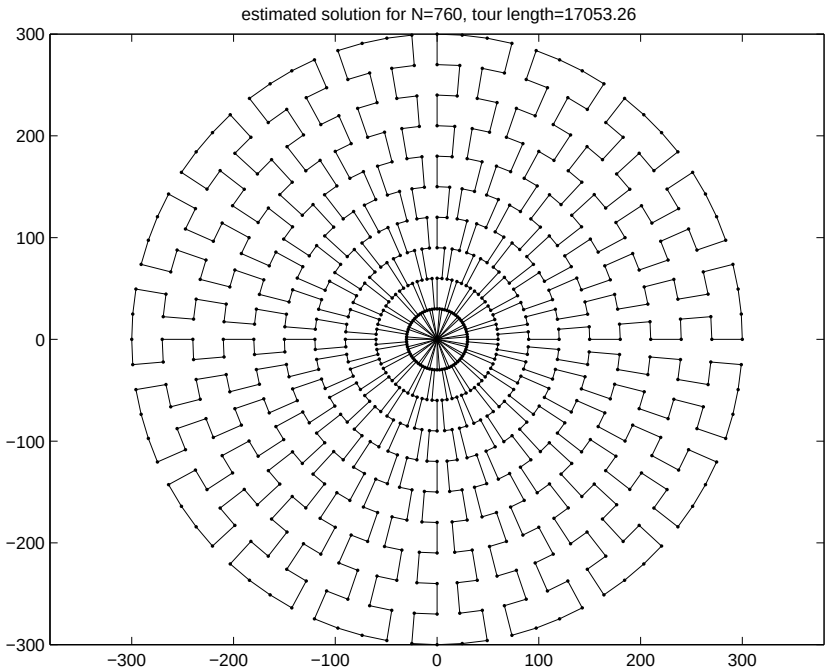


Figure 2.52: 760-node problem estimated tour.

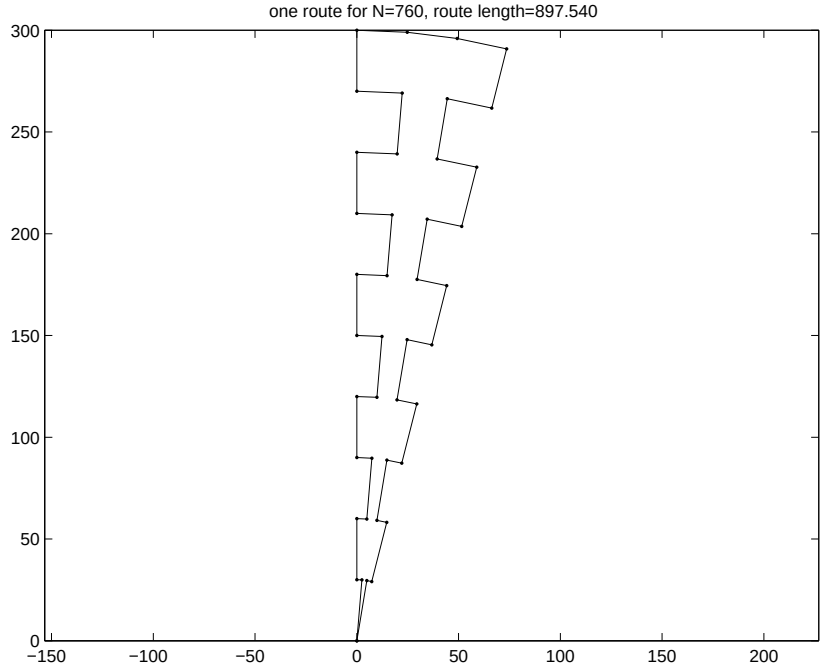


Figure 2.53: One route of 760-node problem estimated solution.

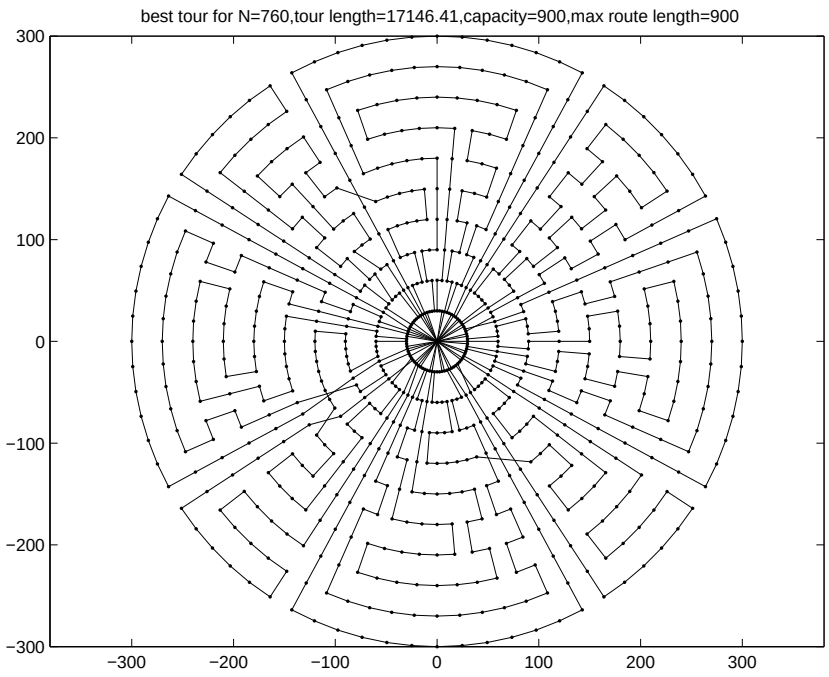


Figure 2.54: 760-node problem best VRTR tour.

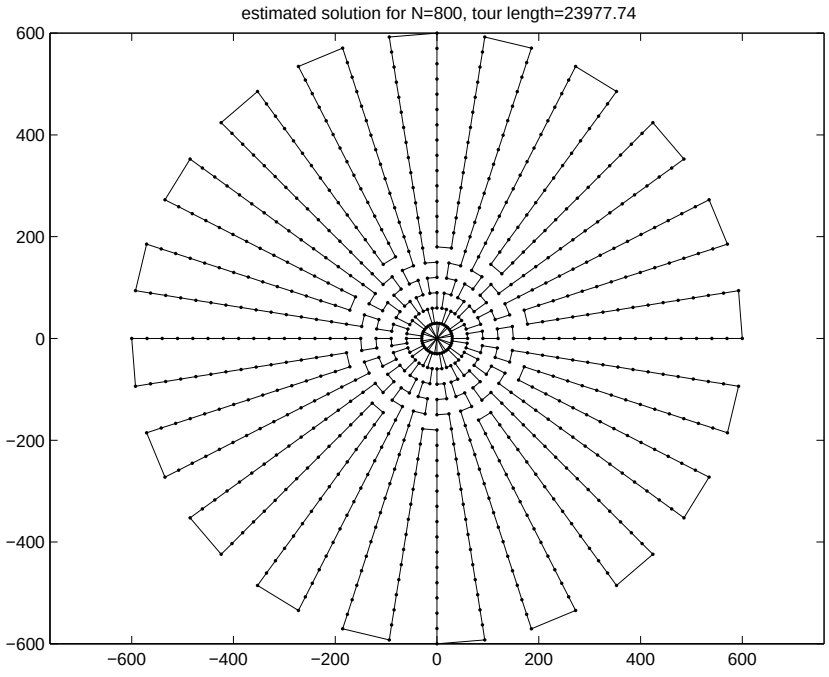


Figure 2.55: 800-node problem estimated tour.

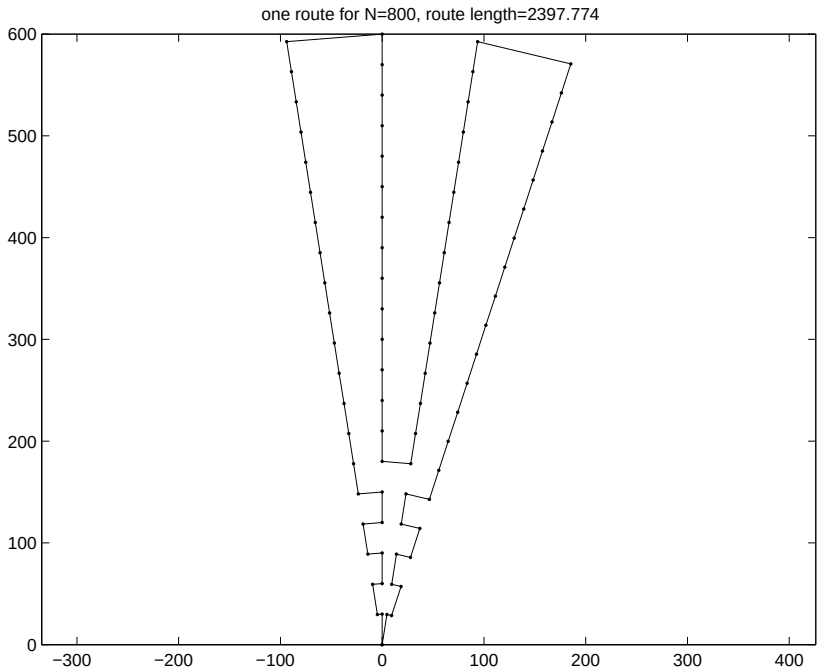


Figure 2.56: One route of 800-node problem estimated solution.

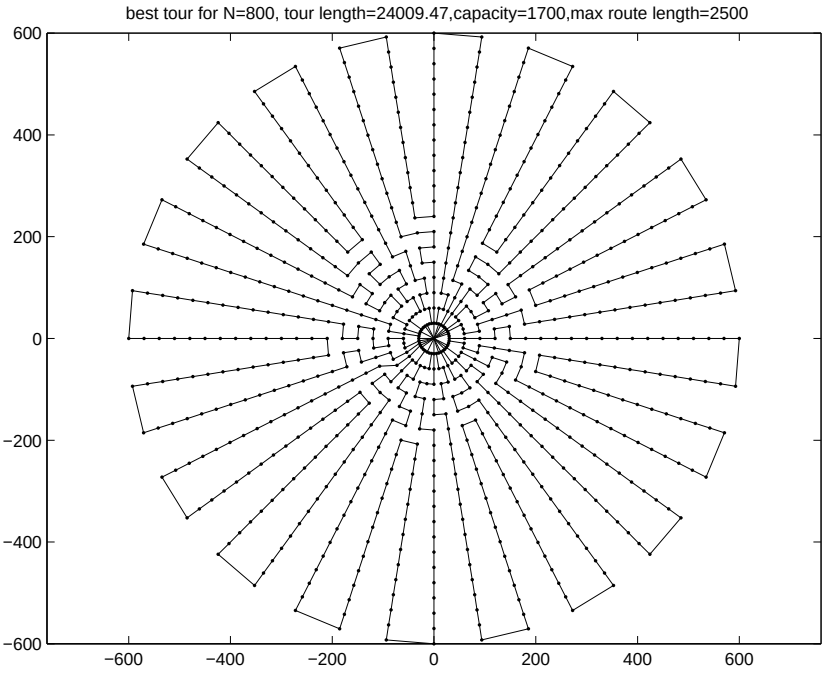


Figure 2.57: 800-node problem best VRTR tour.

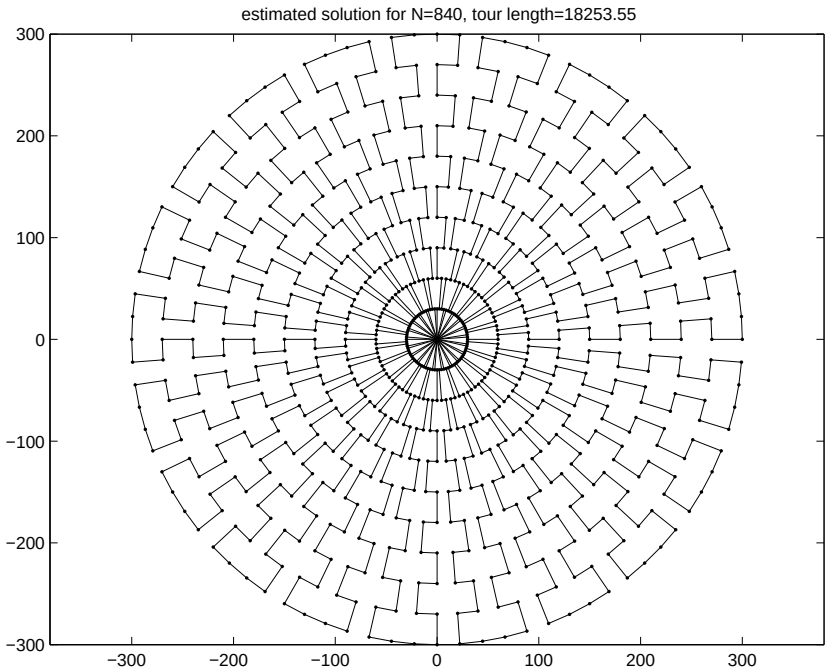


Figure 2.58: 840-node problem estimated tour.

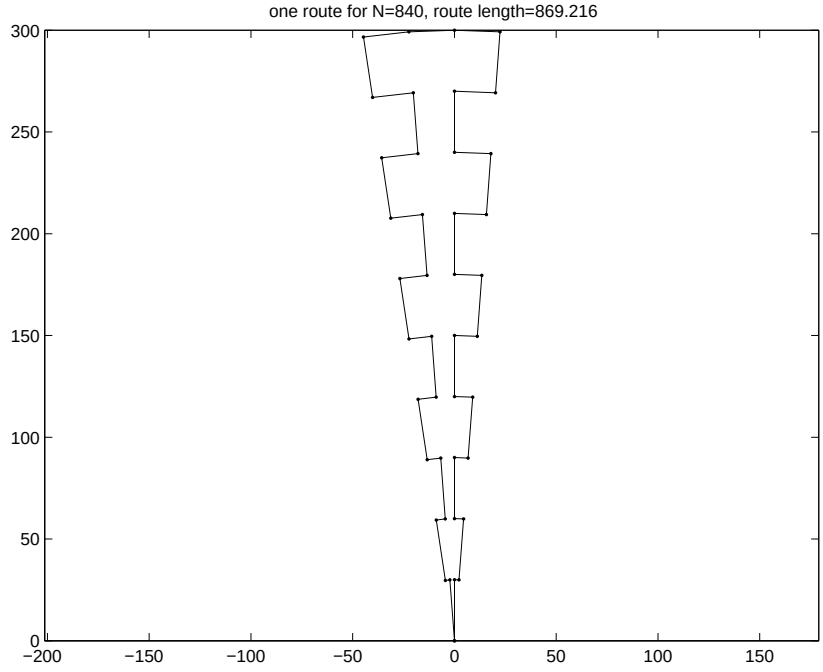


Figure 2.59: One route of 840-node problem estimated solution.

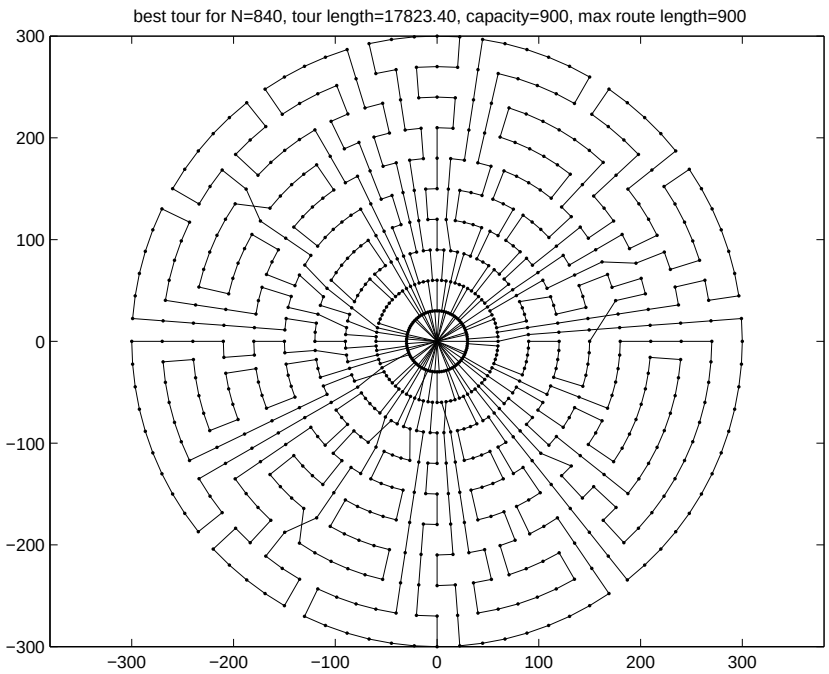


Figure 2.60: 840-node problem best VRTR tour.

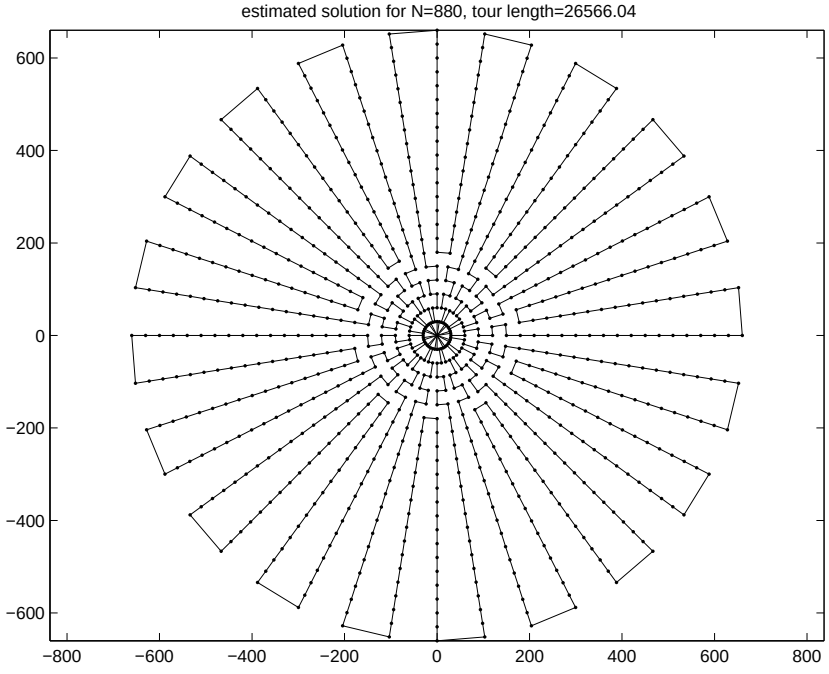


Figure 2.61: 880-node problem estimated tour.

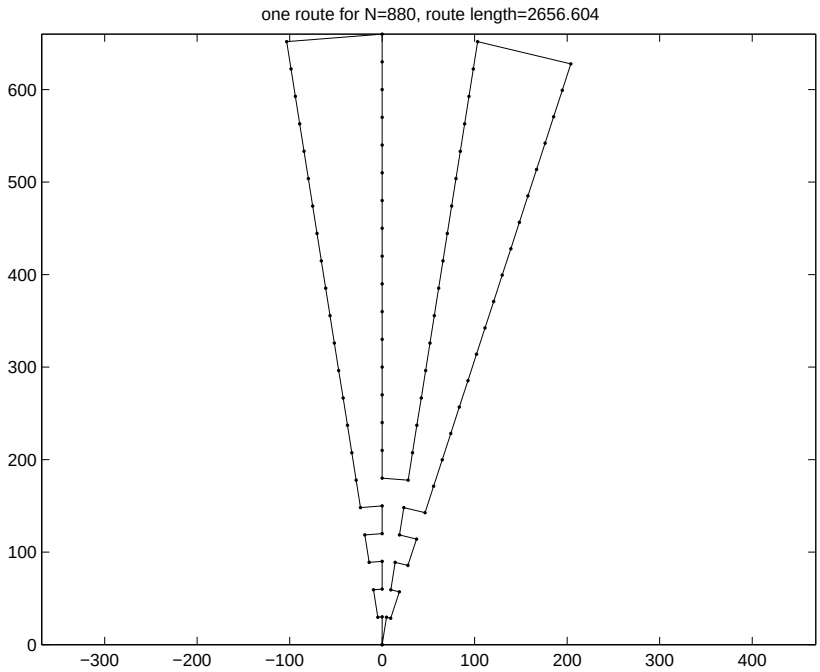


Figure 2.62: One route of 880-node problem estimated solution.

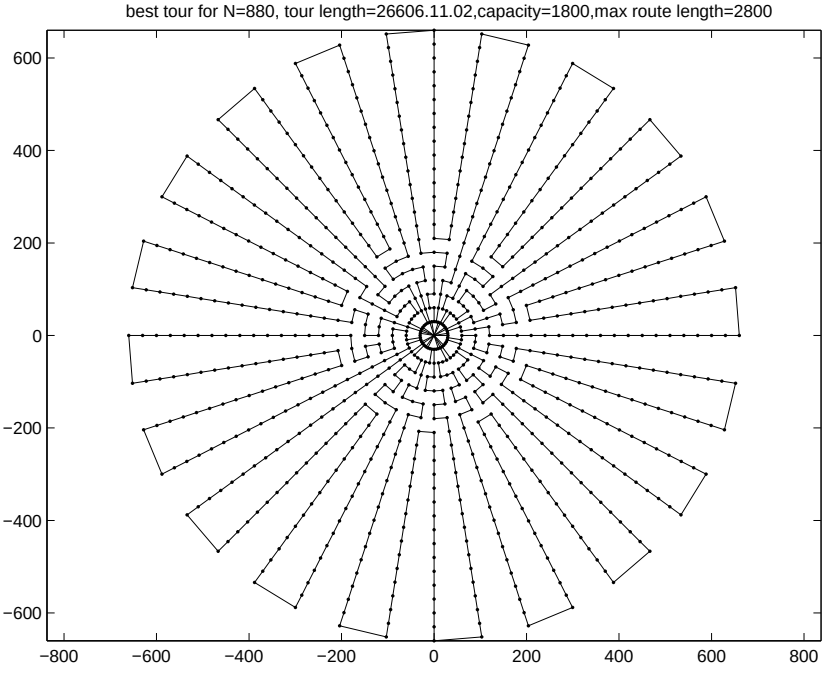


Figure 2.63: 880-node problem best VRTR tour.

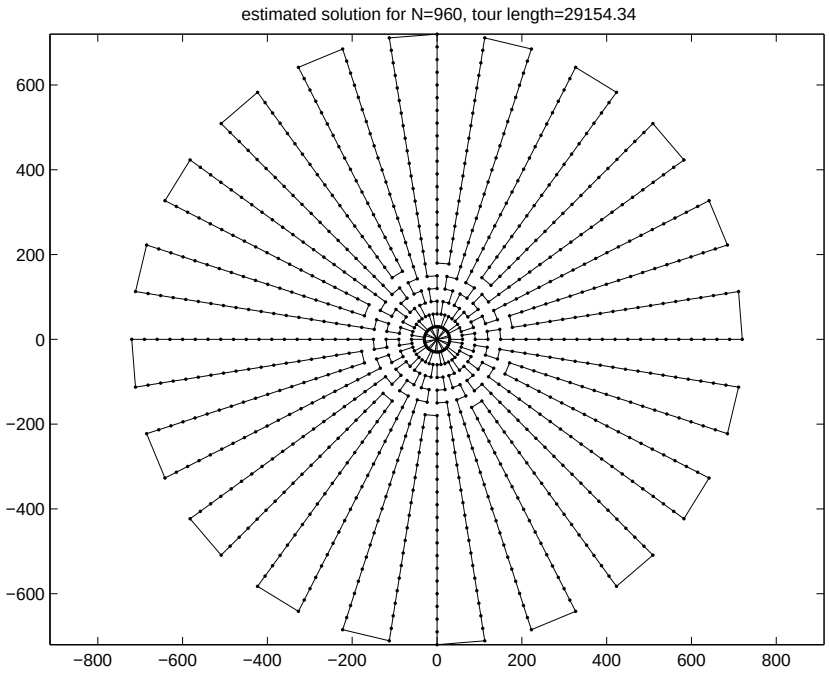


Figure 2.64: 960-node problem estimated tour.

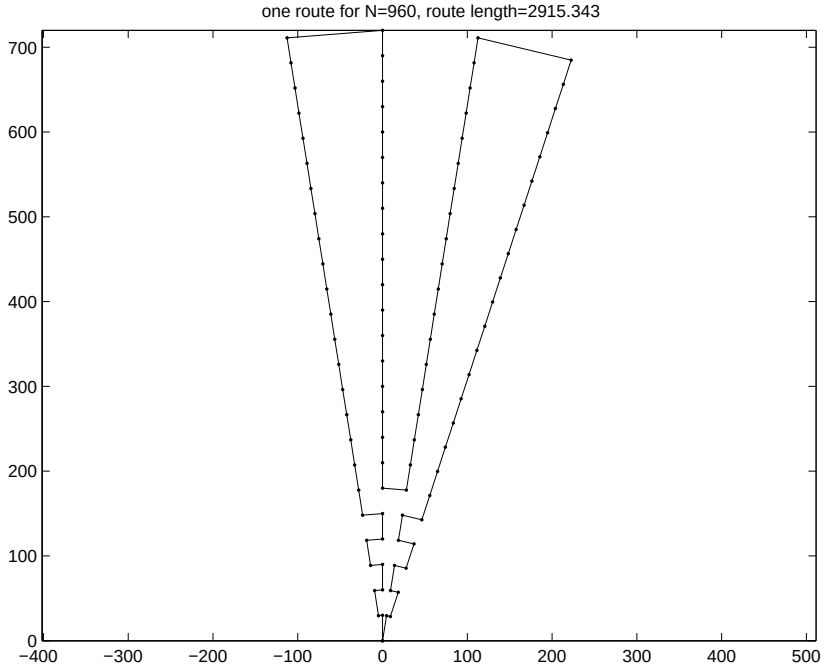


Figure 2.65: One route of 960-node problem estimated solution.

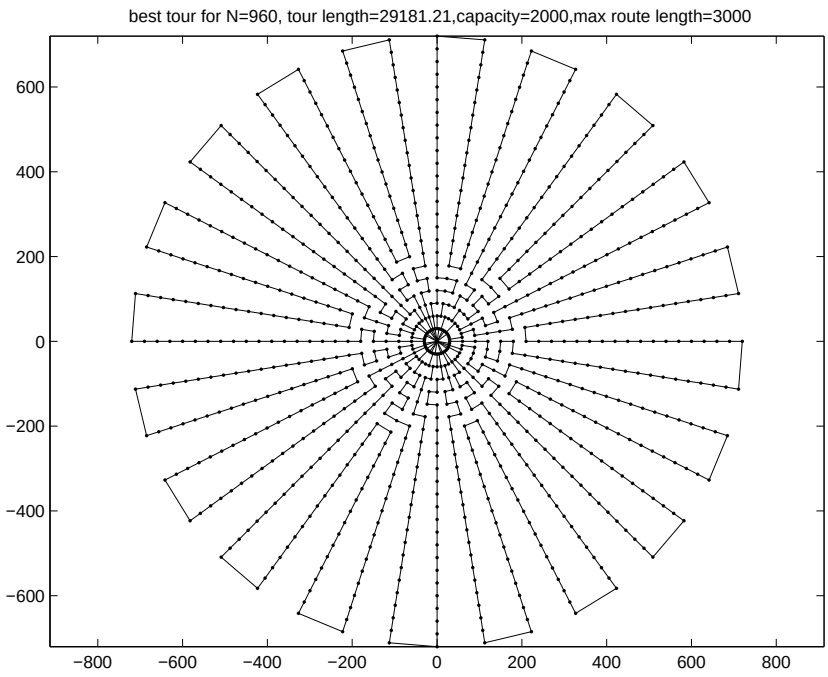


Figure 2.66: 960-node problem best VRTR tour.

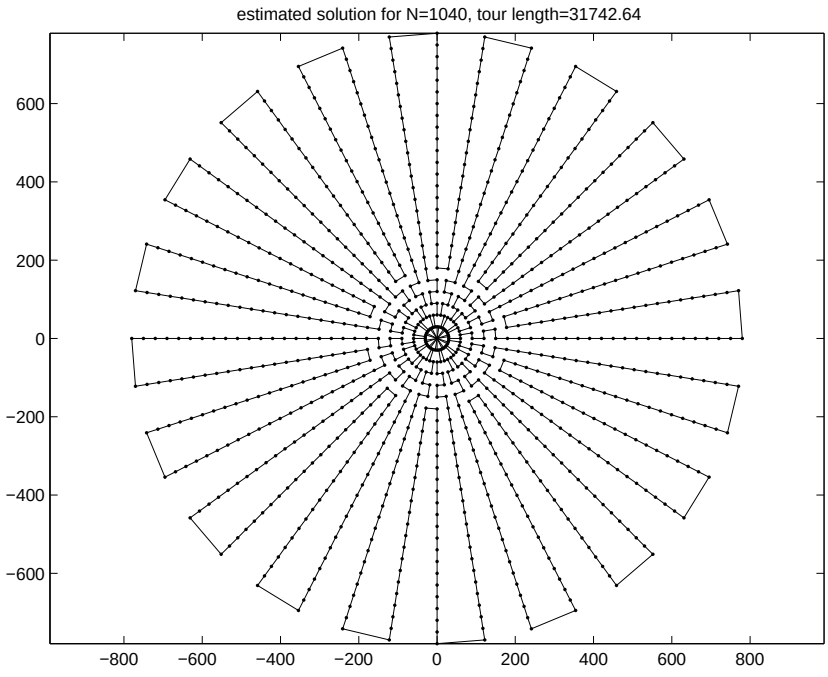


Figure 2.67: 1040-node problem estimated tour.

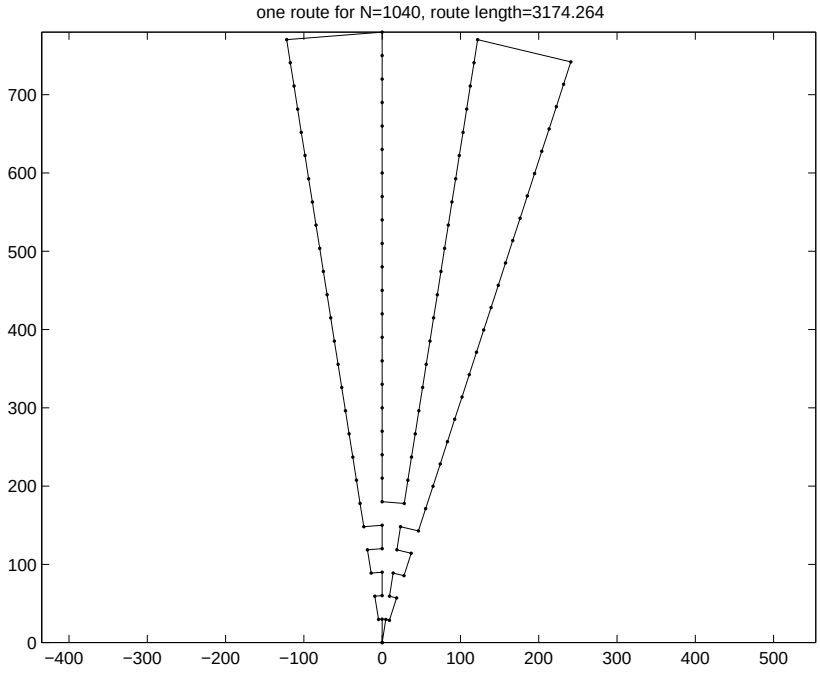


Figure 2.68: One route of 1040-node problem estimated solution.

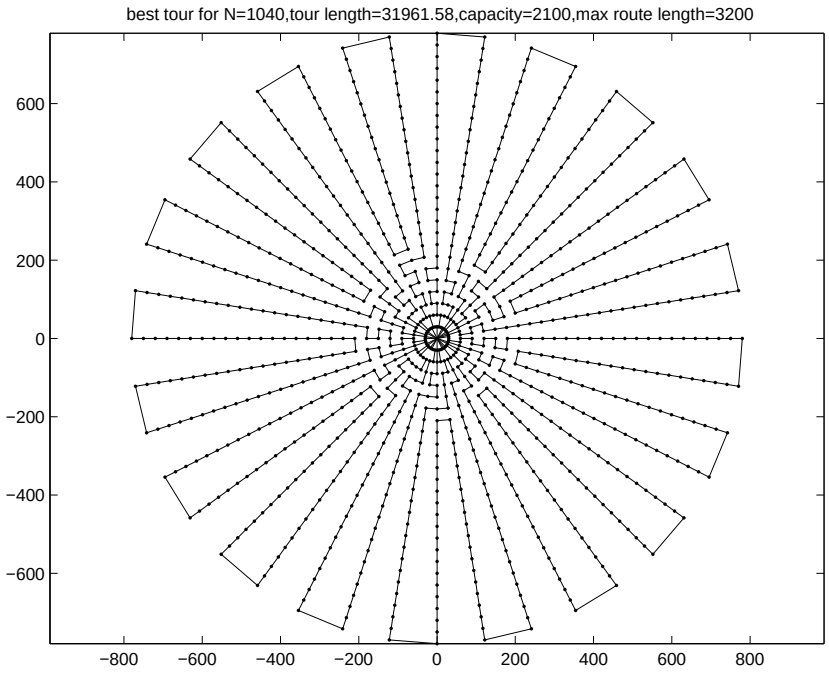


Figure 2.69: 1040-node problem best VRTR tour.

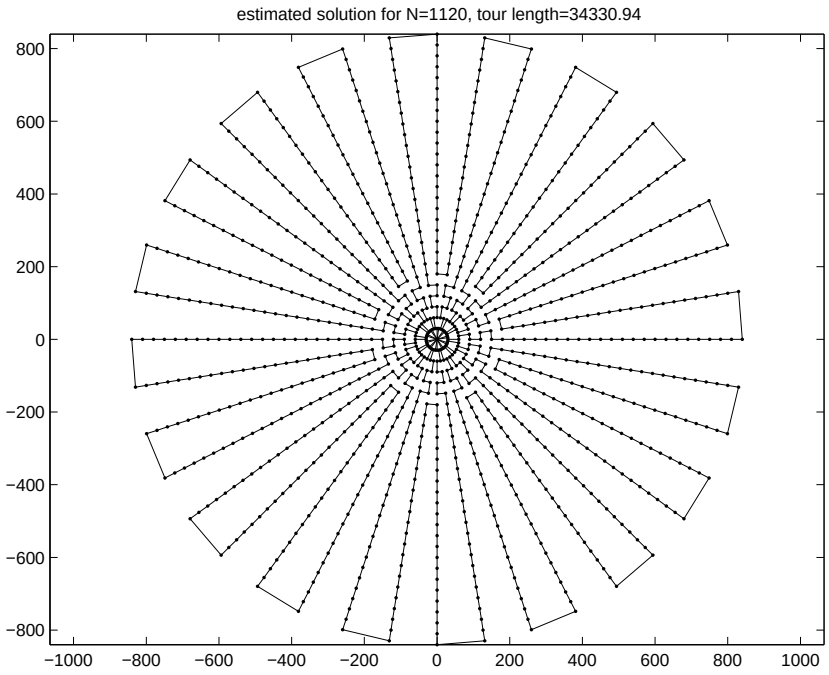


Figure 2.70: 1120-node problem estimated tour.

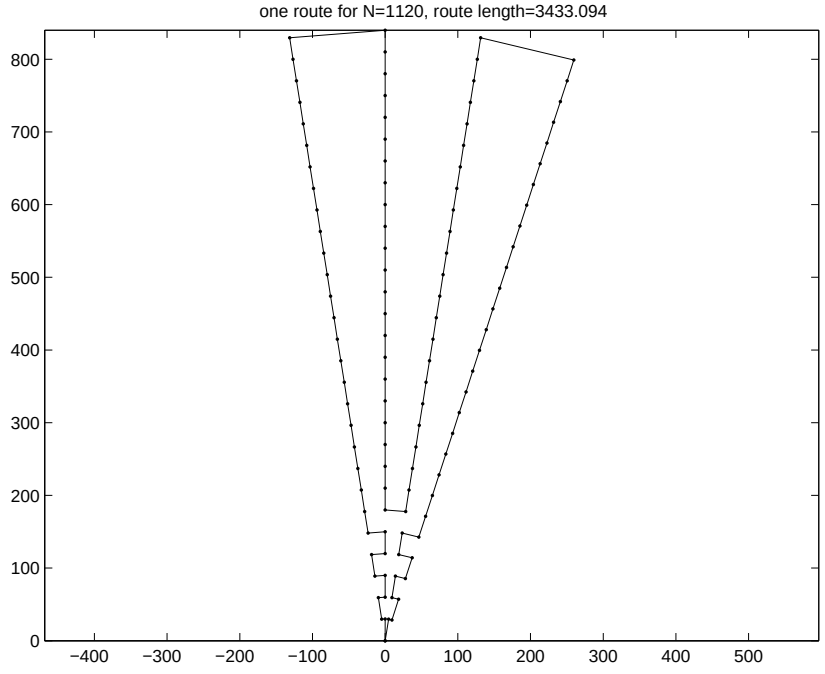


Figure 2.71: One route of 1120-node problem estimated solution.

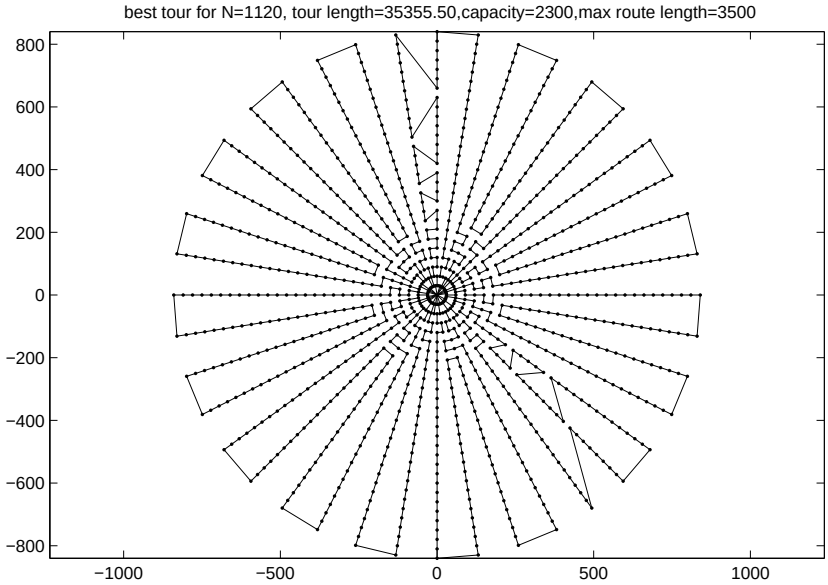


Figure 2.72: 1120-node problem best VRTR tour.

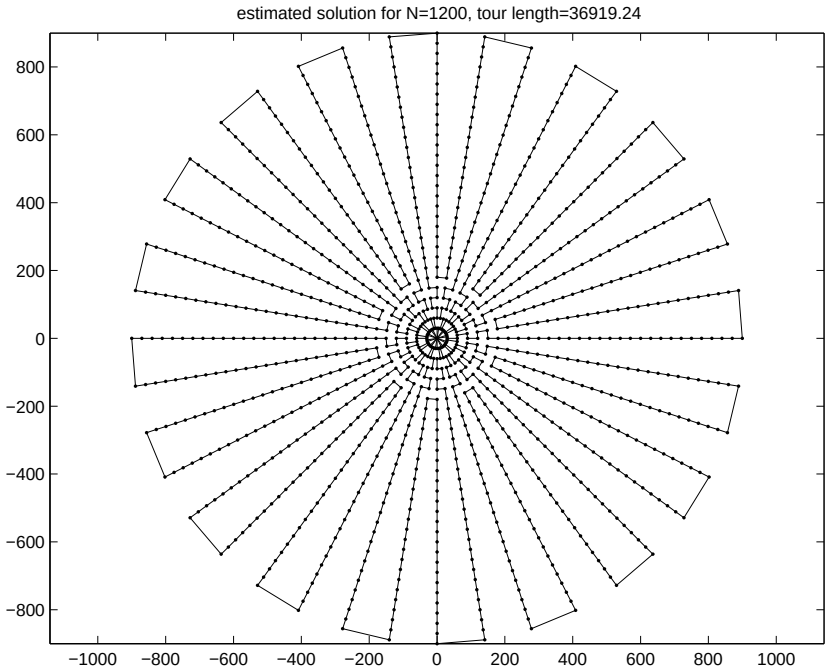


Figure 2.73: 1200-node problem estimated tour.

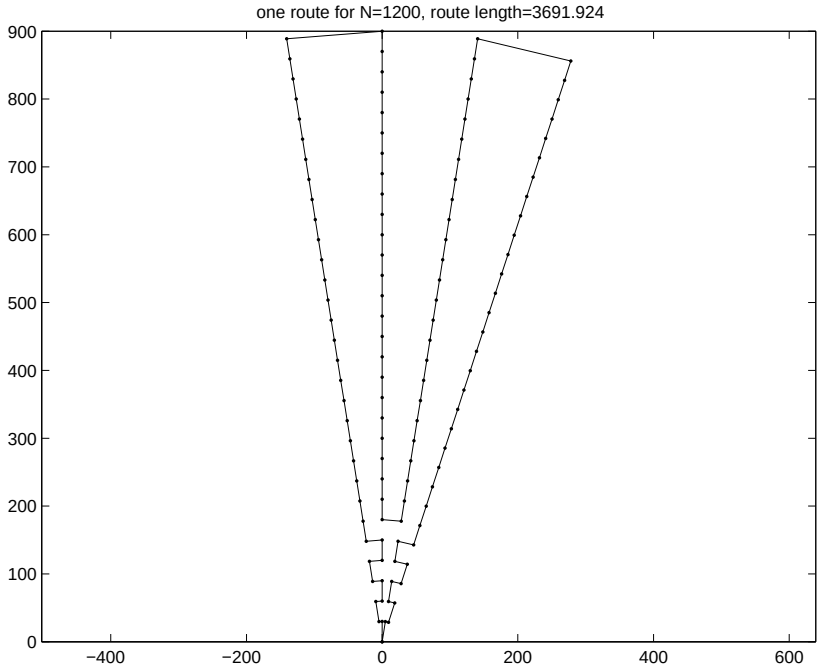


Figure 2.74: One route of 1200-node problem estimated solution.

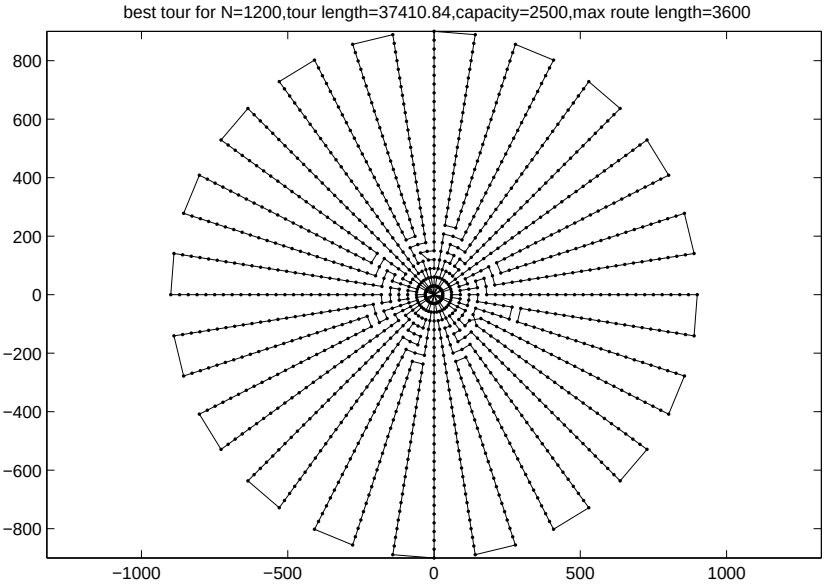


Figure 2.75: 1200-node problem best VRTR tour.

Chapter 3

The Time Dependent Traveling Salesman Problem

3.1 Introduction

In the standard version of the traveling salesman problem (TSP), we are given a set of customers located in and around a city and the distances between each pair of customers, and need to find the shortest tour that visits each customer exactly once. The TSP has been studied for more than 50 years and a wide variety of heuristics has been developed. Applegate et al. [2], Johnson and McGeoch [23], and Junger, Reinelt and Rinaldi [24] are excellent sources for algorithmic and computational aspects of the TSP.

In this chapter, we consider the following variant of the TSP. Suppose that some of the customers are located in the center of the city. Within a window of time, the center city becomes congested so that the time to travel between customers takes longer. Clearly, we would like to construct a tour that avoids visiting customers when the center of the city is congested. This variant of the TSP is known as the time dependent traveling salesman problem (TDTSP).

Recently, Bentner et al. [5] and Schneider [36] studied the TDTSP. They considered the Bier127 problem from TSPLIB [33] and defined a region in the city center in which traffic jams occurs in the afternoon. In Figure 3.1, we show

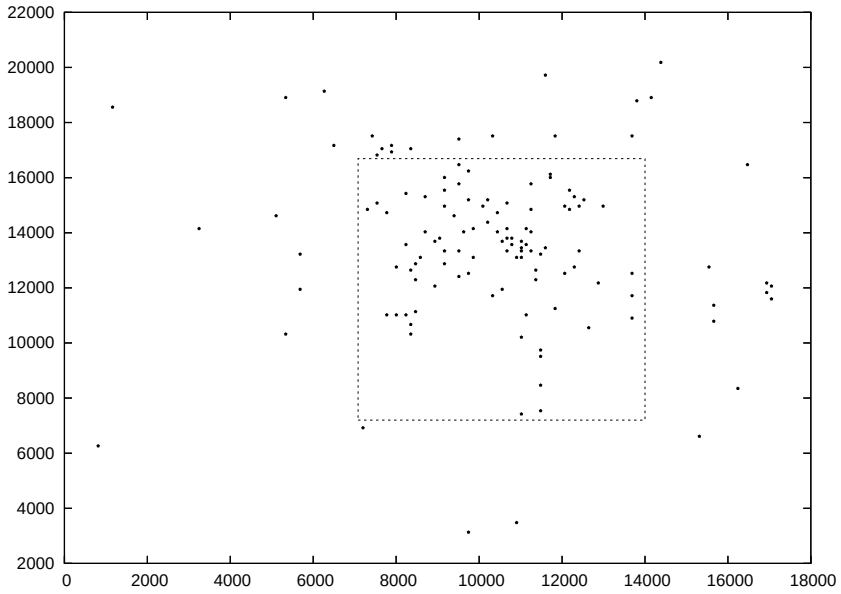


Figure 3.1: Bier127 with the location of 127 beer gardens in and around Augsburg. Afternoon traffic jams occur in the dashed rectangular region.

Bier127 with the locations of 127 beer gardens in and around Augsburg, Germany. Congestion occurs in the afternoon for beer gardens in the dashed rectangle (the traffic jam region), so that the time to drive between two locations in the rectangle is multiplied by a jam factor $f > 1$.

Bentner et al. and Schneider varied the value of the jam factor and generated tours for Bier127 using simulated annealing. As the value of the jam factor increased, they found that locations in the jam factor region were typically avoided in the afternoon. In addition, Bentner et al. compared different traffic jam regions and found that, when the region was small, the salesman could detour and avoid the traffic jam without greatly increasing the tour length. However, when the traffic jam region was large, short detours were not always possible.

We point out that different variants of the TDTSP have been studied in the literature. Malandraki and Dial [30] used step functions to model the time

dependency. The time to traverse edge ij depended on the departure time t_i from the origin node i. On average, there were two or three time periods per edge. The travel times were constant for each time period. Malandraki and Dial used a restricted dynamic programming heuristic to solve the TDTSP with 10 to 55 nodes. At each stage of the dynamic program, only H best subtours were considered. They used $H = 1, 100, 1000, 5000, 15000$ in their computational experiments.

Malandraki and Daskin [29] studied a variant of the time dependent vehicle routing problem (TDVRP). Mixed integer linear programming formulations were presented that treated the travel time functions as step functions. A simple heuristic based on nearest-neighbor was developed for both the TDTSP and the TDVRP. A mathematical-programming-based heuristic using cutting planes for the TDTSP was discussed. Computational results for randomly generated problems with 10 to 25 nodes were reported.

In Section 3.2, we develop two algorithms for solving the TDTSP. In Section 3.3, we conduct computational experiments with both algorithms on Bier127. In Section 3.4, we consider the time dependent vehicle routing problem (TDVRP) and present limited computational results. In Section 3.5, we give our conclusions.

3.2 Algorithms for the TDTSP

In this section, we present two algorithms for solving the TDTSP: one based on record-to-record travel and one based on the chained Lin-Kernighan procedure.

3.2.1 Record-to-record travel algorithm

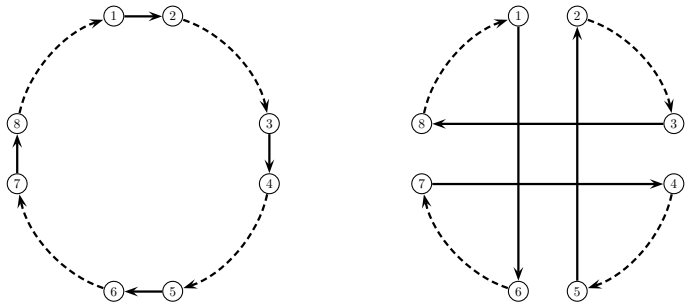
Our record-to-record travel algorithm (RTR) is based on the procedure that we developed to solve the vehicle routing problem (Li, Golden, and Wasil [28]). We describe RTR in Table 3.1. The initial solution is generated by the modified Clarke and Wright algorithm (Golden et al. [20]). We then use two-opt moves and one-point moves, and allow uphill moves. Finally, we try to improve the current solution by allowing only downhill moves.

3.2.2 Chained Lin-Kernighan algorithm

We also developed a variant of the chained Lin-Kernighan algorithm (CLK) to solve the TDTSP (a detailed description of the Lin-Kernighan algorithm is given by Junger et al. [24]; Applegate et al. [3] give a detailed description of CLK). Our variant is described in Table 3.2. In Algorithm 7, the outer loop runs for min{number of nodes/2, 100} iterations. We use a neighbor list with 25 nearest neighbors.

At the end of Algorithm 7, we perturb the current solution. We use the double-bridge kick shown in Figure 3.2 (see Applegate et al. [3] for more details). In Figure 3.2 (a), we randomly select four pairs of nodes from the current solution. We re-link them as shown in Figure 3.2 (b). This changes the structure of the current solution and will hopefully lead to a better local optimum.

In Algorithm 8, we apply an iterative operation on each node that exploits possible two-opt moves. In the traditional two-opt move, each node is examined and only downhill moves are made (see [24] for details). In Algorithm 9, we do not finish processing a node immediately if there is no downhill move. Instead, we do the two-opt move and apply Algorithm 9 (recursively) to the new solution.



(a) Four pairs of adjacent nodes (b) Re-linked nodes

Figure 3.2: Double-bridge kick.

We do this four times. If a downhill move is found, we accept it. Otherwise, we restore the solution that was generated before the recursive call (see [3] for details).

3.3 Computational experiments

In this section, we report the results of two computational experiments. We use the Bier127 problem. The traffic jam region (rectangle) has lower left-corner coordinates (7080, 7200), a width of 6920, and a height of 9490. The starting node (node 1) has coordinates (9860, 14152). A salesman starts at 9 am and

Table 3.1: Record-to-record travel algorithm for the TDTSP.

Algorithm 1 Main Program with Multiple Trials

```
begin
  Set bestRecord = null; OptLength = ∞
  for λ = 0.6 to 2.0 step 0.4 do
      generate an initial TSP tour p by the modified Clarke and Wright
      algorithm with parameter λ
      use record-to-record travel on p to improve the solution
      if p.length < OptLength
        then bestRecord = p; OptLength = p.length;
end
```

Algorithm 2 Record-to-record Travel for TDTSP

```
input: TSP tour p
output: an improved TSP tour p
begin bestTour = p; deviation = 0.01*p.length; M=5, I=10
  for counter= 1 to M do
      for i = 1 to I do (I loop)
          apply two-opt move and one-point move with record-to-record
          travel on p; uphill moves are allowed
          if no move is performed, break I loop
      apply two-opt move and one-point move to the current solution
      only downhill moves are allowed
      if bestTour.length < p.length
        then bestTour = p;  deviation = 0.01*p.length
end
```

Table 3.1: (continued)

Algorithm 3 Two-opt Move with Record-to-record Travel

```
input: record, deviation
begin
    n = number of nodes
   for i = 1 to n do (I loop)
       for j = i + 1 to n do (J loop)
           consider the two-opt move with edge i and j
           if this is a downhill move
             then make the move and continue with the I loop
              else save this move if, after the move,
                    tourLength < record + deviation
       make the best move in the J loop
end
```

Algorithm 4 One-point Move with Record-to-record Travel

```
input: record, deviation
begin
    n = number of nodes
   for i = 1 to n do (I loop)
       for j = 1 to n(j ≠ i) do (J loop)
           insert the ending node of edge i between edge j (this is a
           one-point move)
           if this is a downhill move
             then make the move and continue with the I loop
              else save this move if, after the move,
                    tourLength < record + deviation
       make the best move in the J loop
end
```

Table 3.1: (continued)

Algorithm 5 Two-opt Move

```
begin
   n = number of nodes; improved = true
  while improve do
        improved = false
        for i = 1 to n do (I loop)
            for j = i + 1 to n do (J loop)
                consider the two-opt move for edge i and j
                if this is a downhill move
                  then improve = true
                       make the move and continue with the I loop
end
```

Algorithm 6 One-point Move

```
begin
   n = number of nodes; improved = true
  while improved do
        improve = false
        for i = 1 to n do (I loop)
            for j = 1 to n(j ≠ i) do (J loop)
                consider the one-point move for edge i and j
                if this is a downhill move
                  then improve = true
                       make the move and continue with the I loop
end
```

Table 3.2: Chained Lin-Kernighan algorithm for the TDTSP.

Algorithm 7 Chained Lin-Kernighan

```
begin
  T = the initial tour. bestObj = length(T)
  while stopping rule is not satisfied do
        Linkern(T)
        if length(T) < bestObj
          then bestObj = length(T)
        perturb T
end
```

Algorithm 8 Linkern

```
input:tour T
begin
  level=0
  put each node of T into a queue q
  while q is not empty do
        t1 = q.pop()
        t2 = next(t1)
        record = length(T)
        rval = improve(t1, t2, level, record, q)
        if rval > 0
          then push t1, t2 into q
end
```

Table 3.2: (continued)

Algorithm 9 Improve

```
Input:
 t1, t2 : two consecutive nodes in the tour
 level : current recursive call
 record : best tour length so far
 q : the node queue
Output: 0 if no improvement has been found; positive otherwise
begin
   rval = 0
   if level ≥ 4
     then return 0
      else
          find a set of nodes S belonging to the neighbor set of t2
           that will yield a promising two-opt move
          for each node t3 ∈ S do
          t4 = prev(t3)
          make two-opt move (t1, t2, t4, t3), that is,
          reverse the nodes between t2 and t4 inclusively
          update the tour length
          if the new length < record
            then update record
                 rval = 1
            rval = rval + improve(t1, t4, level+1, record)
            if rval = 0
              then undo the two-opt move (t1, t2, t4, t3)
                   restore the old tour length
               else push t3 and t4 into q
                   return 1
end
```

finishes at 3 pm. The traffic jam occurs from 12 pm to 3 pm. The travel speed is computed by dividing the total distance of the optimal TSP tour (118293.524) by the number of hours in the workday (six). The travel speed is held constant for all values of the jam factor. It is not necessary that a tour fills the work day exactly.

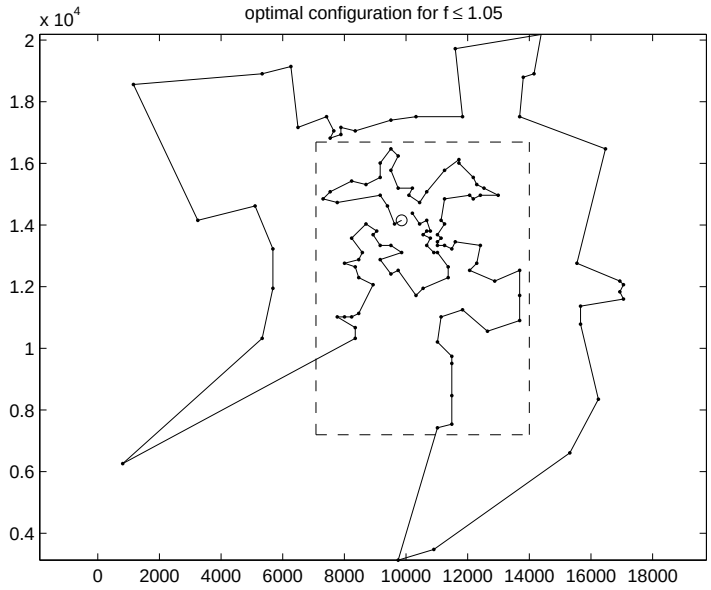


Figure 3.3: Best-known solution for $f \leq 1.05$.

3.3.1 Old assumption

In Bentner et al. [5] and Schneider [36], traffic jams occur on all edges with *both* end points in the rectangle. We refer to this as the *old assumption*.

We apply our record-to-record travel algorithm to Bier127 with the old assumption. The computational results are given in Table 3.3. We present results for 20 different values of the jam factor. The computation time is in minutes on an Athlon 1 GHz computer. We see that RTR finds the best-known solution for four jam factors (1, 1.20, 1.38, and 1.39) and, on average, is 0.30% above the best-known solution. In Figure 3.3 to Figure 3.6, we show the best-known solution for different values of the jam factor f. The salesman starts the tour at the circle. The last edge is not shown in order to indicate the direction of the tour.

Table 3.3: Computational results for RTR on Bier127 with the old assumption.

Jam Factor	*Time(min)*	*Tour Length*	*Percent above Best known*	*Best-known Solution*
1.00	2.61	118293.524	0.00	118293.524
1.03	4.31	118796.154	0.04	118749.356
1.04	2.74	119971.191	0.90	118901.300
1.05	3.29	119503.279	0.38	119053.244
1.06	3.98	119857.323	0.60	119153.582
1.10	2.88	119957.387	0.54	119313.720
1.20	3.61	119714.065	0.00	119714.065
1.30	3.17	120637.093	0.44	120114.410
1.38	2.73	120434.687	0.00	120434.687
1.39	3.10	120453.554	0.00	120453.554
1.50	4.22	120617.178	0.04	120571.743
1.60	4.55	121108.329	0.36	120679.186
1.70	3.72	120898.269	0.09	120786.630
1.80	3.03	121195.816	0.25	120894.074
1.90	3.49	121148.519	0.12	121001.518
2.02	5.58	121298.538	0.14	121125.195
3.00	4.34	122222.204	0.91	121125.195
10.00	3.67	121167.051	0.03	121125.195
100.00	4.47	122280.886	0.95	121125.195
2000.00	3.84	121417.575	0.24	121125.195
Average	3.66		0.30	

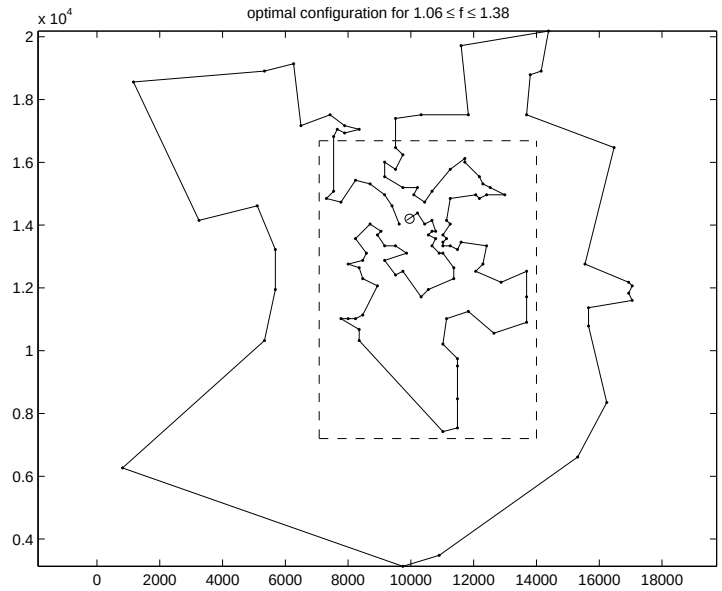


Figure 3.4: Best-known solution for $1.06 \leq f \leq 1.38$.

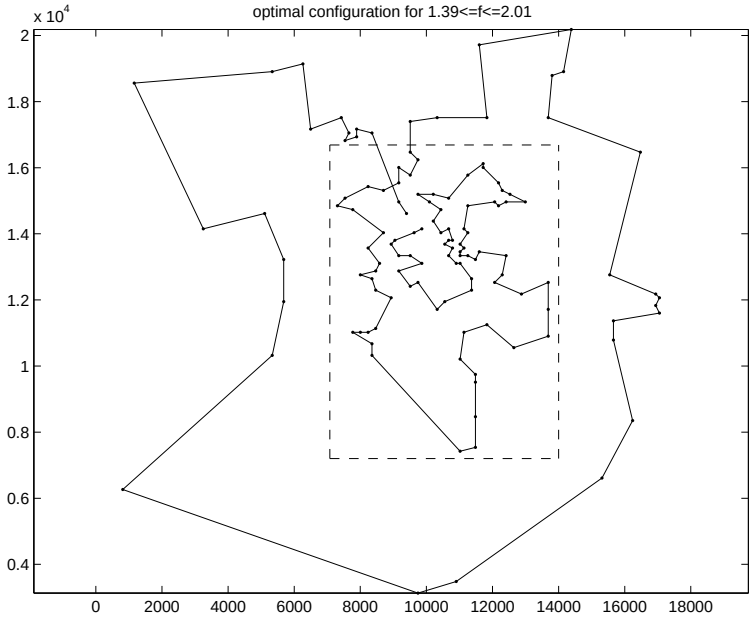


Figure 3.5: Best-known solution for $1.39 \leq f \leq 2.01$.

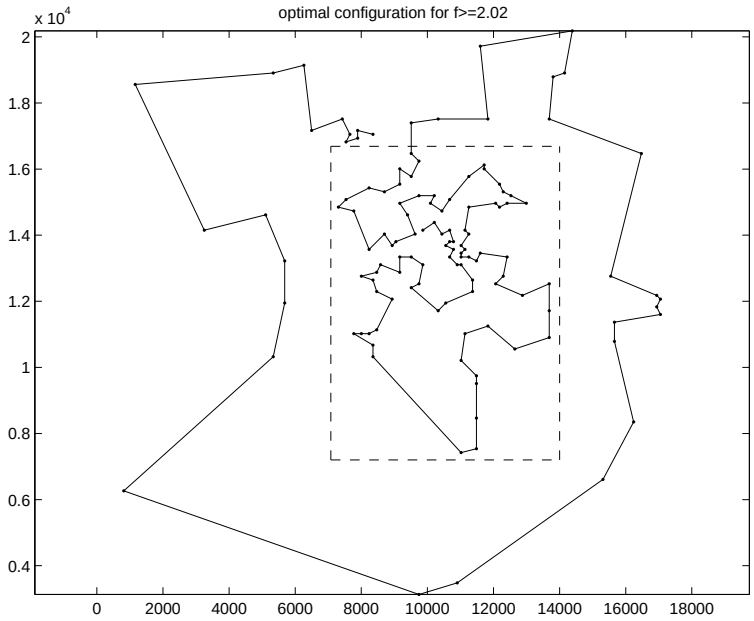


Figure 3.6: Best-known solution for $f \geq 2.02$.

3.3.2 New assumption

With the old assumption, a salesman starts at 9 am and finishes at 3 pm. The traffic jam occurs at noon. An edge ℓ is penalized during the traffic jam only if the following two conditions are satisfied: (1) ℓ is traveled after noon and (2) both end points of ℓ are inside the traffic jam region.

The second condition is not realistic in practice. A salesman might travel along an edge with end points i and j after noon, where i is outside the traffic jam region and j is inside the region. Under condition 2 of the old assumption, this edge would not be penalized. We would like to penalize this edge in proportion to the length inside the traffic jam region and use the following revised conditions, called the *new assumption*. An edge ℓ is penalized during the traffic jam only if the following two conditions are satisfied: (1) some part of ℓ is traveled after noon and (2) some part of ℓ is inside the traffic jam region and only that part is penalized. Thus, we penalize that part of ℓ inside the traffic jam

region that is traveled after noon.

We apply our record-to-record travel algorithm and our chained Lin-Kernighan algorithm to Bier127 with the new assumption. The computational results are given in Table 3.4 and Table 3.5. We present results for 14 different values of the jam factor. The computation time is in minutes on an Athlon 1 GHz computer. In Table 3.5, the results for chained Lin-Kernighan are from 10 runs of the algorithm (randomness is introduced into each run since we use the double-bridge kick). We report the best tour length found in the 10 runs and the total running time for the 10 runs. We see that CLK generates nearly all of the best-known solutions. RTR performs nearly as well — it quickly generates solutions that are, on average, within 0.70% of the best-known solutions.

We now examine the objective function of the TDTSP. Let the value of the objective function be defined by $Obj = L_1 + \alpha f$, where L_1 is the total distance traveled without being penalized, α is the total distance traveled being penalized, and f is the jam factor. We see that Obj is a linear function of f that can be rewritten as $Obj = (L_1 + \alpha) + \alpha(f - 1)$. The first term represents the objective function of the underlying TSP and the second term represents the time dependent part. If we denote the first term by L_0, then each configuration in the TDTSP is uniquely determined by the pair (L_0, α).

If we allow the jam factor to change continuously, there are several boundary values for f where the best configuration for the TDTSP changes. In Table 3.6, we give six boundary intervals for Bier127 with the new assumption. In Figure 3.7 to Figure 3.14, we show the best configuration for eight different boundary intervals. The bold edges are traveled after noon in the traffic jam region. We see that as the value of the jam factor increases (in moving from Figure 3.7 to Figure 3.14) the number of bold edges decreases, that is, the

Table 3.4: Computational results for RTR on Bier127 with the new assumption.

Jam Factor	*Time(min)*	*Tour Length*	*Percent above Best known*	*Best-known Solution*
1.00	0.83	118838.502	0.46	118293.524
1.05	0.86	119593.645	0.39	119125.478
1.06	1.06	119291.869	0.03	119250.182
1.18	1.15	120449.356	0.44	119923.796
1.19	0.83	120803.542	0.70	119968.895
1.70	0.81	123232.728	1.26	121697.112
1.71	1.16	123067.541	1.10	121728.121
2.42	1.13	125203.619	1.07	123874.262
2.43	1.01	125207.611	1.05	123901.699
3.74	1.24	128904.044	1.11	127491.060
3.75	1.07	129093.330	1.24	127518.019
6.53	1.02	135561.752	0.52	134858.670
6.54	1.07	135390.528	0.38	134883.255
132.94	1.49	457651.386	0.01	457624.926
Average	1.05		0.70	

Table 3.5: Computational results for CLK on Bier127 with the new assumption.

Jam Factor	*Time(min)*	*Tour Length*	*Percent above Best known*	*Best-known Solution*
1.00	4.08	118293.524	0.00	118293.524
1.05	4.16	119125.478	0.00	119125.478
1.06	4.53	119250.182	0.00	119250.182
1.18	4.78	119923.796	0.00	119923.796
1.19	4.31	119968.895	0.00	119968.895
1.70	4.20	121697.112	0.00	121697.112
1.71	5.04	121728.121	0.00	121728.121
2.42	4.80	123874.262	0.00	123874.262
2.43	4.96	123901.699	0.00	123901.699
3.74	4.72	127491.060	0.00	127491.060
3.75	4.82	127518.019	0.00	127518.019
6.53	5.19	134916.352	0.04	134858.670
6.54	4.53	134883.255	0.00	134883.255
132.94	5.35	457624.926	0.00	457624.926
Average	4.69		0.003	

Table 3.6: Boundary intervals for the jam factor.

Boundary Intervals for Jam Factor f	L_0	α
[1.00,1.05]	118293.524	16639.086
[1.06,1.18]	118913.374	5613.453
[1.19,1.70]	119325.049	3388.663
[1.71,2.42]	119581.980	3022.734
[2.43,3.74]	119983.542	2739.969
[3.75,6.53]	120256.583	2640.522
[6.54,132.94]	120737.774	2553.335
[132.95,∞]	121187.037	2550.131

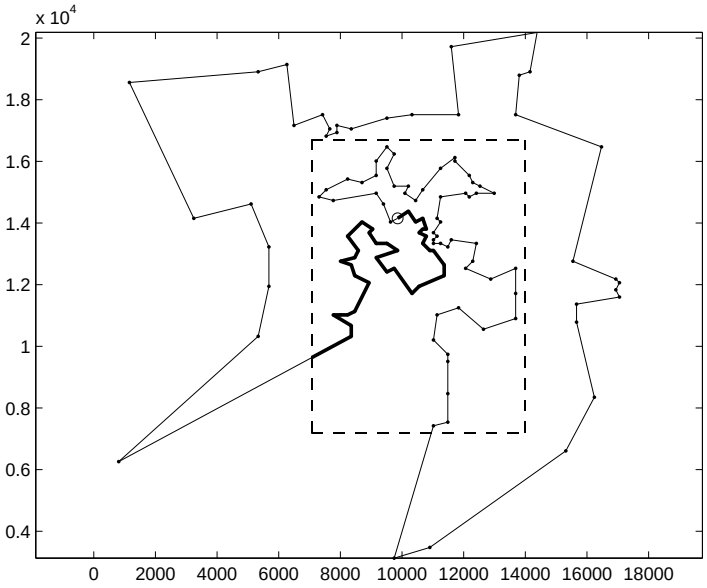


Figure 3.7: Best-known solution for $f \leq 1.05$.

salesman travels fewer edges after noon in the traffic jam region since these edges incur a high penalty. Stated differently, as the value of f increases, L_0 increases slightly and α decreases rapidly.

3.4 Time dependent vehicle routing problem

In the traditional vehicle routing problem, we need to generate a sequence of deliveries for fixed-capacity vehicles in a homogeneous fleet based at a single depot so that all customers are serviced and the total distance traveled by the fleet is minimized. In the time dependent vehicle routing problem (TDVRP), we define a traffic jam region, so that, at a specific time, the center of the city becomes congested and travel time between customers in the region takes longer.

To illustrate the TDVRP, we use the 50-node benchmark vehicle routing problem of Christofides et al. [10]. The traffic jam region is a rectangle with lower left-corner coordinates (15, 20), a width of 30, and a height of 40. A truck

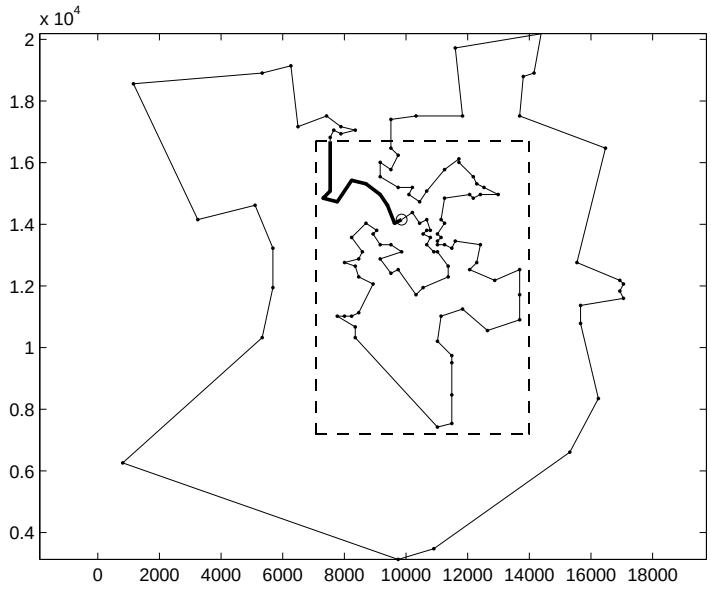


Figure 3.8: Best-known solution for $1.06 \leq f \leq 1.18$.

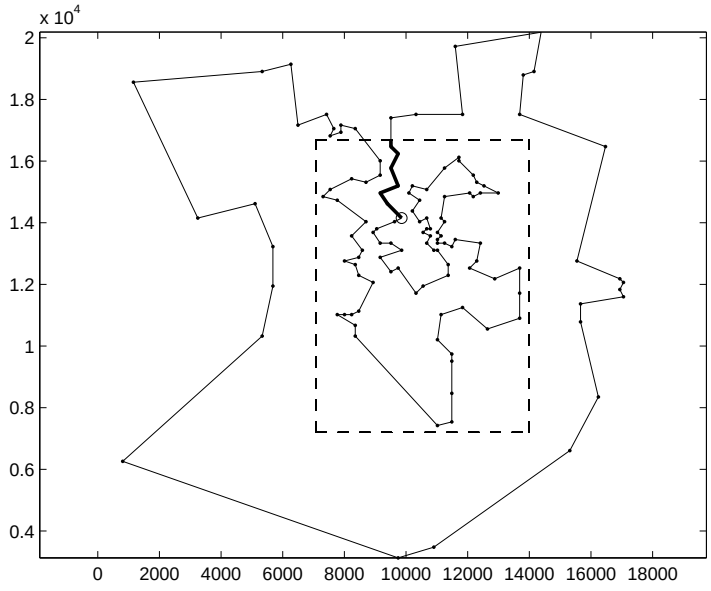


Figure 3.9: Best-known solution for $1.19 \leq f \leq 1.70$.

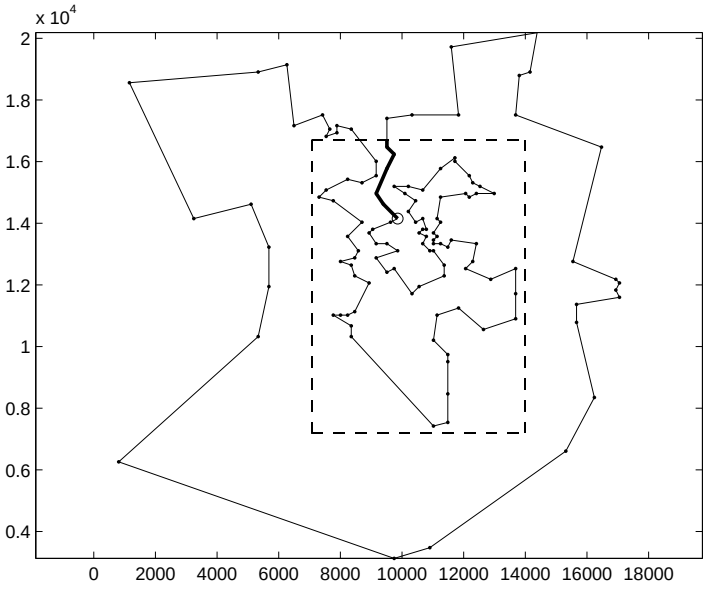


Figure 3.10: Best-known solution for $1.71 \leq f \leq 2.42$.

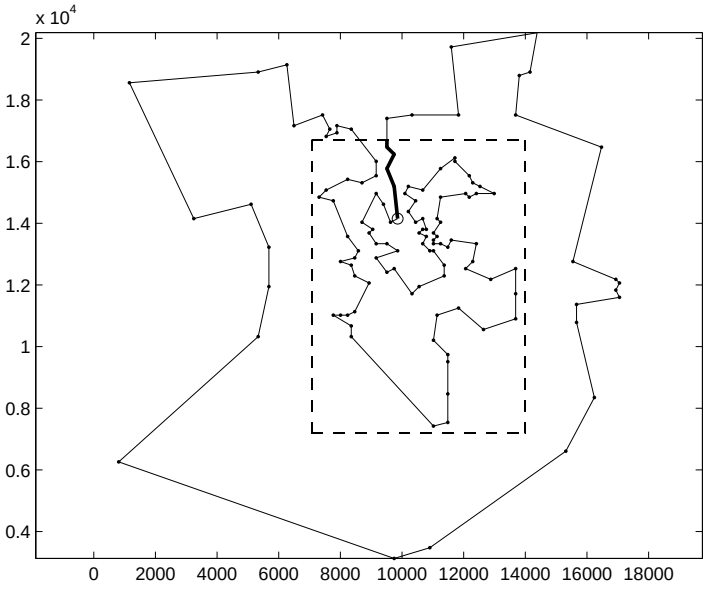


Figure 3.11: Best-known solution for $2.43 \leq f \leq 3.74$.

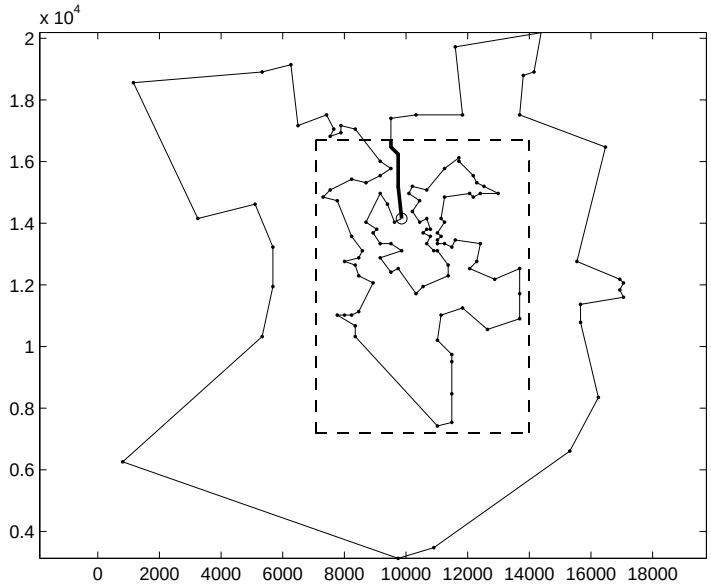


Figure 3.12: Best-known solution for $3.75 \leq f \leq 6.53$.

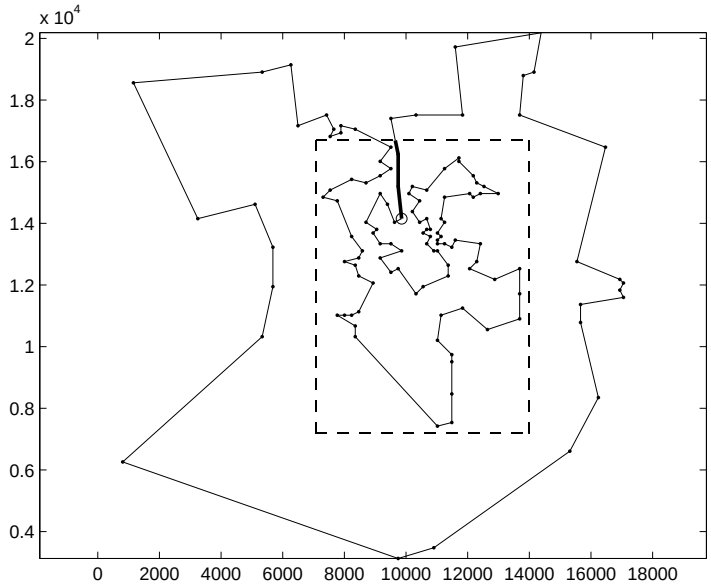


Figure 3.13: Best-known solution for $6.54 \leq f \leq 132.94$.

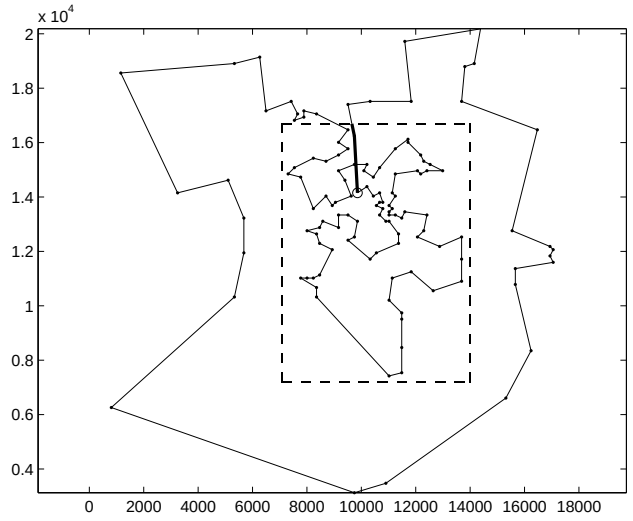


Figure 3.14: Best-known solution for $f \geq 132.95$.

starts delivery at 8 am and finishes at 5 pm. The traffic jam starts at 12 pm. The travel speed is computed by dividing the distance of the longest route in the optimal solution to the VRP by the number of hours in the work day (nine). The travel speed is held constant for all values of the jam factor. It is not necessary that a route fills the work day exactly.

We applied our record-to-record travel algorithm (Li et al. [28]) to the 50-node problem with the new assumption. The results are illustrated in Figure 3.15 to Figure 3.18.

As f increases in value, we see that fewer customers are serviced in the traffic jam region after noon (the bold edges are traveled after noon) and the value of L_0 increases. The average running time for the four different jam factors is about 2.4 minutes on an Athlon 1 GHz computer.

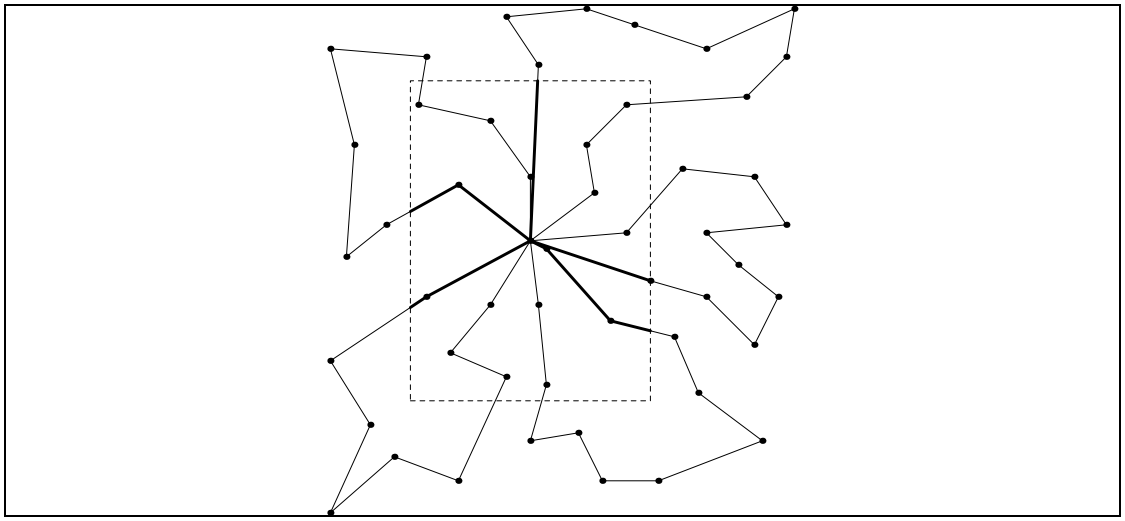


Figure 3.15: Best-known solution for $f \leq 1.02, L_0 = 524.61$.

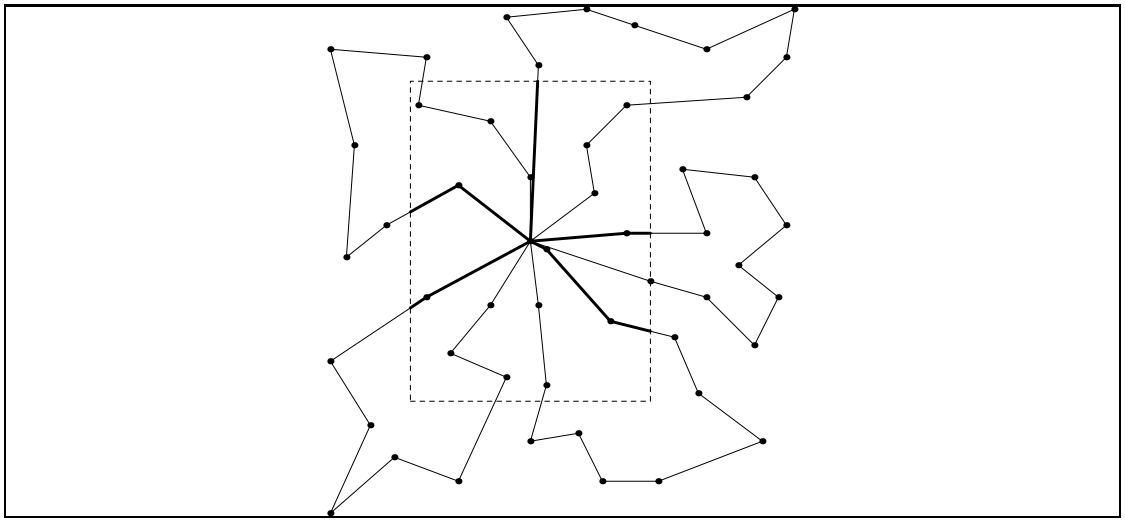


Figure 3.16: Best-known solution for $1.03 \leq f \leq 1.77, L_0 = 524.63$.

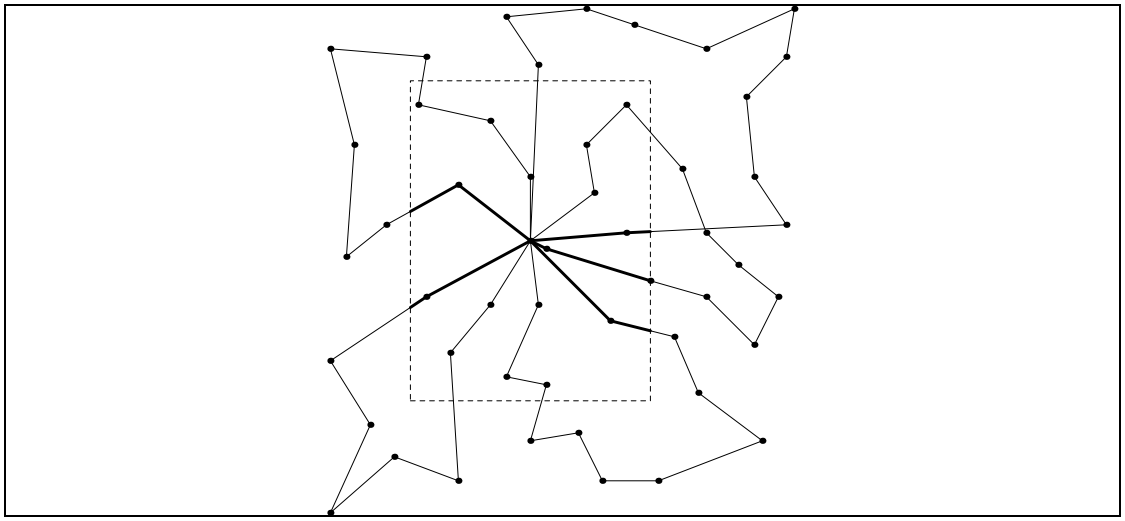


Figure 3.17: Best-known solution for $1.78 \leq f \leq 2.27, L_0 = 527.98$.

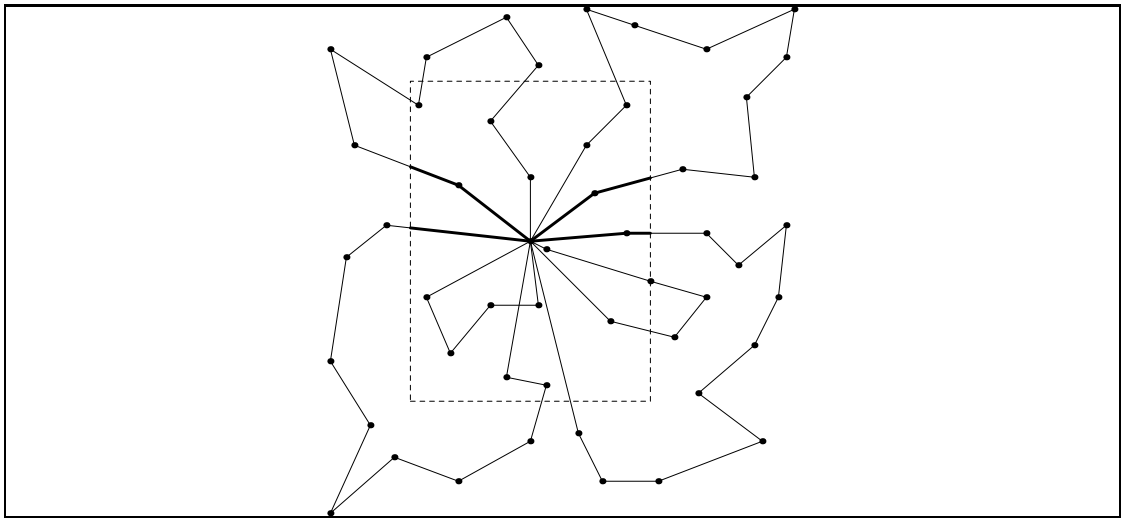


Figure 3.18: Best-known solution for $2.28 \leq f \leq 3.75, L_0 = 553.88$.

3.5 Conclusions

In this chapter, we described the time dependent traveling salesman problem, extended the problem to include a realistic travel assumption, and developed two solution algorithms — record-to-record travel and chained Lin-Kernighan — to solve it. We applied both algorithms to a problem from the literature and showed how the configurations of tours changed as the value of the jam factor was increased. We extended the notion of time dependency to the vehicle routing problem and used record-to-record travel to solve a benchmark problem with a traffic jam region. In the future work, we will model and solve problems that involve two rush hours (for example, a morning rush hour and an evening rush hour).

Chapter 4

Analysis of the Noisy Euclidean Traveling Salesman Problem

Consider a truck that visits n households each day. The specific households (and their locations) vary slightly from one day to the next. In the noisy traveling salesman problem, we develop a rough (skeleton) route that can then be adapted and modified to accommodate the actual node locations that need to be visited from day to day. In this chapter, we conduct extensive computational experiments on problems with $n = 100$, 200, and 300 nodes in order to compare several heuristics for solving the noisy traveling salesman problem including a new method based on quad trees. We find that the quad tree approach generates high-quality results quickly.

4.1 Introduction

The Euclidean Traveling Salesman Problem (TSP) is a well-known combinatorial optimization problem that is easy to state – given a complete graph $G = \{N, E\}$, where N is the set of nodes, E is the set of edges, and the distances are Euclidean and symmetric, find the shortest tour that visits every node in N exactly once – and difficult to solve optimally. Algorithmic

developments and computational results are covered by Junger et al. [24], Johnson and McGeoch [23], Coy et al. [13], and Pepper et al. [31].

Recently, Braun and Buhmann [8] introduced the following variant of the TSP which they refer to as the Noisy Traveling Salesman Problem (NTSP).

> Consider a salesman who makes weekly trips. At the beginning of each week, the salesman has a new set of appointments for the week, for which he has to plan the shortest round-trip. The location of the appointments will not be completely random, because there are certain areas which have a higher probability of containing an appointment, for example cities or business districts within cities. Instead of solving the planning problem each week from scratch, a clever salesman will try to exploit the underlying density and have a rough trip pre-planned, which he will only adapt from week to week.

Braun and Buhmann viewed each node in a TSP as being sampled from a probability distribution, so that many TSP instances could be drawn from the same distribution. They used the sampled instances to build an *average trajectory* that was not forced to visit every node. For Braun and Buhmann, the average trajectory was "supposed to capture the essential structure of the underlying probability density." The average trajectory would then be used as the "seed" to generate an actual tour for each new week of appointments. Braun and Buhmann applied their average trajectory approach to a problem with 100 nodes.

In this chapter, we conduct extensive computational experiments using three different data sets with different underlying structures to test Braun and Buhmann's approach, a simple convex hull, cheapest insertion heuristic, and a new heuristic (called the *quad tree* approach) that we develop for generating an average trajectory.

To make the problem more concrete, consider the following. Each day, companies such as Federal Express and United Parcel Service send thousands of

trucks to make local deliveries to households all across the United States. Let's focus on one of these trucks. Each day, it visits approximately the same number of households in the same geographic region. The specific households may change from one day to the next, but the basic outline of the route remains the same. For example, if the truck visits the household located at 10 Main Street today, it might visit 15 Main Street instead (across the street) tomorrow. In the noisy traveling salesman problem, we develop a rough (skeleton) route that can then be adapted and modified to accommodate the actual node locations that need to be visited from day to day.

We point out that the NTSP is similar to, but different from, the Probabilistic Traveling Salesman Problem (PTSP). In the PTSP, only a subset k ($0 \leq k \leq n$) out of n demand points needs to be visited on a daily basis. The demand point locations are known with certainty (see Jaillet [22] for details).

In this chapter, we conduct extensive computational experiments using three different data sets with different underlying structures to test Braun and Buhmann's approach, a simple convex hull, cheapest insertion heuristic, and a new heuristic (called the *quad tree* approach) that we develop for generating an average trajectory.

In Section 4.2, we describe Braun and Buhmann's approach. We show how they generate an average trajectory and then use it to produce an actual tour. We present their limited computational results. In Section 4.3, we conduct our extensive computational experiments. In Section 4.4, we develop the quad tree approach and test its performance. In Section 4.5, we apply the quad tree approach to the Probabilistic Traveling Salesman Problem (PTSP) and compare the results with the traditional approach to the PTSP. In Section 4.6, we present our conclusions and directions for future research.

4.2 Average trajectory approach

First, we provide the background that is needed to develop an average trajectory. We then generate an average trajectory for a small problem with seven nodes. Second, we give the details of the average trajectory approach and present Braun and Buhmann's computational results.

4.2.1 Generating an average trajectory

We demonstrate how to generate an average trajectory for a problem with seven nodes. The coordinates of the seven nodes are given in Table 4.1. Consider the following three trajectories that pass through all seven nodes:

$$\phi_1 = [1, 2, 3, 4, 5, 6, 7], \phi_2 = [1, 2, 4, 5, 3, 7, 6], \phi_3 = [2, 3, 6, 5, 4, 7, 1].$$

Each of these (ϕ_1, ϕ_2, and ϕ_3) is a tour. As we will see, not all trajectories are tours. The distance between two specific trajectories is illustrated below.

$$\begin{aligned}
\text{distance}(\phi_1, \phi_2) &= \parallel \phi_1 - \phi_2 \parallel \\
&= \max\{\mid x_1 - x_1 \mid, \mid y_1 - y_1 \mid\} + \max\{\mid x_2 - x_2 \mid, \mid y_2 - y_2 \mid\} \\
&\quad + \max\{\mid x_3 - x_4 \mid, \mid y_3 - y_4 \mid\} + \max\{\mid x_4 - x_5 \mid, \mid y_4 - y_5 \mid\} \\
&\quad + \max\{\mid x_5 - x_3 \mid, \mid y_5 - y_3 \mid\} + \max\{\mid x_6 - x_7 \mid, \mid y_6 - y_7 \mid\} \\
&\quad + \max\{\mid x_7 - x_6 \mid, \mid y_y - y_6 \mid\}.
\end{aligned}$$

Note that $\parallel \phi_1 - \phi_2 \parallel$ is defined as the L_1 norm.

Of course, we observe that ϕ_2 is equivalent to a set E of equivalent trajectories. Given symmetry, the tour $1 - 2 - 3$ may be represented by $3 \times 2 = 6$ equivalent trajectories: [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1].

Let E be the set of all $2n$ trajectories equivalent to ϕ_2. Then, using ϕ_1 as a basis of comparison, find the specific trajectory that solves $\min_{\phi \in E} \parallel \phi - \phi_1 \parallel$.

Node	Coordinates	
	x	y
1	3.375	2.375
2	2.875	3.375
3	3.375	4.625
4	3.875	4.875
5	4.875	4.625
6	5.125	3.875
7	4.875	2.625

Table 4.1: Coordinates of a seven-node TSP.

ϕ_2	$\parallel \phi_1 - \phi_2 \parallel$	ϕ_3	$\parallel \phi_1 - \phi_3 \parallel$
[1,2,4,5,3,7,6]	5.50	[2,3,6,5,4,7,1]	8.75
[2,4,5,3,7,6,1]	8.00	[3,6,5,4,7,1,2]	11.75
[4,5,3,7,6,1,2]	11.25	[6,5,4,7,1,2,3]	13.00
[5,3,7,6,1,2,4]	13.50	[5,4,7,1,2,3,6]	13.25
[3,7,6,1,2,4,5]	13.75	[4,7,1,2,3,6,5]	11.75
[7,6,1,2,4,5,3]	11.25	[7,1,2,3,6,5,4]	8.00
[6,1,2,4,5,3,7]	5.75	[1,2,3,6,5,4,7]	2.50
[6,7,3,5,4,2,1]	9.50	[7,4,5,6,3,2,1]	11.00
[7,3,5,4,2,1,6]	9.25	[4,5,6,3,2,1,7]	10.50
[3,5,4,2,1,6,7]	8.50	[5,6,3,2,1,7,4]	11.75
[5,4,2,1,6,7,3]	11.50	[6,3,2,1,7,4,5]	12.00
[4,2,1,6,7,3,5]	11.75	[3,2,1,7,4,5,6]	9.75
[2,1,6,7,3,5,4]	10.50	[2,1,7,4,5,6,3]	6.00
[1,6,7,3,5,4,2]	8.00	[1,7,4,5,6,3,2]	8.00

Table 4.2: Computing shortest distance between ϕ_1 and ϕ_2 and between ϕ_1 and ϕ_3.

Let ϕ_2^* be that trajectory. $\| \phi_2^* - \phi_1 \|$ represents the shortest distance between ϕ_1 and ϕ_2. These calculations are presented in Table 4.2. In particular, the shortest distances between ϕ_1 and ϕ_2 and between ϕ_1 and ϕ_3 are computed and marked accordingly. The average trajectory of ϕ_1 and ϕ_2 becomes $(\phi_1 + \phi_2^*)/2$. Similaryly, the average trajectory of ϕ_1, ϕ_2, and ϕ_3 becomes $(\phi_1 + \phi_2^* + \phi_3^*)/3$.

In Table 4.2, we show how to compute $\| \phi_1 - \phi_2 \|$. For example, $\phi_1 = [1, 2, 3, 4, 5, 6, 7]$ and $\phi_2 = [1, 2, 4, 5, 3, 7, 6]$, we have

$$\| \phi_1 - \phi_2 \| = 0 + 0 + 0.5 + 1.0 + 1.5 + 1.25 + 1.25 = 5.50.$$

In particular, consider the calculation of the final component in $\| \phi_1 - \phi_2 \|$. The final stop in ϕ_1 is node 7 with coordinates of (4.875,2.625). The final stop in ϕ_2 is node 6 with coordinates of (5.125,3.875). The difference in x coordinate is 0.25 and the difference in y coordinate is 1.25, so that the maximum difference is 1.25.

For all 14 trajectories ϕ that are from the same tour as ϕ_2, we select the closest trajectory, that is, we select [1, 2, 4, 5, 3, 7, 6] since it has the minimum norm (5.50). We average the coordinates of ϕ_1 and ϕ_2 to obtain the average trajectory. Similarly, for all 14 trajectories ϕ that are from the same tour as ϕ_3, we select the closest trajectory, that is, we select [1,2,3,6,5,4,7] since it has the minimum norm (2.50), and average the coordinates. The coordinates of the average are given in Table 4.3. In Figure 4.1, we show ϕ_1, ϕ_2, and ϕ_3 and the average trajectory $(\phi_1 + \phi_2)/2$ and $(\phi_1 + \phi_2 + \phi_3)/3$. We see that the two average trajectories do not pass through all seven nodes.

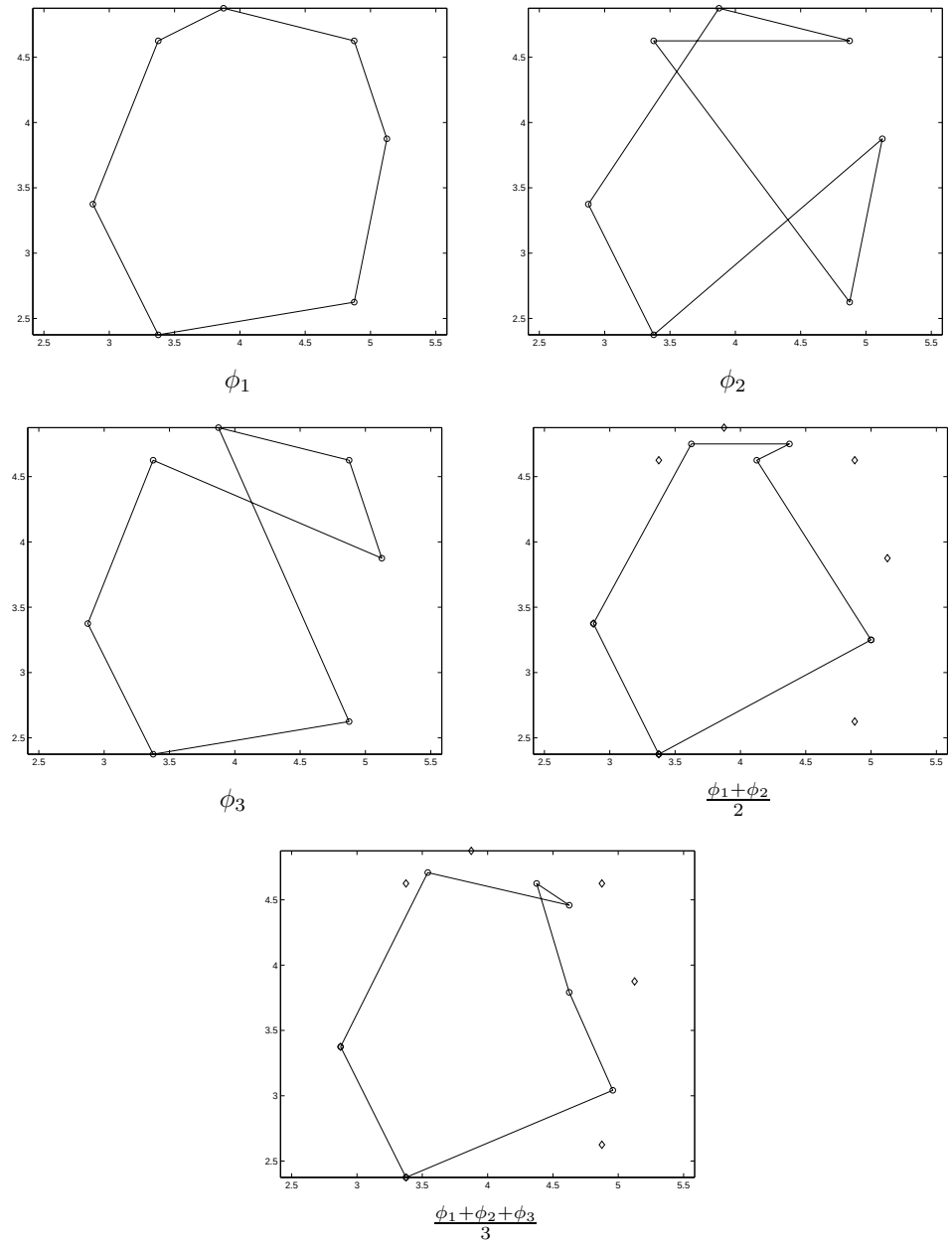


Figure 4.1: Three trajectories (ϕ_1, ϕ_2,ϕ_3) that pass through all seven nodes of TSP and two average trajectories $((\phi_1 + \phi_2)/2, (\phi_1 + \phi_2 + \phi_3)/3)$ that pass through two nodes of the TSP and do not pass through five nodes of the TSP (denoted by the unconnected diamonds).

(a) Average of ϕ_1 and ϕ_2.

Node	Coordinates	
	x	y
1	3.375	2.375
2	2.875	3.375
3	3.625	4.750
4	4.375	4.750
5	4.125	4.625
6	5.000	3.250
7	5.000	3.250

(b) Average of ϕ_1, ϕ_2 and ϕ_3.

Node	Coordinates	
	x	y
1	3.375	2.375
2	2.875	3.375
3	3.542	4.708
4	4.625	4.458
5	4.375	4.625
6	4.625	3.792
7	4.958	3.042

Table 4.3: Coordinates of two average trajectories for a seven-node TSP.

4.2.2 Braun and Buhmann's approach

In practice, we face two questions in finding an average trajectory: (1) How do we generate sample trajectories that will produce a good average trajectory? (2) How do we use the average trajectory to generate a tour for a new problem instance? In this section, we describe the approach of Braun and Buhmann [8] to sampling and then generating a tour.

Braun and Buhmann start with one TSP instance and construct a Markov chain whose state space contains all permutations of the nodes. They sample from a Markov chain using random two-opt as the transition between two states and update the Markov chain using the Metropolis algorithm (see Kirkpatrick et al. [25]).

Specifically, let x be the current state of the Markov chain, x' be the next state, $\hat{x}$ be the random two-opt of x, $\delta\ell = \ell(\hat{x}) - \ell(x)$ where $\ell(x)$ is the length of x, and U is a uniform (0,1) random variable. If $\delta\ell < 0$, then $x' = \hat{x}$ with probability one, that is, always accept a downhill move. If $\delta\ell > 0$, then $x' = \hat{x}$ if $\exp(-\delta/T) > U$ where T is the temperature; otherwise $x' = x$. This is referred to as a Markov Chain Monte Carlo simulation (denoted by MCMC).

Braun and Buhmann draw one thousand samples from the Markov chain in

order to generate the average trajectory. Between two consecutive samples, 100 transitions are performed to decouple the samples, that is, to reduce the dependency between the samples.

We summarize Braun and Buhmann's approach below. In Figure 4.2, we show an average trajectory for a 100-node problem that was generated by their approach.

Step 1. Select one instance, find an initial tour, and run MCMC to generate sample tours.
Step 2. Average all sample tours to produce the average trajectory.
Step 3. For each new instance, apply the finite-horizon adaption technique followed by a local search post-processor to generate the final tour.

Step 3 requires a bit of explanation. In order to generate a tour for a new TSP instance, Braun and Buhmann used a *finite-horizon* adaption technique. First, the domain of the mapping for the average trajectory is extended from $1, \ldots, n$ to the interval $[1, n+1)$ which is called the *passing time*. For each node v_i in the new TSP instance, a point on the average trajectory with minimum distance to v_i is identified, and the passing time t_i is computed by linear interpolation. The permutation that sorts t_i, $i = 1, \ldots, n$ gives the initial solution for the new instance. A post-processor performs local optimization to remove intersections. We illustrate the finite-horizon adaption technique in Figure 4.3.

Nodes 1 to 7 (in black) are from the average trajectory, and they are mapped into 1 to 7 on the passing-time axis, respectively. Nodes a to g (in white) are from the new instance. For each node v_i from the new instance, we first find the nearest point on the average trajectory (this is indicated by the straight, dashed line with open arrow heads). Linear interpolation is then used to find the passing-time t_i for a nearest node. Finally, we sort according to t_i to produce the ordering of the nodes in the new instance. For example, in Figure 4.3, the nearest

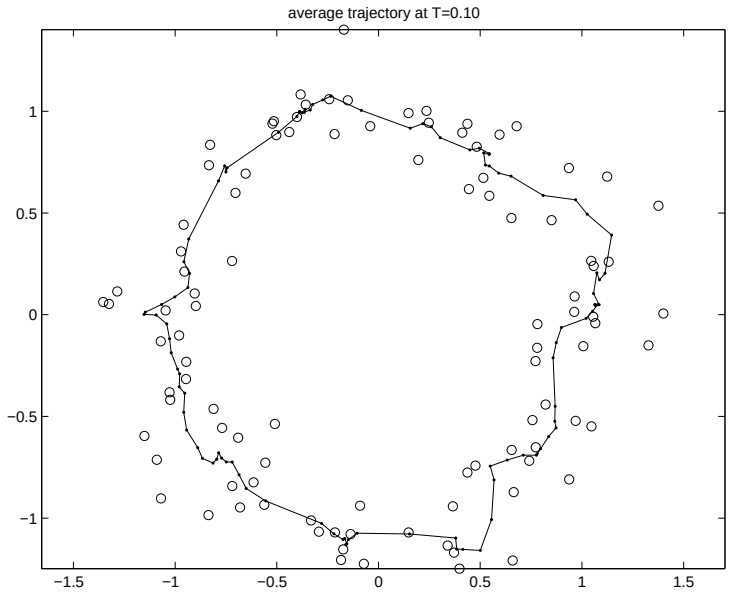


Figure 4.2: Average trajectory for a 100-node problem generated by the approach of Braun and Buhmann.

point on the average trajectory to node a is A, and its t value is 1.3.

We point out that there are several key limitations to Braun and Buhmann's approach. There are three parameters – temperature, number of samples, and interval between samples – whose values need to be set. Results are sensitive to the value of the temperature parameter. For example, average trajectories computed at low temperatures tend to over fit to the noise in the data.

It is not clear how many samples to select in MCMC. Braun and Buhmann used a sample size of 1,000 which requires lots of computation time. In our preliminary experiments, a sample size of 100 performed nearly as well and required much less computation time.

There is very limited computational experience with the three-step approach. Braun and Buhmann reported results for a single 100-node problem.

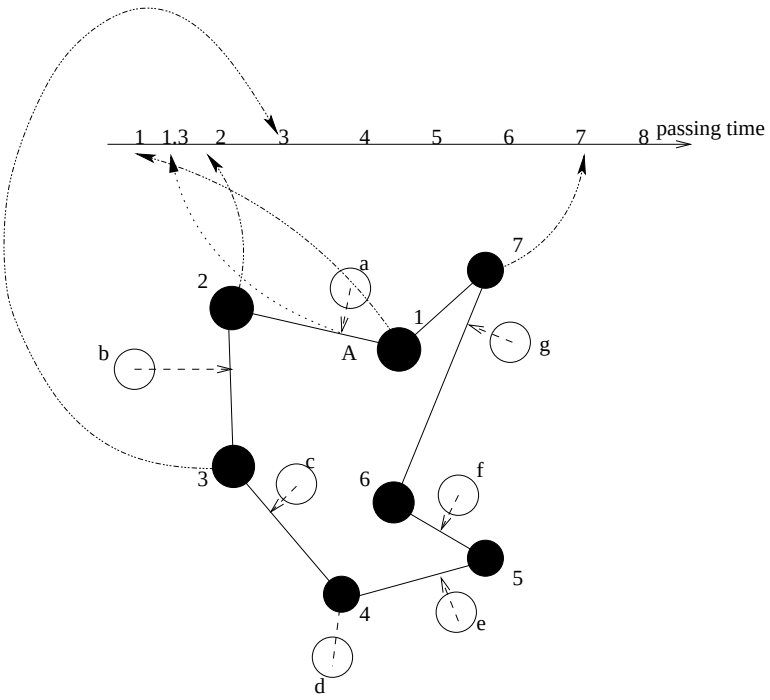


Figure 4.3: Finite-horizon adaption technique to generate a tour for a new instance.

4.3 Computational experiments

In this section, we conduct extensive computational experiments on Braun and Buhmann's approach. We provide the details of our experimental design, report computational results, and develop a new procedure for solving the NTSP.

4.3.1 Description of data sets

In this section, we describe the data sets for our experiments. We use three data sets with different topologies and different sizes (100, 200, and 300 nodes).

4.3.1.1 Data set one

Data set one is the same one used by Braun and Buhmann [8]. The mean demand locations are uniformly located on the unit circle, that is,

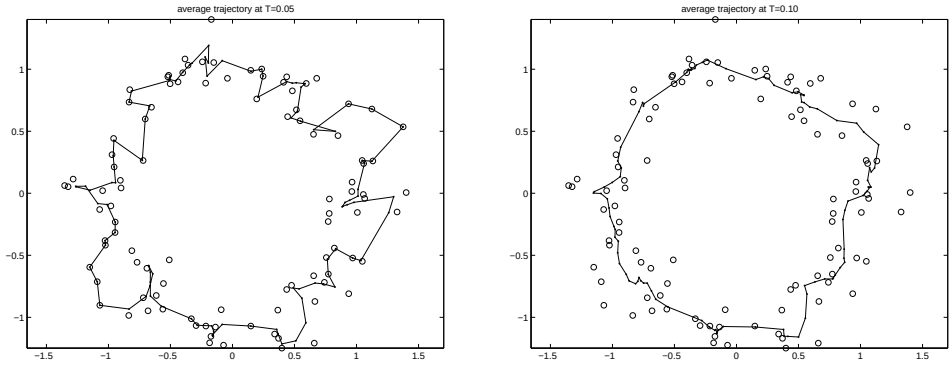
$m_i = (\cos(\frac{2\pi i}{n}), \sin(\frac{2\pi i}{n})), i = 1, \ldots, n$ and the noise is normally distributed, that is, $\mathcal{N}(0, \sigma^2)$. The coordinates of node $v_i = (x_i, y_i)$ are given by

$$\begin{aligned} x_i &= \cos(\frac{2\pi i}{n}) + \zeta_i \\ y_i &= \sin(\frac{2\pi i}{n}) + \eta_i \end{aligned}$$

where $\zeta_i, \eta_i \sim \mathcal{N}(0, \sigma^2)$. In Figure 4.4, we show an instance with average trajectories generated by four different temperatures ($T = 0.05, 0.10, 0.15, 0.20$). We see that temperature acts like a smoothing parameter. If the temperature is low, then the average trajectory tends to overfit the data, i.e., the average trajectory tends to ignore the underlying probabilistic distribution and pays too much attention on the current instance. When the value of the temperature is high, MCMC will not work well, and the average trajectory tends to under fit the data. In Figure 4.4, the middle temperatures seem to work the best.

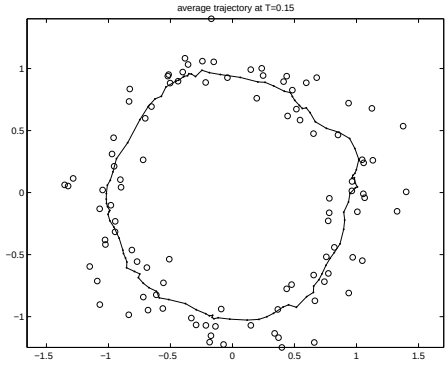
4.3.1.2 Data set two

In the second data set, we have a hierarchical structure. At the first level, there are m clusters of means located around $(\cos(\frac{2\pi i}{m}), \sin(\frac{2\pi i}{m}))$ for $i = 1, \ldots, m$. For each cluster i, there are n means $(r\cos(\frac{2\pi j}{n}) + \cos(\frac{2\pi i}{m}), r\sin(\frac{2\pi j}{n}) + \sin(\frac{2\pi i}{m}))$ with independent Gaussian noise $\mathcal{N}(0, \sigma^2)$. In Figure 4.5, we show an instance from this data set with $m = 6, n = 25, r = 0.25$ and $\sigma^2 = 0.001$, and four different temperatures ($T = 0.02, 0.05, 0.10, 0.20$). The third temperature ($T = 0.10$) seems to work the best.

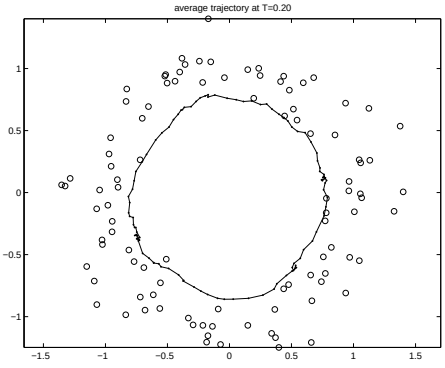


T=0.05

T=0.10



T=0.15



T=0.20

Figure 4.4: Four average trajectories for a problem with $n = 100$ and $\sigma^2 = 0.01$ from data set one.

4.3.1.3 Data set three

In the third data set, we also have a hierarchical structure, but the means are no longer fixed. The x and y coordinates of the mean are randomly sampled from $\mathcal{N}(0, 1)$. Each data point is then sampled around the mean with distribution $\mathcal{N}(0, \sigma^2)$. In Figure 4.6, we show an instance from this data set with $n = 200$ and $\sigma^2 = 0.01$, and nine different temperatures. In this case, it is not clear that any of the temperatures work well.

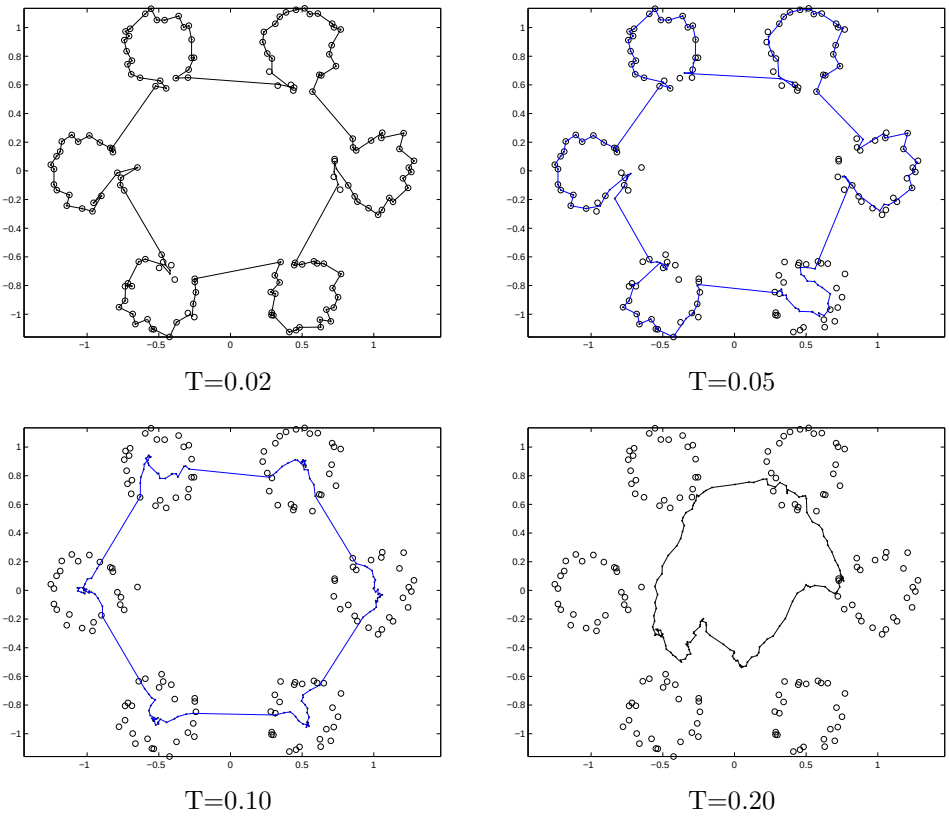


Figure 4.5: Four average trajectories for a problem with $m = 6, n = 25, r = 0.25$, and $\sigma^2 = 0.001$ from data set two.

4.3.2 Experiments with three data sets

For each of the three data sets, we conducted the following experiments. We used three sets of nodes ($n = 100, 200, 300$), three variances ($\sigma^2 = 0.01, 0.005, 0.0025$), and 30 temperatures ($T = 0.01$ to 0.30). At each temperature, we selected a sample of 100 instances in order to compute an average trajectory, generated the initial tour for a new instance using the finite-horizon adaption technique, and applied the two-opt algorithm to the initial tour. For comparison purposes, we generated a tour for each instance using a simple convex hull, cheapest insertion heuristic (denoted by CHCI). CHCI works in the following way: (1) Generate the convex hull of the instance;

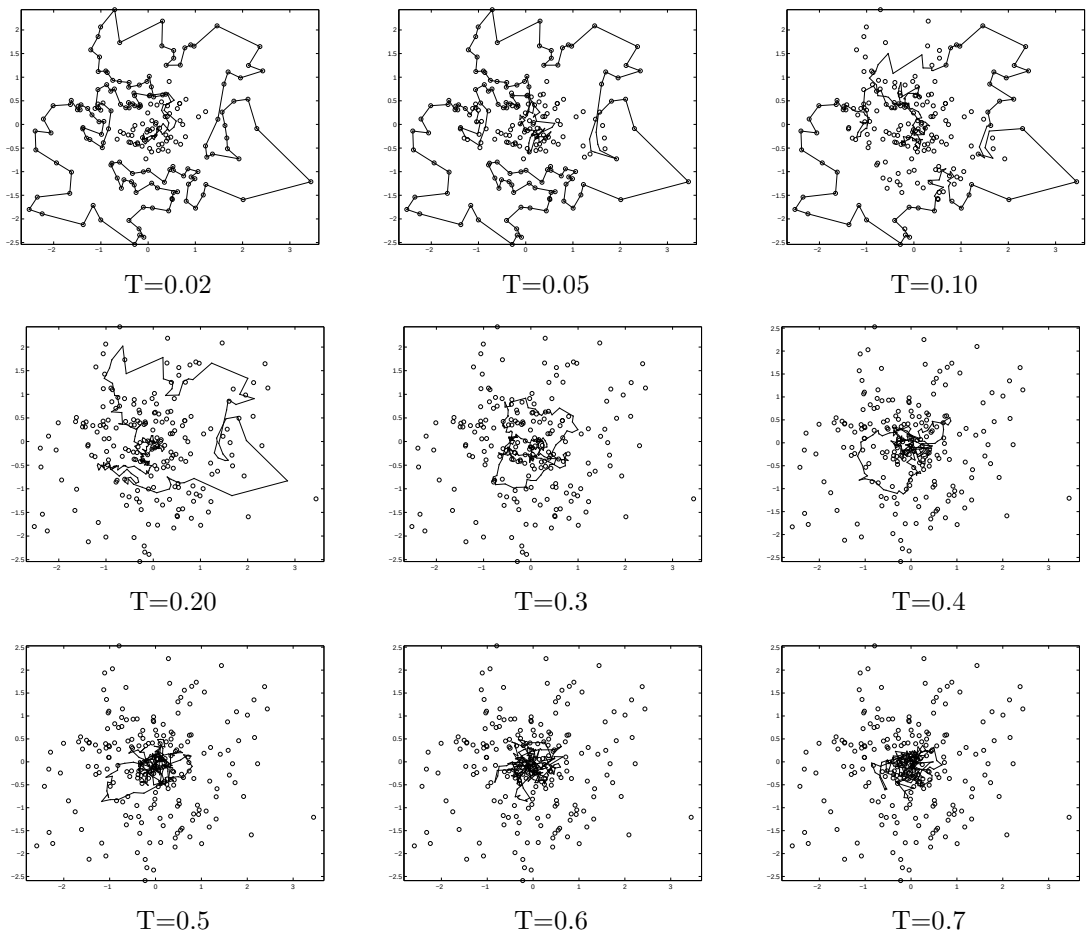


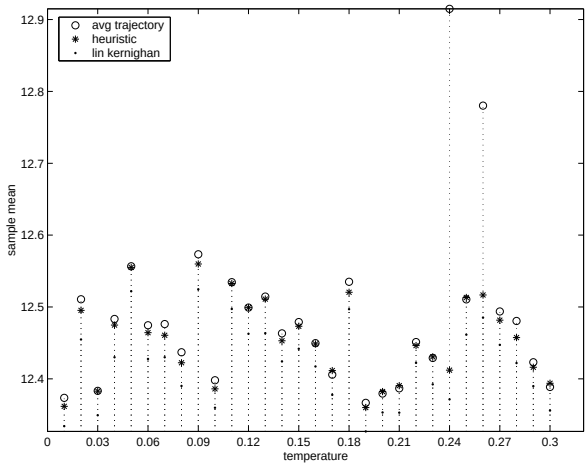
Figure 4.6: Nine average trajectories for a problem with $n = 200$ and $\sigma^2 = 0.01$ from data set three.

(2) For each node that is not a vertex of the convex hull, try to insert it into the convex hull in the lowest-cost way (see Junger et al. [24]). In Figure 4.7 to Figure 4.15, we show the results of our experiments. In Figure 4.7 to Figure 4.9, we show the comparisons of the average trajectory approach versus CHCI for data set one for the three sets of nodes and the three variances. The x-axis gives the temperature and the y-axis gives the percentage that the average trajectory approach generates a lower-cost tour than CHCI. In Figure 4.10 to Figure 4.15, we provide the results for data set two and data set three, respectively. In examining these figures, we make several observations.

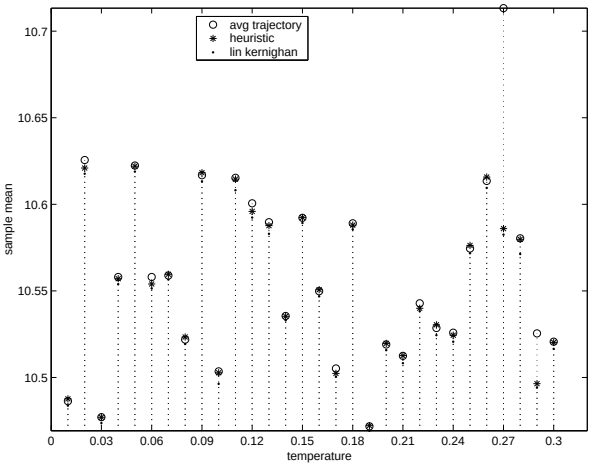
1. When the variance is *small*, the average trajectory approach performs better than CHCI.
2. When the variance is *large*, the average trajectory approach does not perform as well as CHCI.
3. The topology of the data set plays a role in the performance of the average trajectory approach. CHCI is computationally faster on many instances.
4. There is a need to tune the temperature parameter (T) in the average trajectory approach in order to produce good results.
5. Problem size does not play much of a role in the performance of either method.
6. The tour produced by the average trajectory approach may not be visually appealing.

4.4 New heuristic for the NTSP

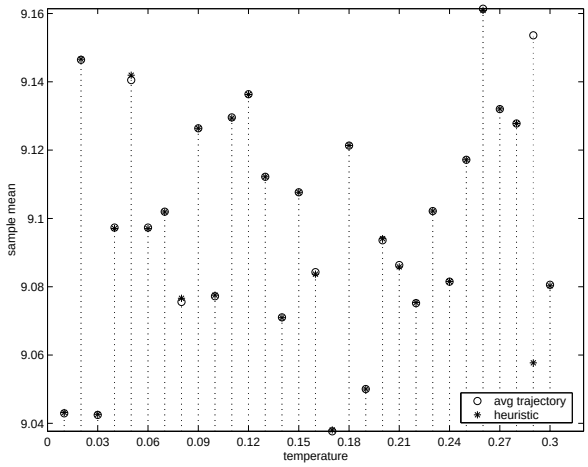
In this section, we develop a new heuristic for the NTSP based on the quad tree and compare its performance to the average trajectory approach.



n=100, $\sigma^2 = 0.01$

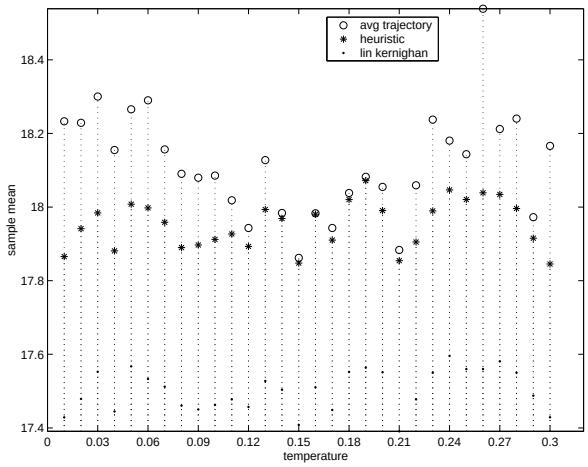


n=100, $\sigma^2 = 0.005$

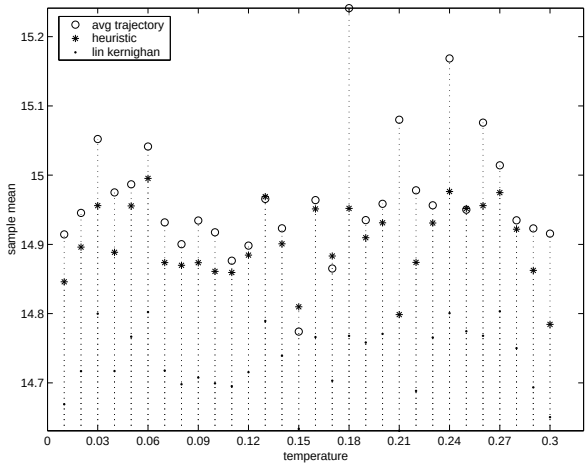


n=100, $\sigma^2 = 0.0025$

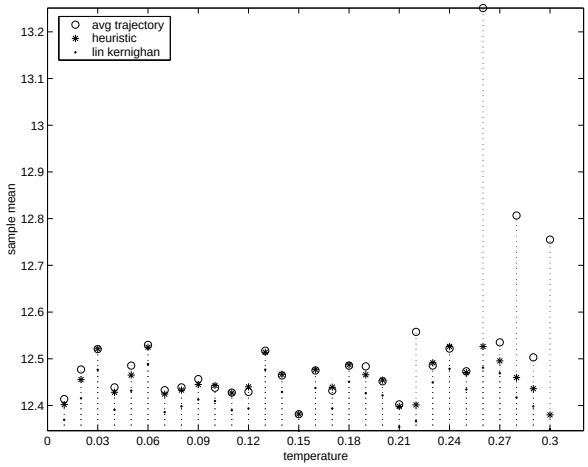
Figure 4.7: Computational results for data set one for $n = 100, \sigma^2 = 0.01, 0.005,$ and 0.0025, and $T = 0.01$ to 0.30.



n=200, $\sigma^2 = 0.01$

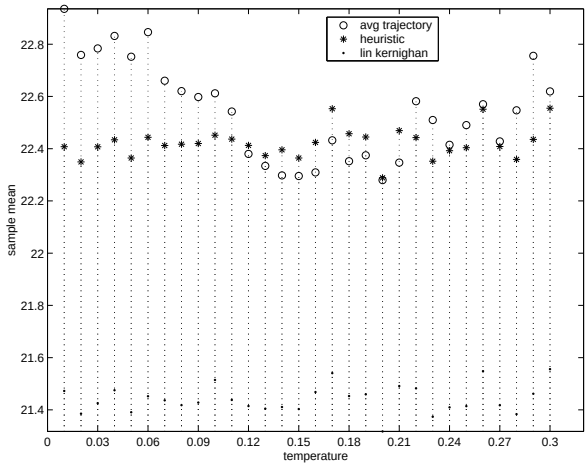


n=200, $\sigma^2 = 0.005$

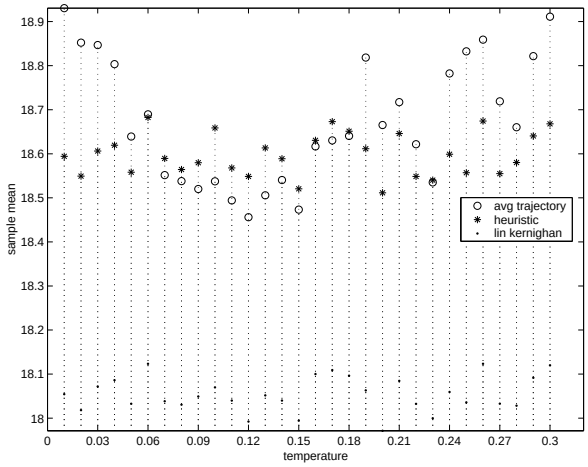


n=200, $\sigma^2 = 0.0025$

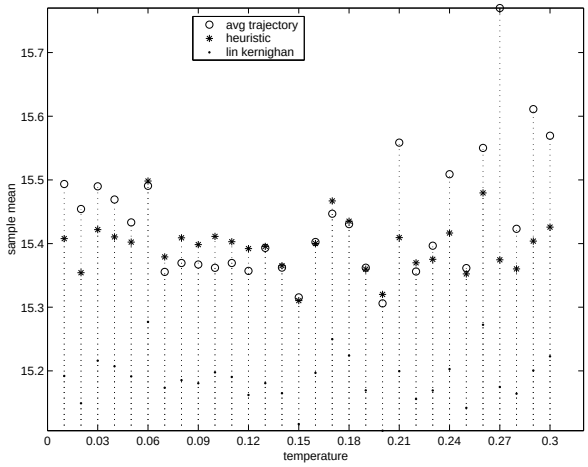
Figure 4.8: Computational results for data set one for $n = 200, \sigma^2 = 0.01, 0.005,$ and 0.0025, and $T = 0.01$ to 0.30.



n=300, $\sigma^2 = 0.01$

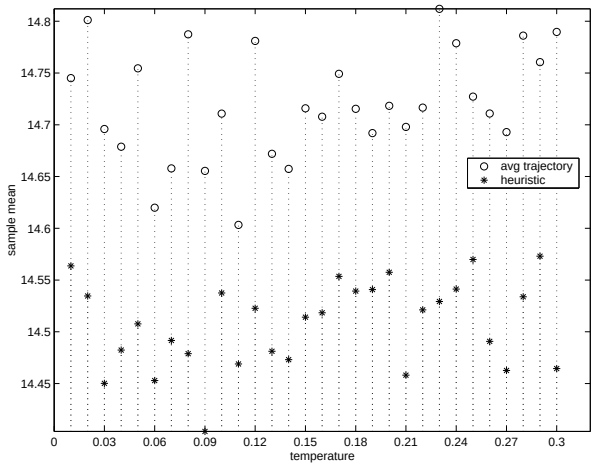


n=300, $\sigma^2 = 0.005$

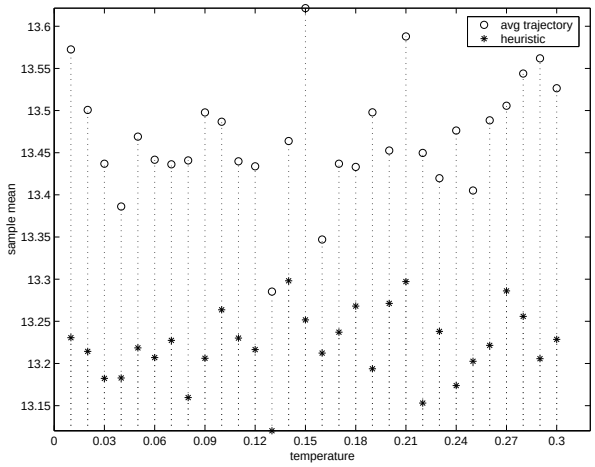


n=300, $\sigma^2 = 0.0025$

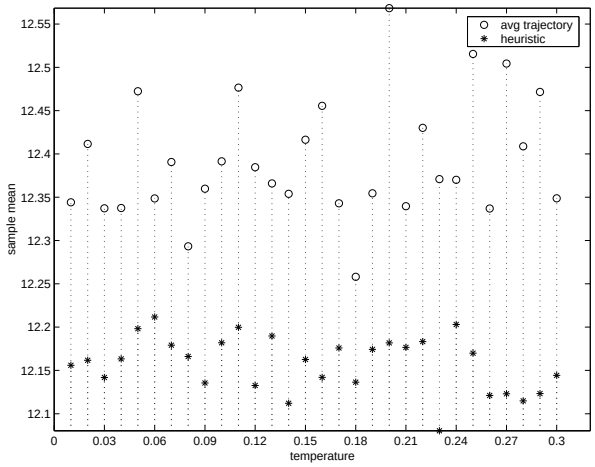
Figure 4.9: Computational results for data set one for $n = 300, \sigma^2 = 0.01, 0.005,$ and 0.0025, and $T = 0.01$ to 0.30.



n=100, $\sigma^2 = 0.01$

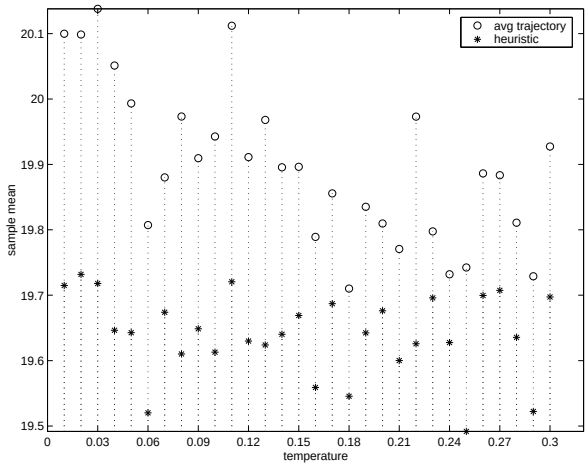


n=100, $\sigma^2 = 0.005$

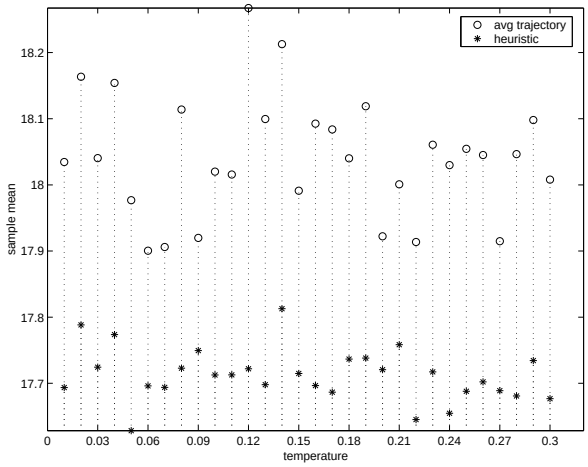


n=100, $\sigma^2 = 0.0025$

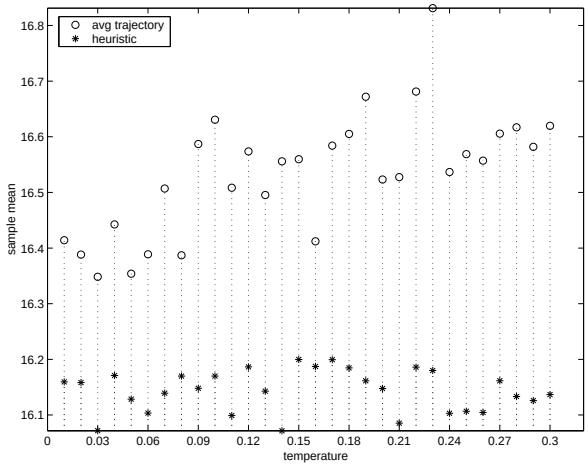
Figure 4.10: Computational results for data set two for $n = 100, \sigma^2 = 0.01, 0.005,$ and 0.0025, and $T = 0.01$ to 0.30.



n=200, $\sigma^2 = 0.01$

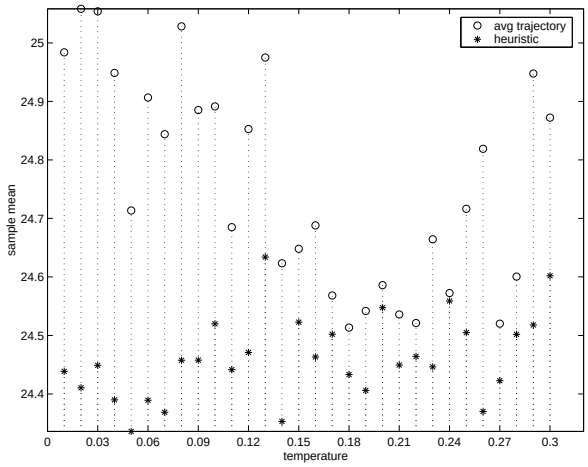


n=200, $\sigma^2 = 0.005$

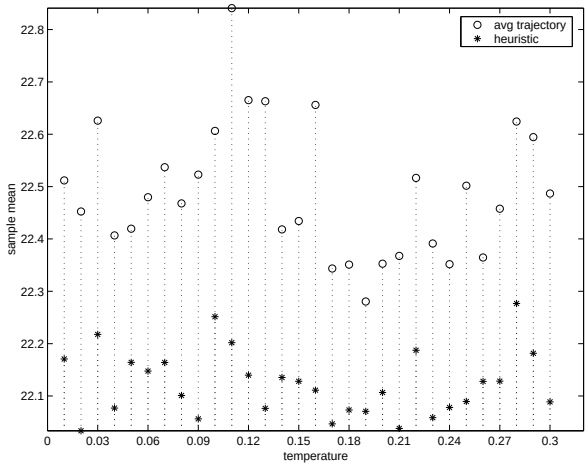


n=200, $\sigma^2 = 0.0025$

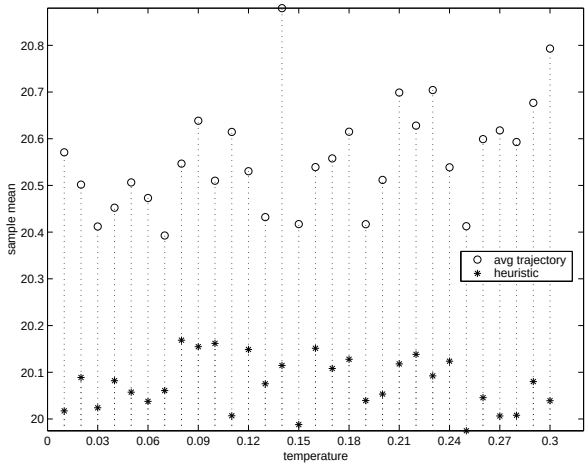
Figure 4.11: Computational results for data set two for $n = 200, \sigma^2 = 0.01, 0.005,$ and 0.0025, and $T = 0.01$ to 0.30.



n=300, $\sigma^2 = 0.01$

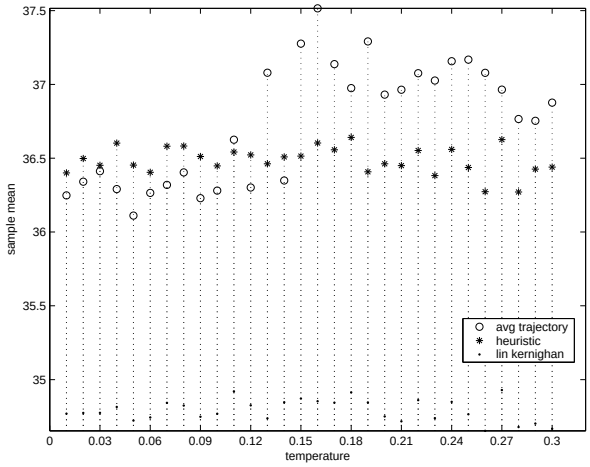


n=300, $\sigma^2 = 0.005$

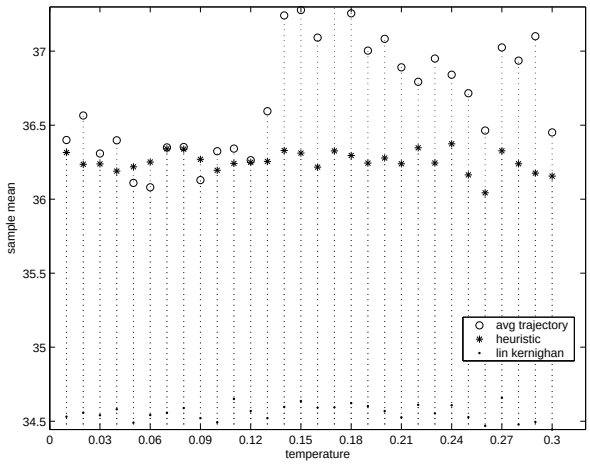


n=300, $\sigma^2 = 0.0025$

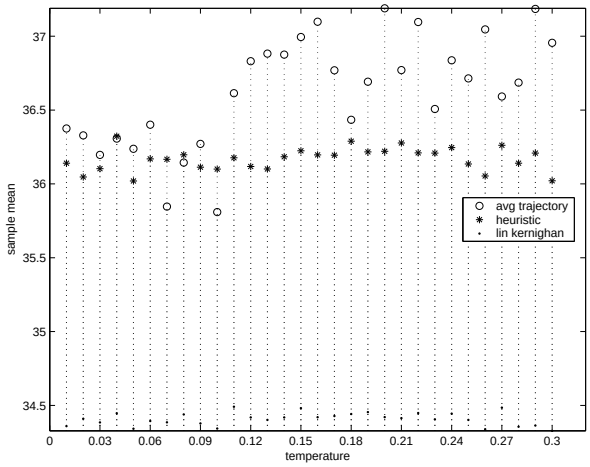
Figure 4.12: Computational results for data set two for $n = 300, \sigma^2 = 0.01, 0.005,$ and 0.0025, and $T = 0.01$ to 0.30.



n=100, $\sigma^2 = 0.01$

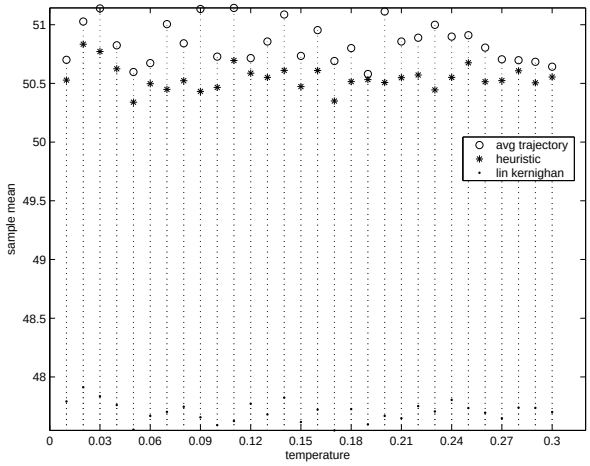


n=100, $\sigma^2 = 0.005$

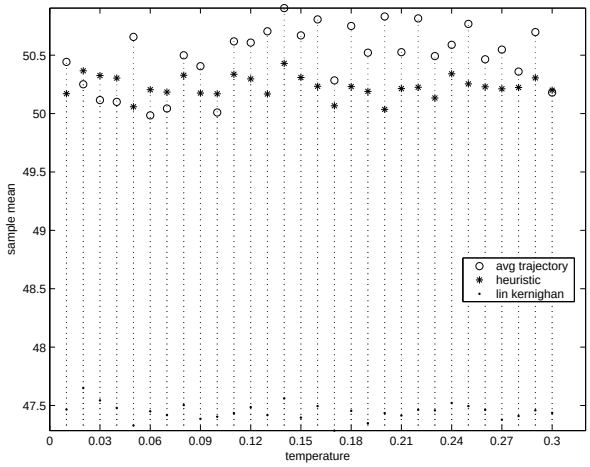


n=100, $\sigma^2 = 0.0025$

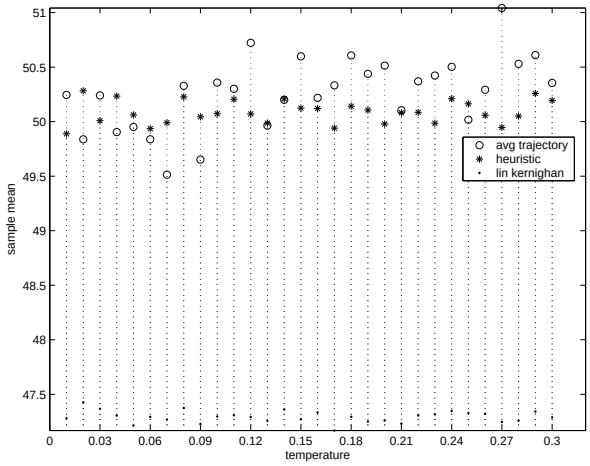
Figure 4.13: Computational results for data set three for $n = 100, \sigma^2 = 0.01, 0.005,$ and 0.0025, and $T = 0.01$ to 0.30.



n=200, $\sigma^2 = 0.01$

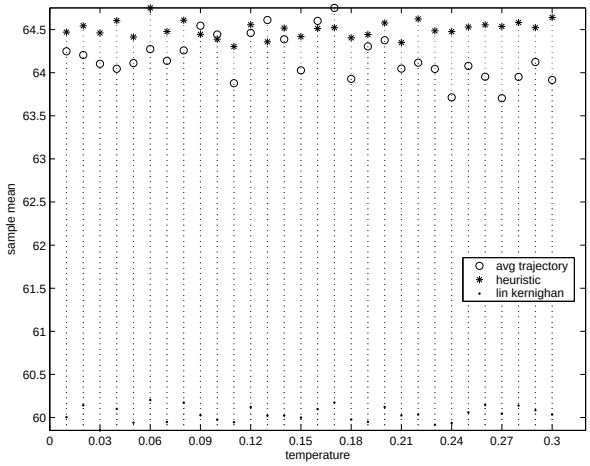


n=200, $\sigma^2 = 0.005$

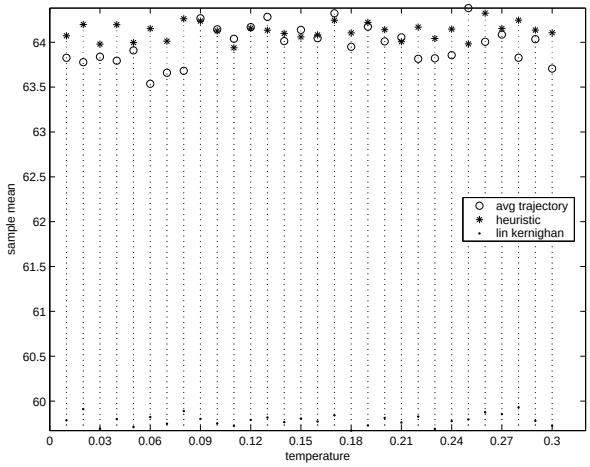


n=200, $\sigma^2 = 0.0025$

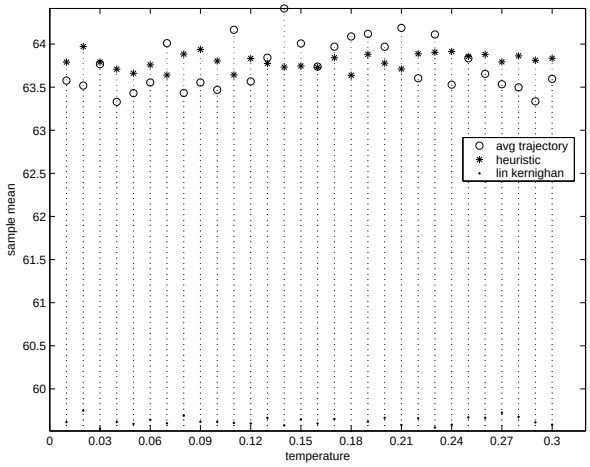
Figure 4.14: Computational results for data set three for $n = 200, \sigma^2 = 0.01, 0.005,$ and 0.0025, and $T = 0.01$ to 0.30.



n=300, $\sigma^2 = 0.01$



n=300, $\sigma^2 = 0.005$



n=300, $\sigma^2 = 0.0025$

Figure 4.15: Computational results for data set three for $n = 300, \sigma^2 = 0.01, 0.005,$ and 0.0025, and $T = 0.01$ to 0.30.

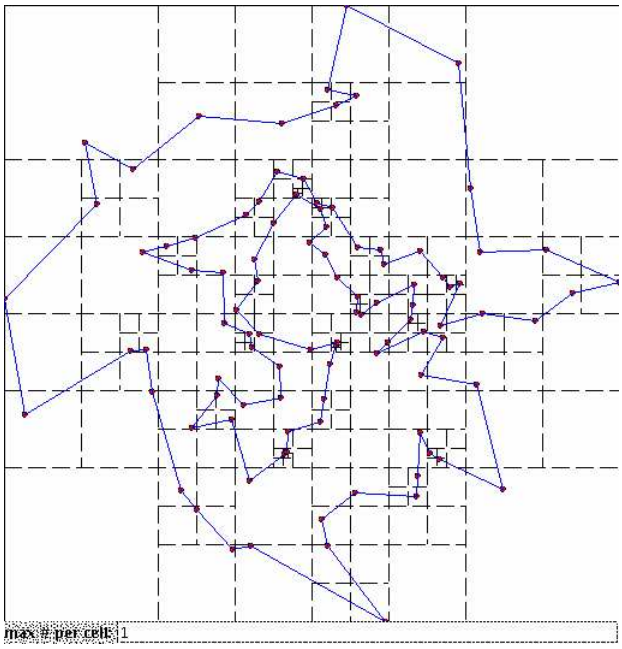
4.4.1 Quad tree approach

A quad tree is the extension of a binary tree in two-dimensional space (see Finkel and Bentley [14] for details). Given a rectangle TSP instance, we find a rectangle enclosing it and subdivide the rectangle into four regions called quadrants with the same geometry. We then generate a quad tree as follows. Select a quadrant and check the number of nodes that it contains. If the number of nodes does not exceed a specified value (we denote the parameter by Max), then we stop processing this quadrant. If the number of nodes is greater than Max, then we subdivide the quadrant into four quadrants and continue. We stop when all quadrants have been processed. By selecting different values for Max, we can generate a family of quad trees. For example, if we set Max = 1, then each quadrant contains at most one node. If we set Max = number of nodes in the TSP instance, then the quad tree has just one region.

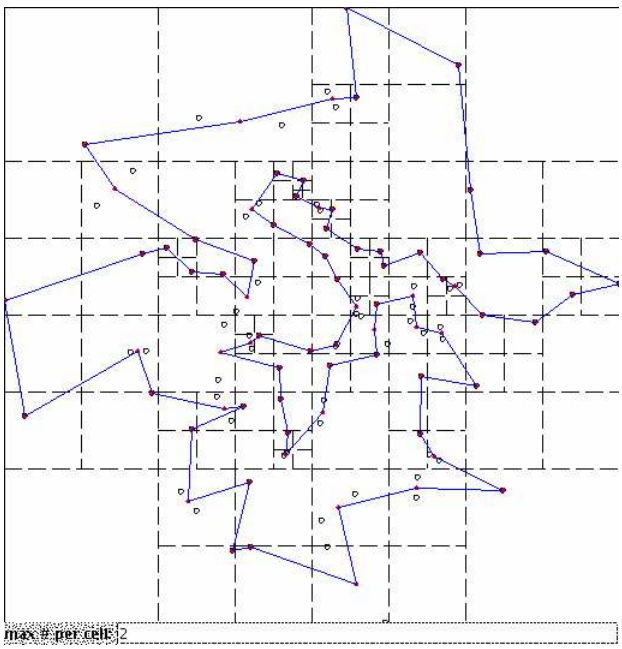
After the quad tree is generated, we compute the centroid of each quadrant. We link all of the centroids using CHCI to form the average trajectory. We use the finite-horizon adaption technique to generate a tour for a new instance. In Figure 4.16 to Figure 4.19, we show the average trajectory generated by the quad tree approach for a 100-node problem with values of Max from 1 to 8. We see that the parameter Max in the quad tree approach behaves much like the temperature parameter T in the average trajectory approach (compare Figure 4.4 and Figure 4.16 to Figure 4.19).

4.4.2 Computational experiments with the quad tree approach

We selected a 100-node problem from data set three and conducted the following experiments with the average trajectory approach and the quad tree

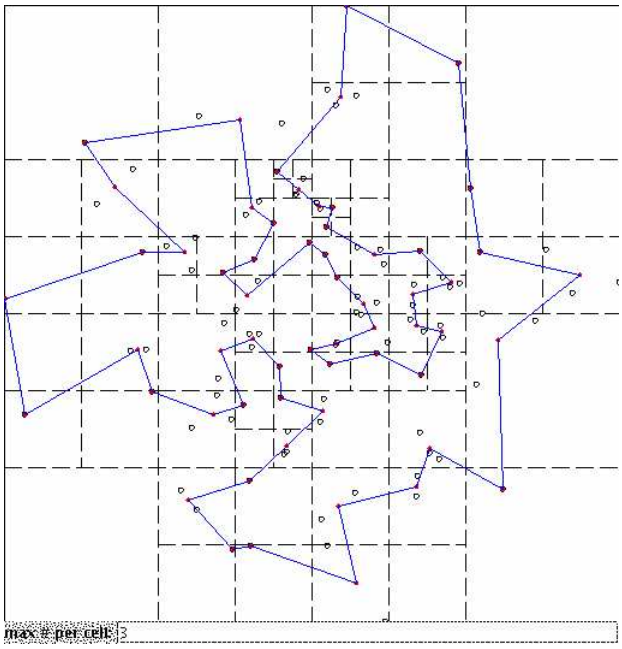


Max = 1

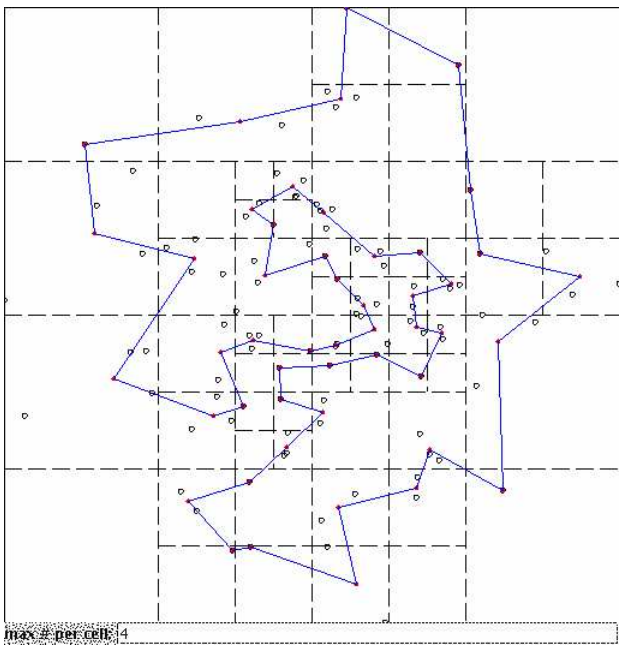


Max = 2

Figure 4.16: Average trajectories generated by quad tree approach with $n = 100$ and Max $= 1, 2$.

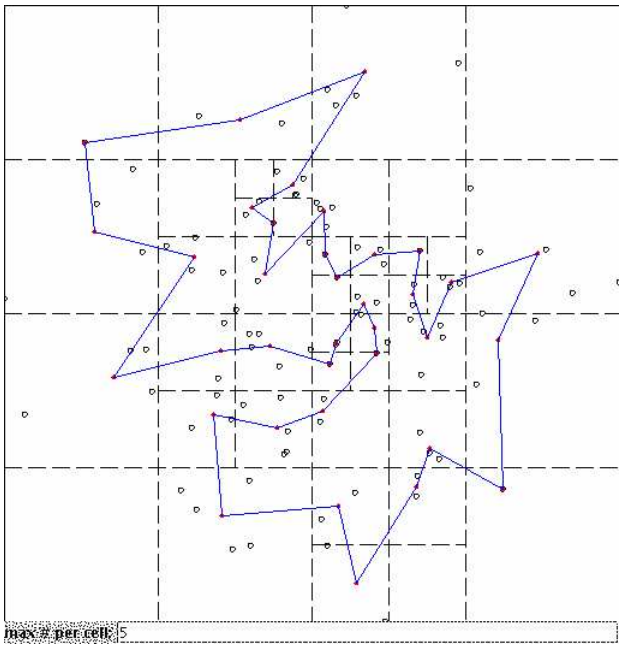


Max = 3

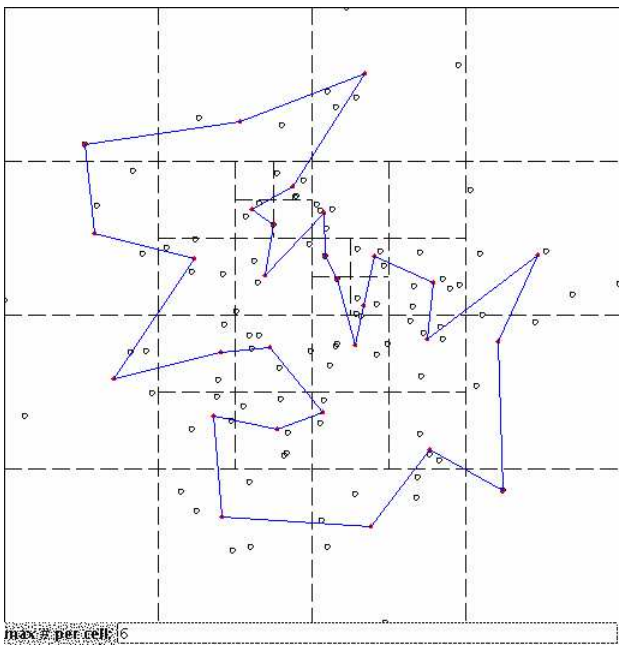


Max = 4

Figure 4.17: Average trajectories generated by quad tree approach with $n = 100$ and Max $= 3, 4$.

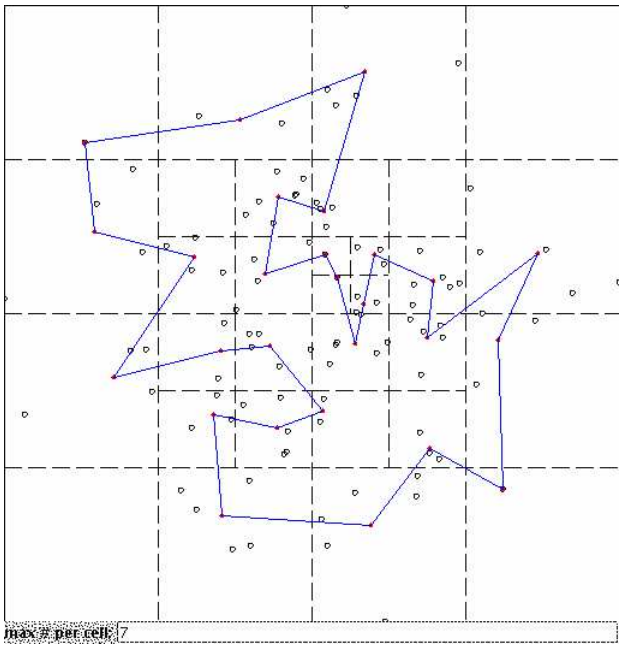


Max = 5

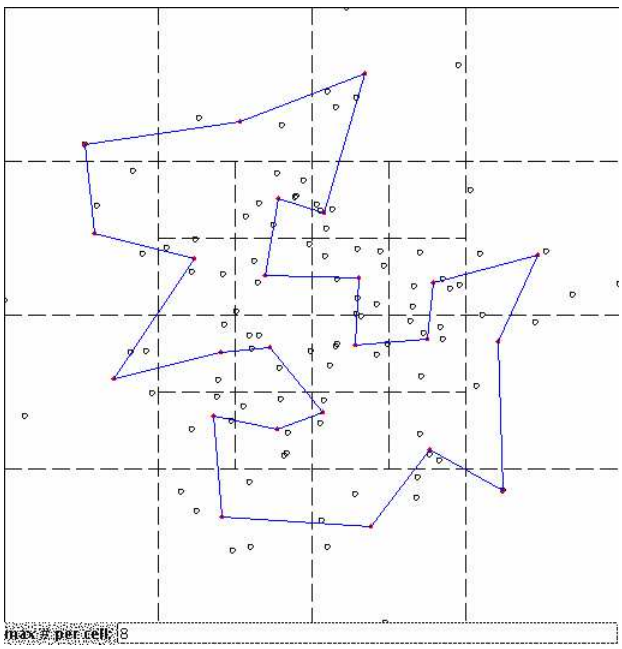


Max = 6

Figure 4.18: Average trajectories generated by quad tree approach with $n = 100$ and Max $= 5, 6$.



Max = 7



Max = 8

Figure 4.19: Average trajectories generated by quad tree approach with $n = 100$ and Max $= 7, 8$.

approach. We used three variances ($\sigma^2 = 0.01, 0.005, 0.0025$), 30 temperatures for the average trajectory approach ($T = 0.01$ to 0.30), and four values of Max for the quad tree approach (Max = 2, 4, 6, 8). We sampled 100 instances (at each temperature) and generated a tour for each instance by both methods.

In Figure 4.20 to Figure 4.31, we present our computational results. In each figure, the top figure gives the average tour length over 100 instances, while the bottom figure gives the percentage that the quad tree approach generates a lower-cost tour than the average trajectory approach over the 100 instances. In examining the figures, it is clear that, as Max increases in value (especially when Max = 6 and 8), the quad tree approach gives much better results.

We point out that we have essentially conducted a large simulation experiment in which we varied problem settings (variance, temperature, Max) and randomly sampled 100 instances at each temperature to generate solutions. We calculated the average tour length over the 100 instances for the quad tree approach (these are the circles in Figure 4.20 to Figure 4.31) and for the average trajectory approach (these are the asterisks) at each temperature. Although a visual inspection of the results in Figure 4.20 to Figure 4.31 clearly demonstrates that the quad tree approach outperforms the average trajectory approach as the value of Max increases, we could formally examine the performance by conducting an hypothesis test of means at each temperature (H_0: Mean of quad tree solutions at temperature T equals Mean of average trajectory solutions at temperature T, H_a: Mean of quad tree solutions at temperature T is less than Mean of average trajectory solutions at temperature T). Kelton and Law [26] provide the details for conducting hypothesis tests for the mean in a simulation study.

We point out that the average time to construct the average trajectory using

MCMC in Braun and Buhmann's approach is about 60 ms (milliseconds) on an Athlon 1 GHz computer. The quad tree approach takes an average of about 160 ms when Max = 1 and about 20 ms when Max = 8 to construct the average trajectory. CHCI does not construct an average trajectory.

To generate the final tour for a new TSP instance, Braun and Buhmann's approach takes about 12 ms on average, while the quad tree approach takes about 10 ms and CHCI takes about 16 ms. Clearly, all three procedures are very quick.

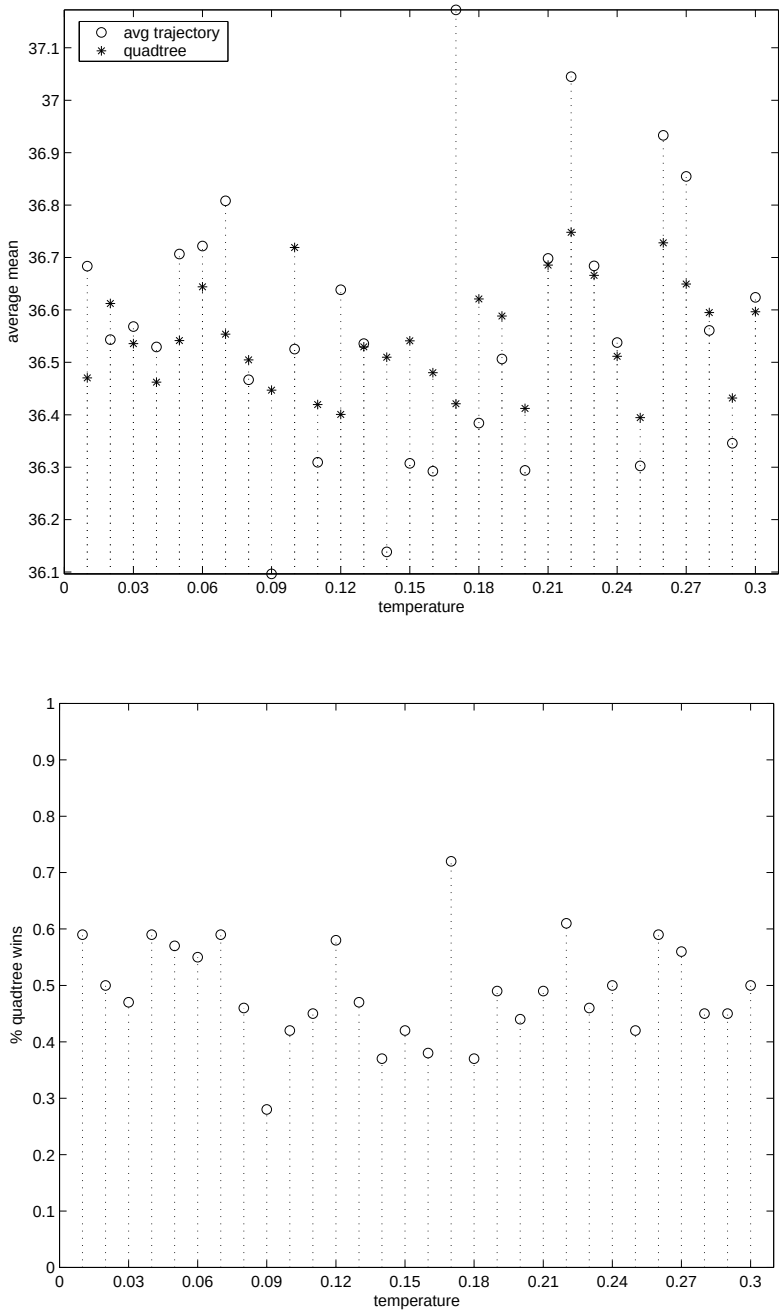
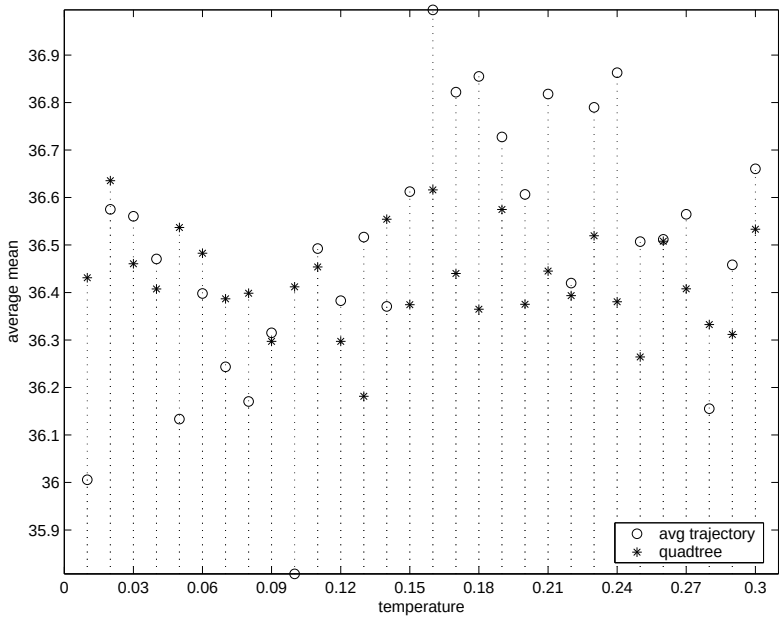


Figure 4.20: Computational results for data set three for $n = 100$, $\sigma^2 = 0.01$, $T = 0.01$ to 0.30, and Max = 2.



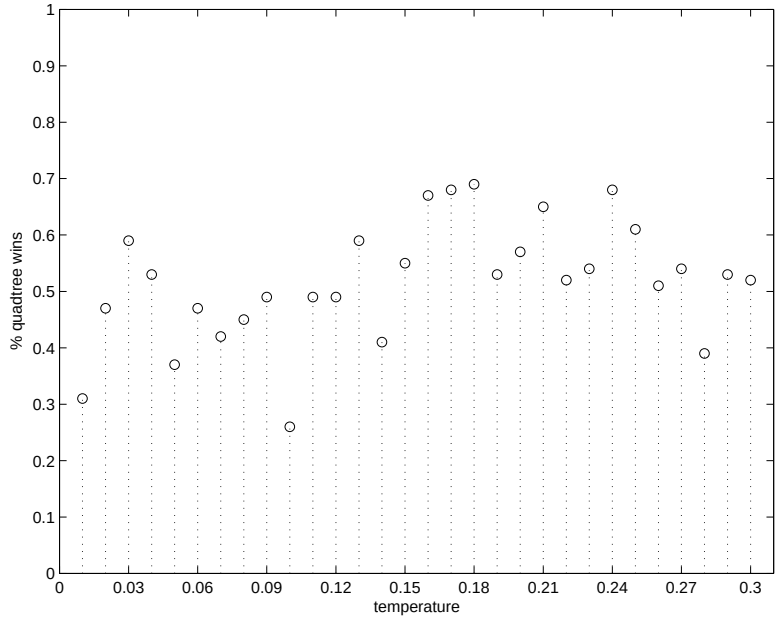
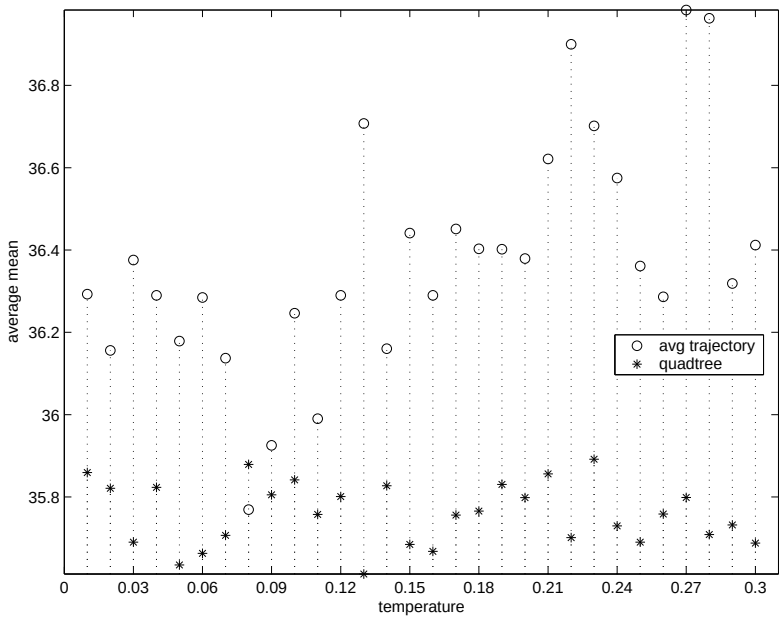


Figure 4.21: Computational results for data set three for $n = 100$, $\sigma^2 = 0.01$, $T = 0.01$ to 0.30, and Max = 4.



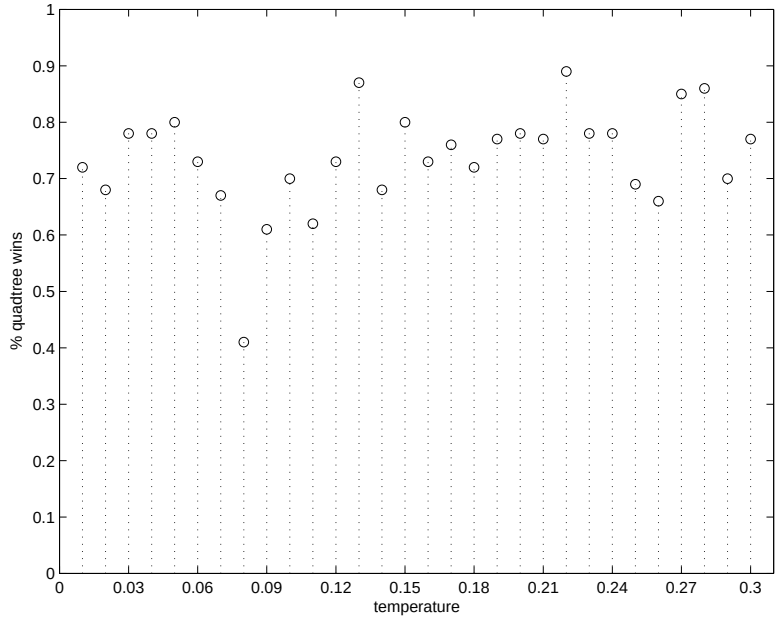


Figure 4.22: Computational results for data set three for $n = 100$, $\sigma^2 = 0.01$, $T = 0.01$ to 0.30, and Max = 6.

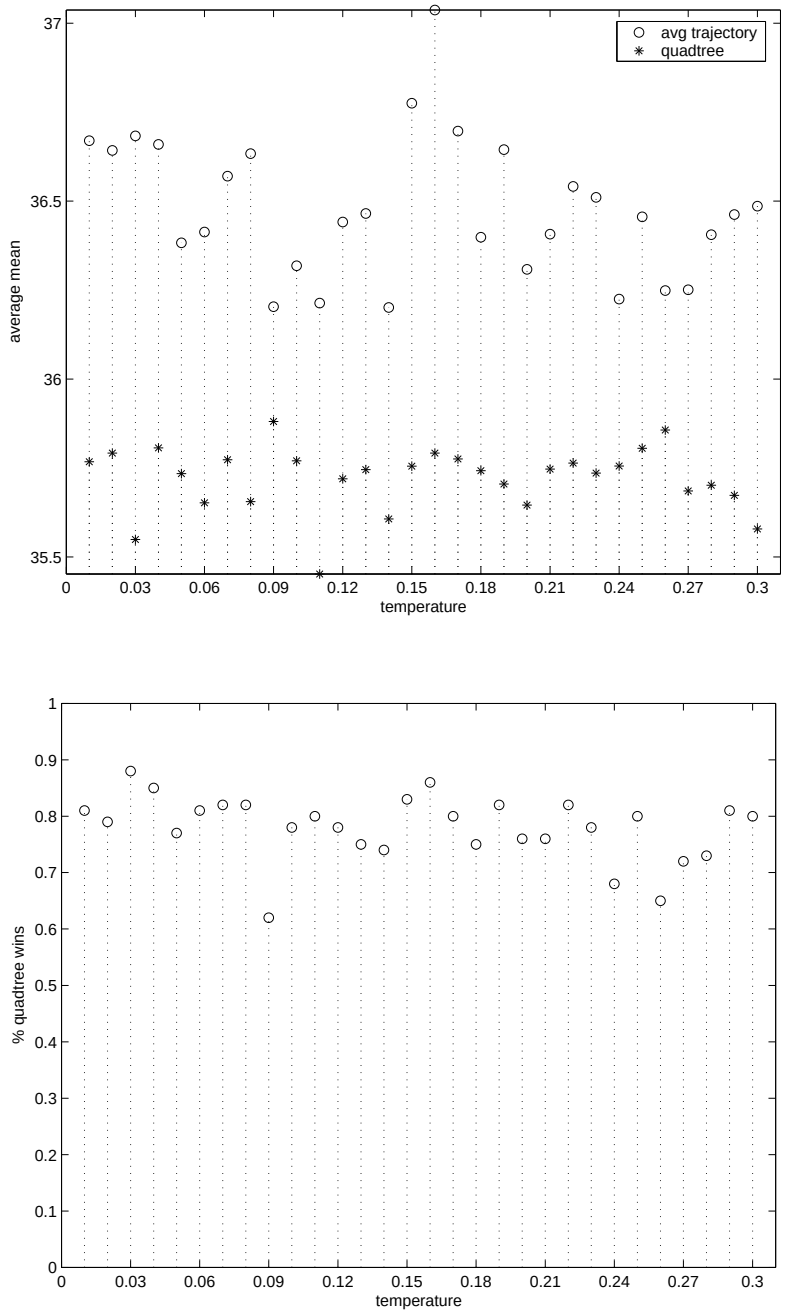


Figure 4.23: Computational results for data set three for $n = 100$, $\sigma^2 = 0.01$, $T = 0.01$ to 0.30, and Max = 8.

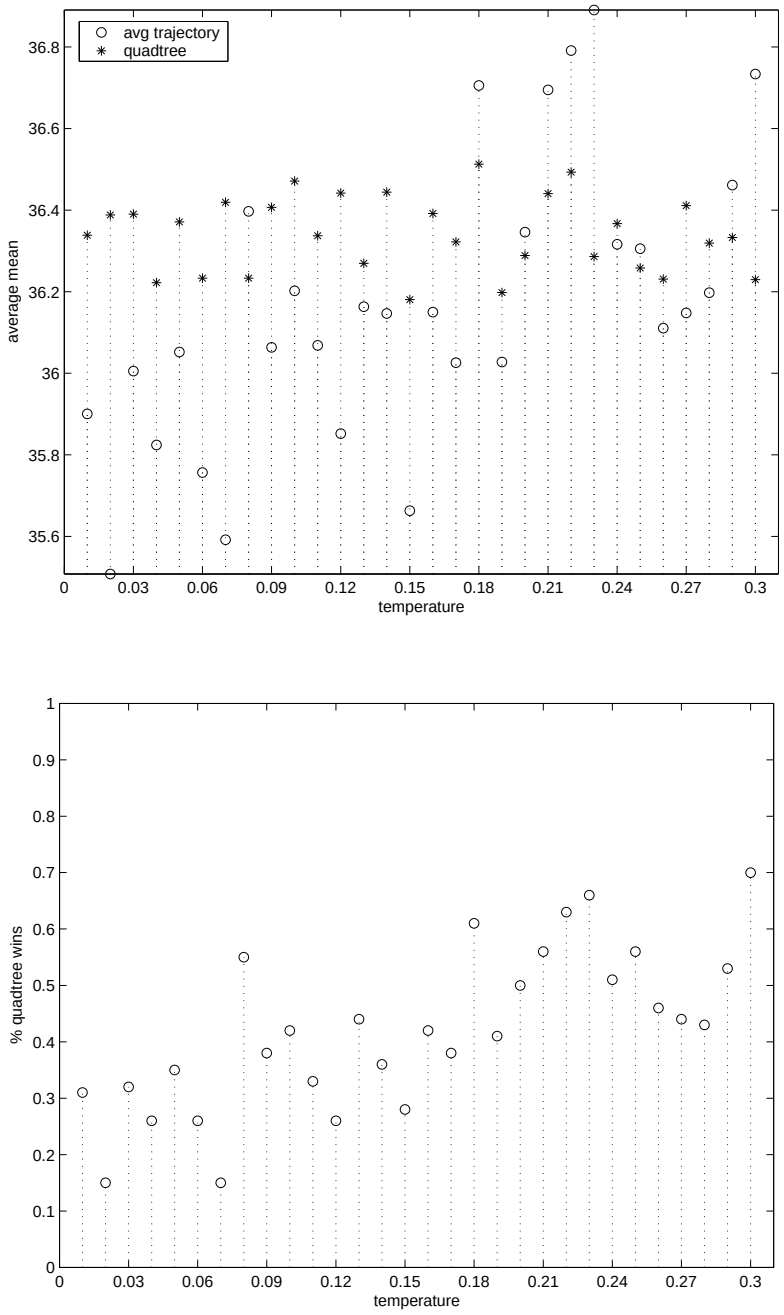


Figure 4.24: Computational results for data set three for $n = 100$, $\sigma^2 = 0.005$, $T = 0.01$ to 0.30, and Max = 2.

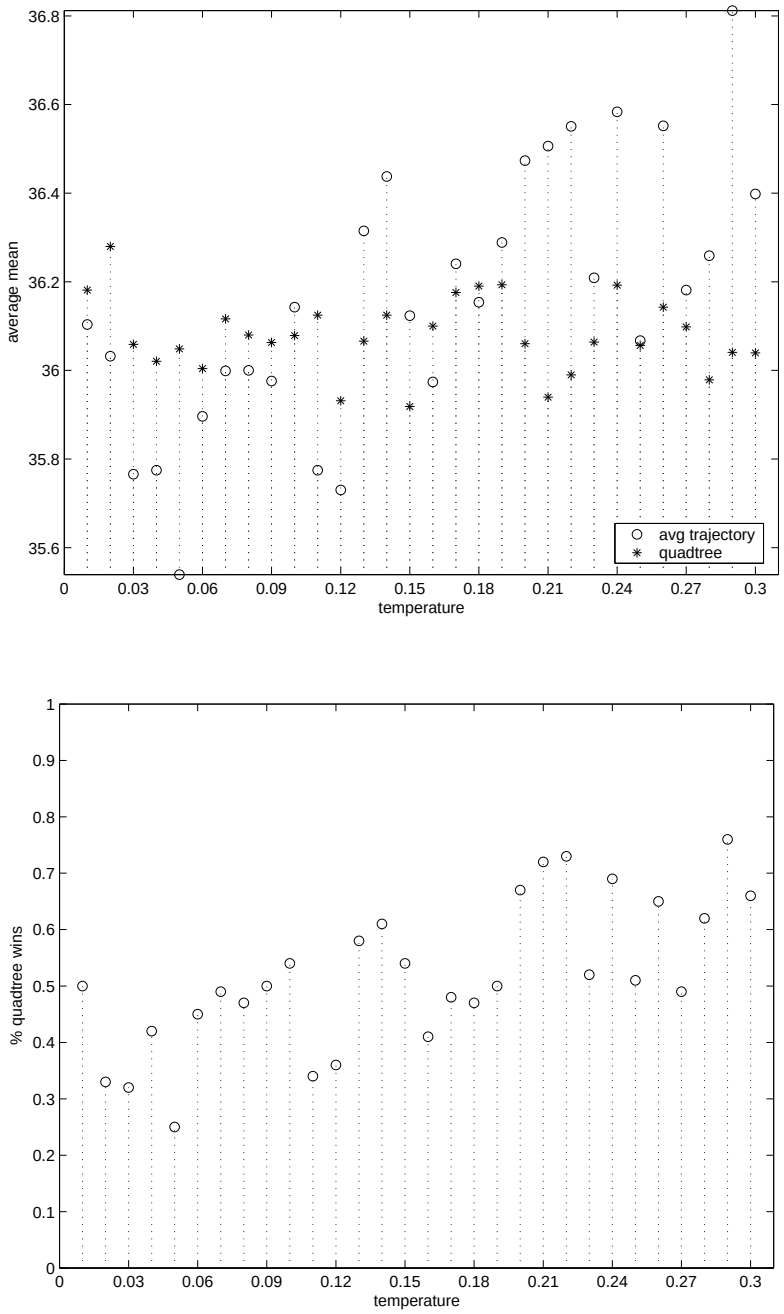


Figure 4.25: Computational results for data set three for $n = 100$, $\sigma^2 = 0.005$, $T = 0.01$ to 0.30, and Max = 4.

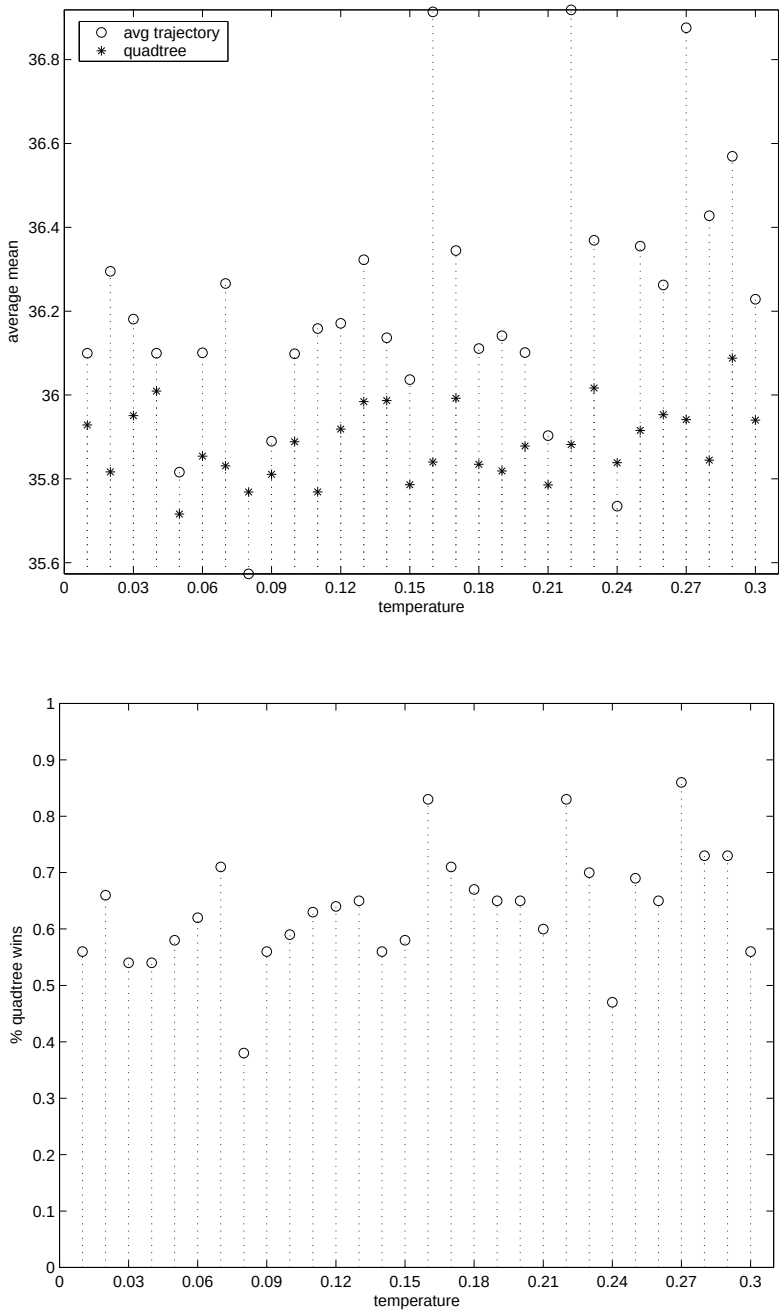


Figure 4.26: Computational results for data set three for $n = 100$, $\sigma^2 = 0.005$, $T = 0.01$ to 0.30, and Max = 6.

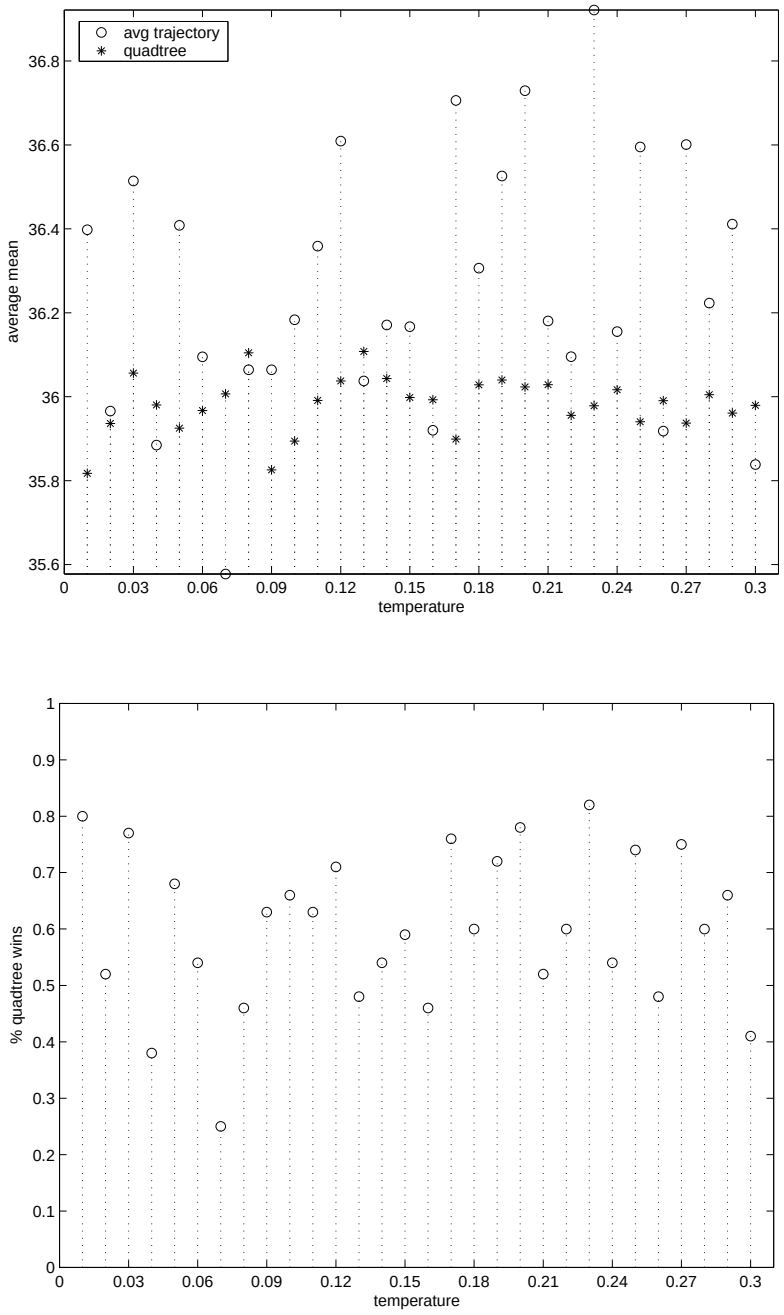


Figure 4.27: Computational results for data set three for $n = 100$, $\sigma^2 = 0.005$, $T = 0.01$ to 0.30, and Max = 8.

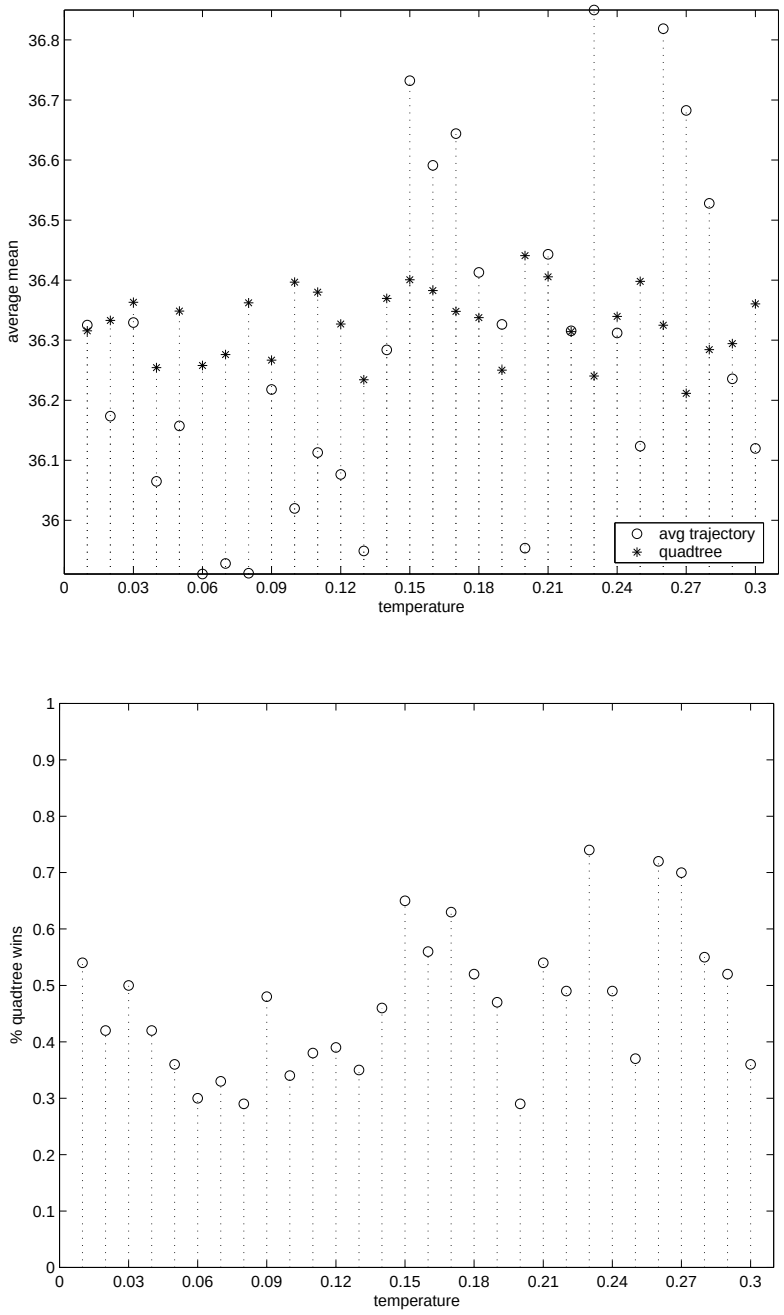
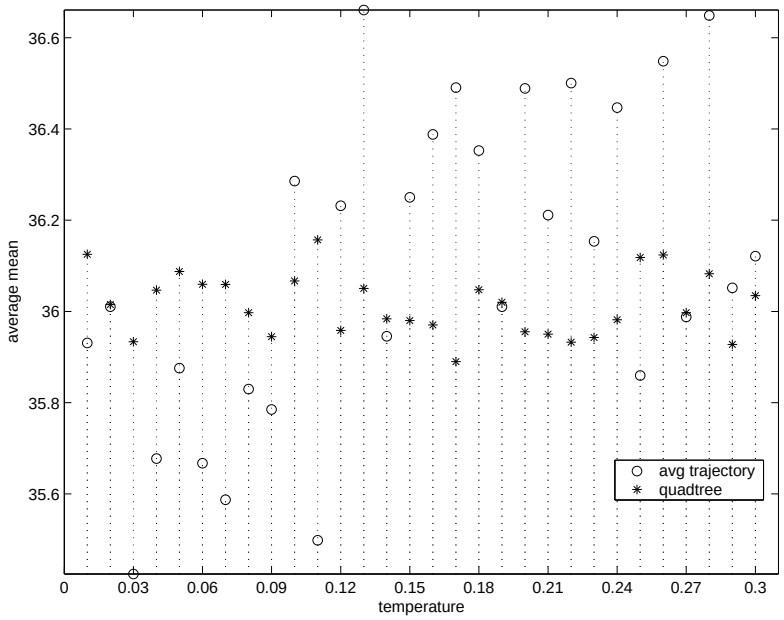


Figure 4.28: Computational results for data set three for $n = 100$, $\sigma^2 = 0.0025$, $T = 0.01$ to 0.30, and Max = 2.



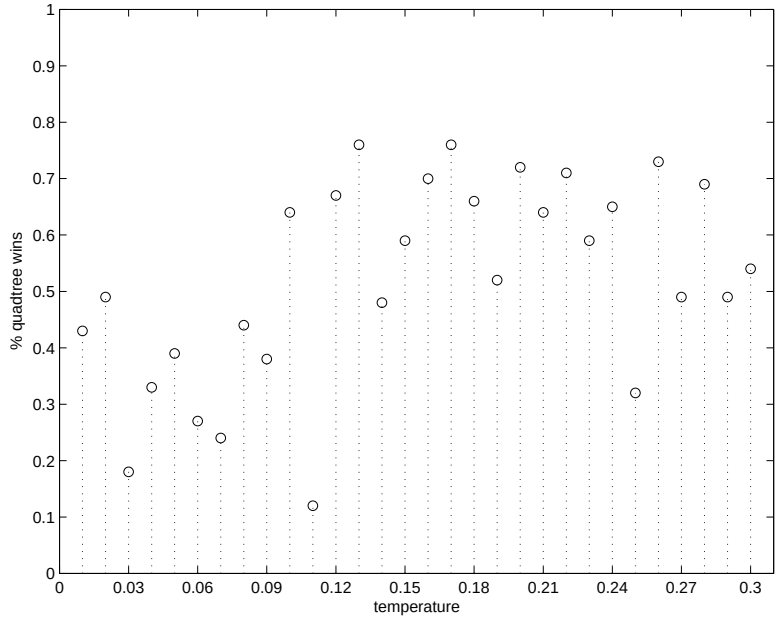


Figure 4.29: Computational results for data set three for $n = 100$, $\sigma^2 = 0.0025$, $T = 0.01$ to 0.30, and Max = 4.

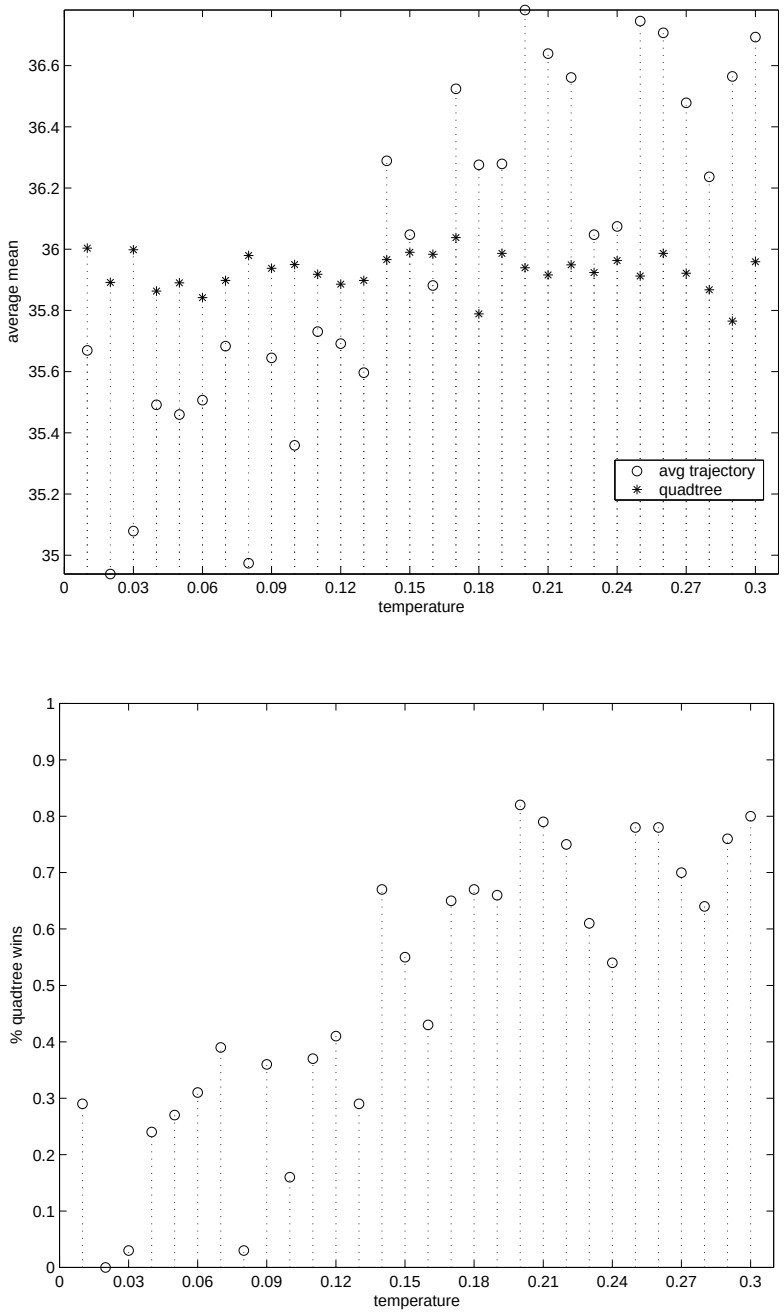


Figure 4.30: Computational results for data set three for $n = 100$, $\sigma^2 = 0.0025$, $T = 0.01$ to 0.30, and Max = 6.

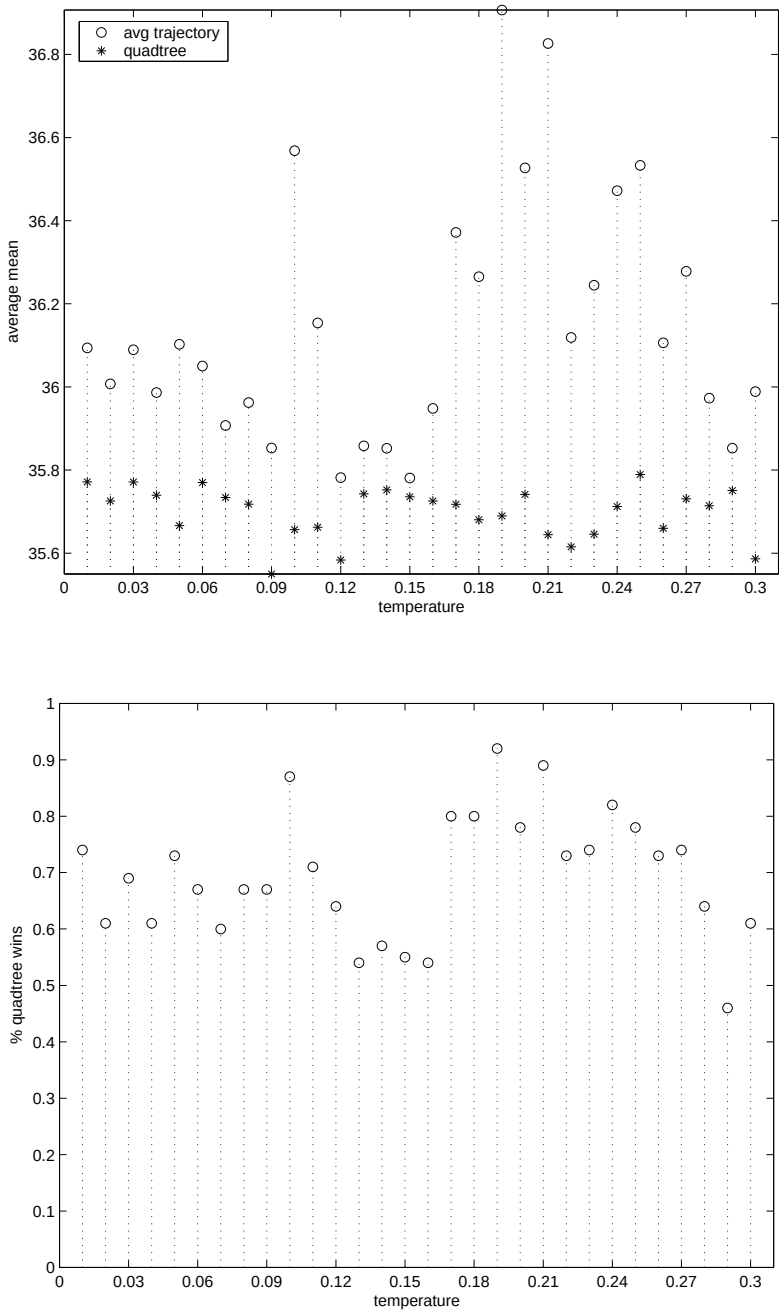


Figure 4.31: Computational results for data set three for $n = 100$, $\sigma^2 = 0.0025$, $T = 0.01$ to 0.30, and Max = 8.

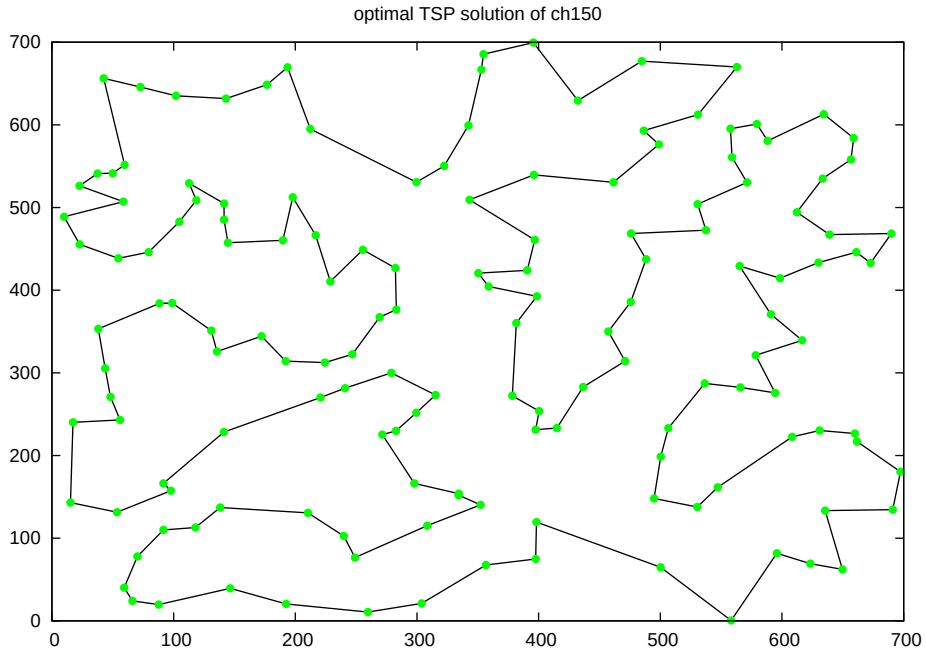


Figure 4.32: Optimal solution of ch150.

4.5 Applying the quad tree approach to the PTSP

The probabilistic traveling salesman problem (PTSP) is a variant of the TSP where only a subset of the nodes needs to be visited (Jaillet [22]). Here we consider the following variant of the PTSP. For each node i, the probability that i needs to be visited is p (a known parameter). The objective function is to find an a priori tour visiting all the nodes with minimum average tour length. In this section, we use a 150-node problem taken from TSPLIB [33] for our computational results. In Figure 4.32, we show the optimal solution of ch150.

We use seven different values of Max from 2 to 8. Each one gives us an average trajectory with fewer than 150 nodes. In Figure 4.33 to Figure 4.39, we show these seven average trajectories.

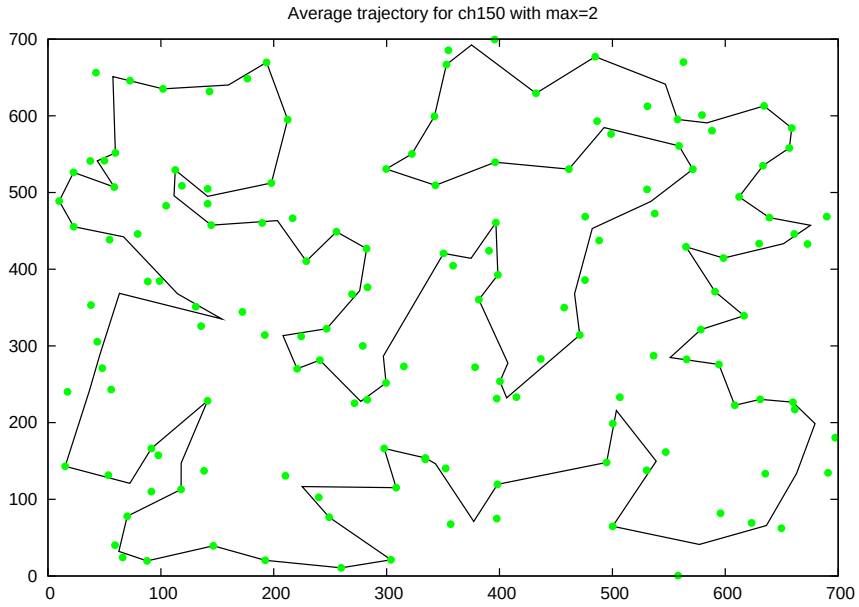


Figure 4.33: Average trajectory of ch150 with Max = 2.

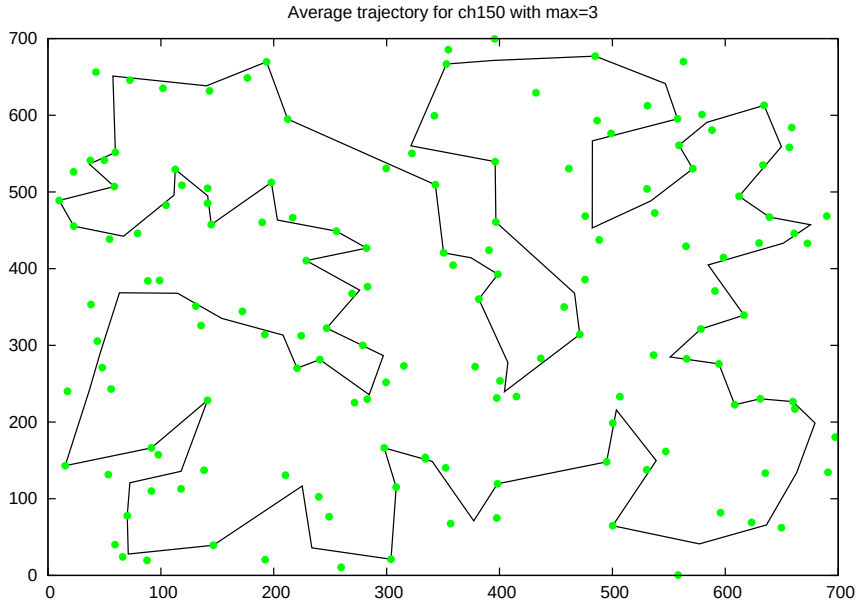


Figure 4.34: Average trajectory of ch150 with Max = 3.

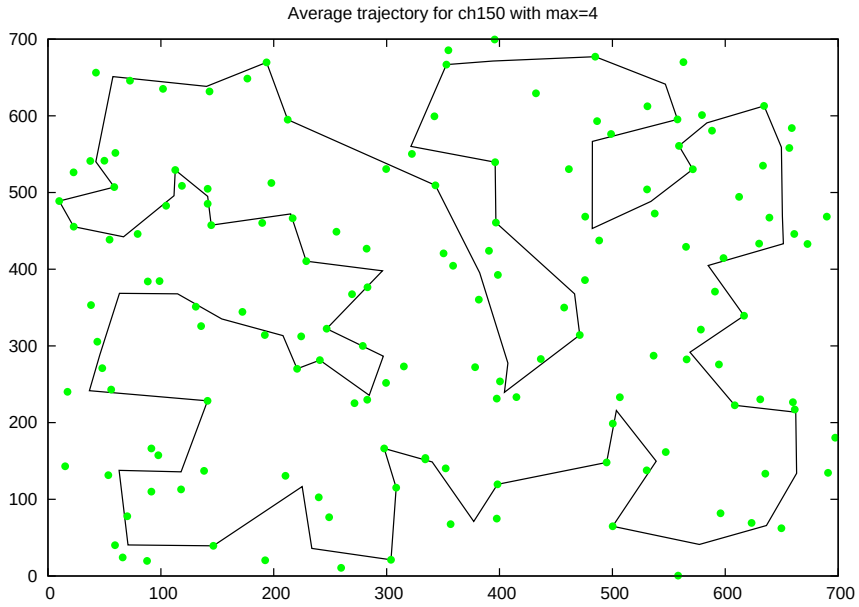


Figure 4.35: Average trajectory of ch150 with Max = 4.

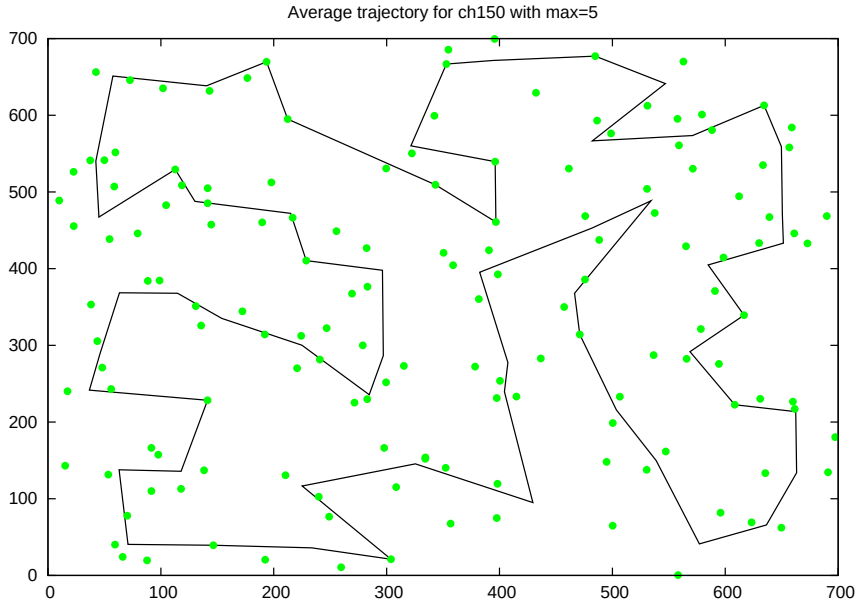


Figure 4.36: Average trajectory of ch150 with Max = 5.

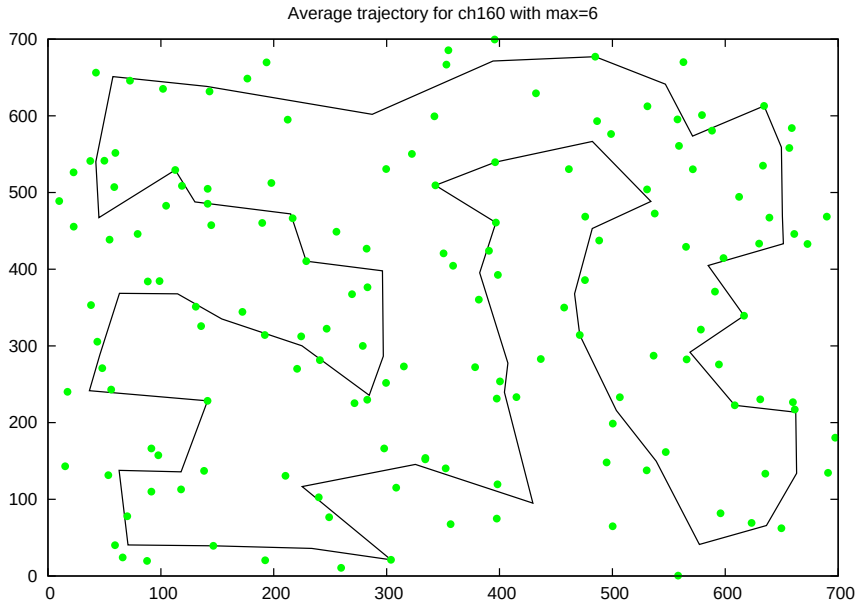


Figure 4.37: Average trajectory of ch150 with Max = 6.

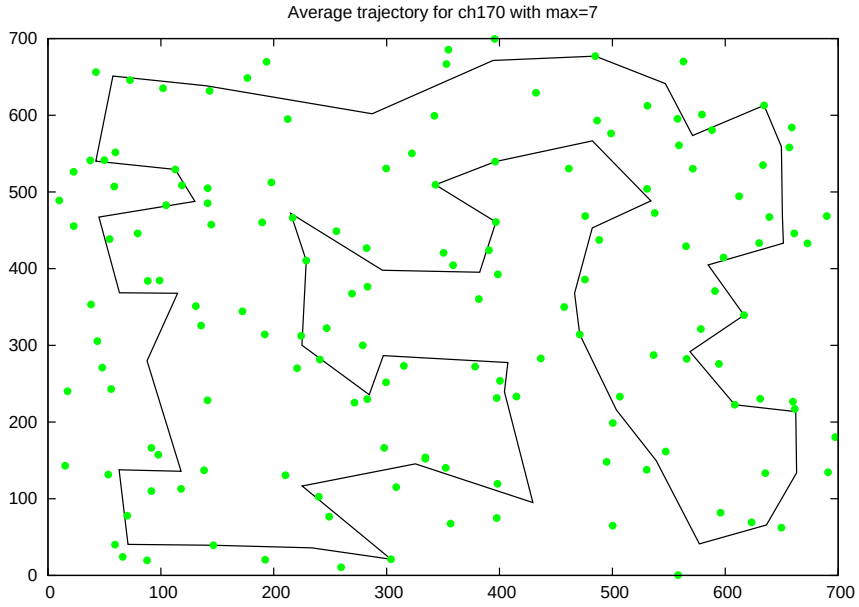


Figure 4.38: Average trajectory of ch150 with Max = 7.

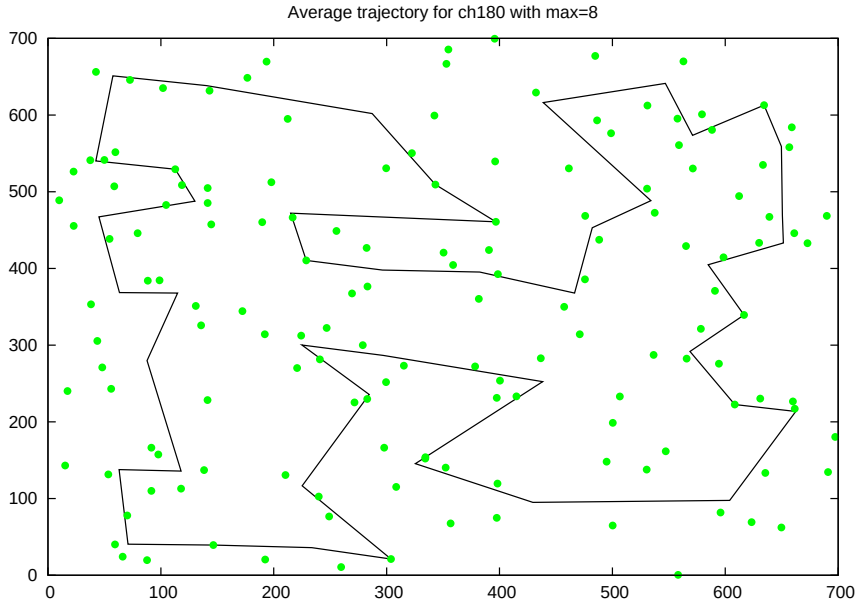


Figure 4.39: Average trajectory of ch150 with Max = 8.

After we have the average trajectory, we re-sequence the original TSP nodes by a linear interpolation over the average trajectory (this is what we did in the NTSP). Specifically, for each TSP node, we find its t value (the passing time) on the average trajectory and we sequence the nodes by the value of the passing time. In Figure 4.40 to Figure 4.46, we show the PTSP solutions for different values of Max.

The PTSP objective function for probability p is given by

$$E[L_\lambda] = p^2 \sum_{r=0}^{n-2} (1-p)^r L_\lambda^{(r)} \tag{4.1}$$

where $L_\lambda^{(r)} = \sum_{j=0}^{n-1} d(j, (j+1+r) \mod n)$. The $L_\lambda^{(r)}$'s have the combinatorial interpretation of being the lengths of a collection of $\gcd(n, r+1)$ sub-tours. In Figure 4.47, we show an example of $n = 6$ and $r = 1, 2$. To make it easy to understand, Equation 4.2 shows the case of $r = 1$. The sum in the first

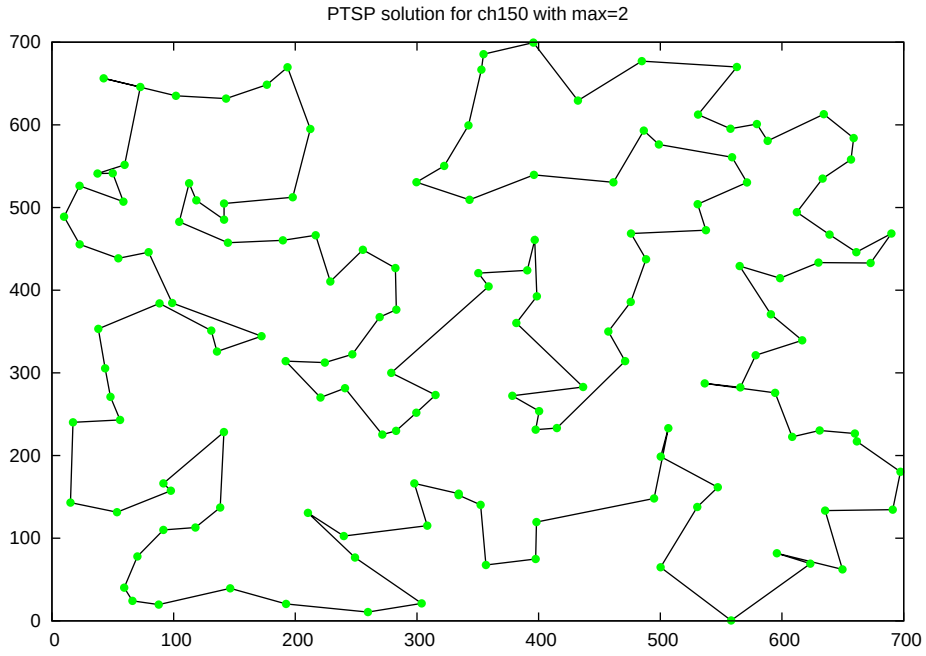


Figure 4.40: PTSP solution for Max = 2.

parenthesis corresponds to the perimeter of one triangle in Figure 4.47. So is the sum in the second parenthesis in Equation 4.2 below:

$$\begin{aligned} L_\lambda^r &= d(0,2) + d(1,3) + d(2,4) + d(3,5) + d(4,0) + d(5,1) \\ &= (d(0,2) + d(2,4) + d(4,0)) + (d(1,3) + d(3,5) + d(5,1)). \end{aligned} \tag{4.2}$$

In Table 4.4, we compare the tour length of the average trajectory from our quad tree approach against the optimal solution for different values of p. We see that for values of $p \leq 0.5$, the tour length of average trajectory is comparable to that of the optimal PTSP solution.

Table 4.4: Objective function values for different values of p and Max

p	opt	Max							
		1	2	3	4	5	6	7	8
0.1	2728.322	2665.063	2669.730	2676.650	2676.321	2664.679	2803.882	2798.066	2610.850
0.2	3792.643	3631.285	3609.336	3685.622	3681.123	3655.387	3751.384	3721.818	3626.185
0.3	4442.263	4285.040	4287.476	4370.677	4361.745	4348.004	4363.376	4381.635	4377.110
0.4	4901.584	4800.461	4829.571	4904.824	4893.280	4916.285	4861.816	4936.932	5009.485
0.5	5264.174	5234.607	5284.580	5354.120	5342.908	5417.820	5306.205	5431.347	5575.599
0.6	5571.036	5612.449	5682.203	5749.635	5741.864	5878.923	5719.228	5885.928	6100.490
0.7	5842.310	5947.640	6041.188	6108.248	6106.720	6313.834	6111.749	6312.514	6597.902
0.8	6088.917	6248.834	6373.556	6440.120	6447.126	6731.099	6489.905	6718.190	7075.958
0.9	6317.185	6521.862	6687.281	6751.699	6768.975	7136.414	6857.719	7107.606	7539.767
1.0	6530.903	6770.803	6987.875	7047.173	7075.870	7534.000	7218.192	7484.304	7992.755

4.5.1 Comparison between the traditional PTSP heuristic and the quad tree heuristic

In this section, we present limited computational results for the quad tree average trajectory heuristic. We solve the PTSP by interpolating new PTSP instances over the average trajectory, and then compare these results to the traditional PTSP heuristic based on the optimal TSP solution.

For the traditional PTSP heuristic, we first solve the TSP optimally. For each new instance, we simply link the active nodes in the order in which they appear in the optimal TSP solution. No post-optimization is done.

For the quad tree average trajectory approach, we first compute the average trajectory for a given value of Max. Then, for each new instance, we do a linear interpolation over the average trajectory to generate the final solution of the PTSP. No post-optimization is done.

In Figure 4.48 to Figure 4.59, we show solutions generated by these two approaches.

We also sample 100 instances for each value of p and compare the value of the average PTSP solution. In Table 4.5, we show the computational results. Max

goes from 2 to 8 on the columns, and p goes from 0.1 to 0.8 on the rows. For each value of Max, the left column is the result from the optimal TSP solution and the middle column is the result from the quad tree approach (both averaged over 100 instances). The right column is the percentage the quad tree solution beats the optimal TSP solution over the 100 PTSP instances. We observe that, for $p \geq 0.5$, the traditional PTSP heuristic dominates the quad tree approach. For $p < 0.5$, we see that the two approaches are similar in performance. The cases in which the quad tree solution outperforms the traditional PTSP heuristic are shown in bold.

	2			3			4			5		
0.1	**2703.67**	**2667.25**	**53%**	**2714.06**	**2667.35**	**55%**	**2734.23**	**2677.82**	**56%**	**2695.37**	**2655.42**	**54%**
0.2	**3789.24**	**3618.17**	**70%**	**3817.97**	**3698.03**	**63%**	**3794.89**	**3687.66**	**63%**	**3790.63**	**3645.04**	**66%**
0.3	**4447.26**	**4293.07**	**70%**	**4439.74**	**4371.49**	**62%**	**4453.18**	**4365.68**	**60%**	**4439.51**	**4360.27**	**61%**
0.4	**4902.49**	**4826.46**	**61%**	4907.17	4918.53	45%	**4907.58**	**4893.28**	**52%**	4908.23	4927.97	47%
0.5	5260.21	5276.59	47%	5267.08	5349.30	35%	5267.82	5351.20	35%	5270.34	5417.31	29%
0.6	5573.69	5687.61	27%	5578.96	5760.59	19%	5571.77	5740.01	20%	5574.56	5881.70	10%
0.7	5842.08	6046.53	11%	5850.28	6113.60	8%	5848.10	6097.96	10%	5847.95	6301.07	2%
0.8	6090.08	6368.89	3%	6093.25	6443.14	1%	6316.41	6775.48	0%	6088.47	6731.32	0%

	6			7			8			
0.1	2716.43	2796.93	41%	2722.30	2794.66	45%	**2718.40**	**2592.85**	**62%**	
0.2	**3812.19**	**3765.65**	**57%**	**3800.32**	**3733.06**	**59%**	**3829.29**	**3618.70**	**74%**	
0.3	**4443.60**	**4363.49**	**61%**	**4438.93**	**4381.58**	**60%**	**4448.41**	**4390.48**	**58%**	
0.4	**4908.75**	**4858.77**	**58%**	4900.05	4942.32	42%	4893.91	4990.41	37%	
0.5	5266.88	5309.63	41%	5268.01	5436.57	24%	5253.32	5556.20	13%	
0.6	5568.77	5709.01	24%	5568.91	5884.28	7%	5570.02	6100.01	3%	
0.7	5846.45	6116.11	8%	5837.59	6305.33	2%	5841.05	6603.46	0%	
0.8	6092.44	6483.77	1%	6089.82	6719.73	0%	6091.94	7075.74	0%	

Table 4.5: Computational results on two approaches.

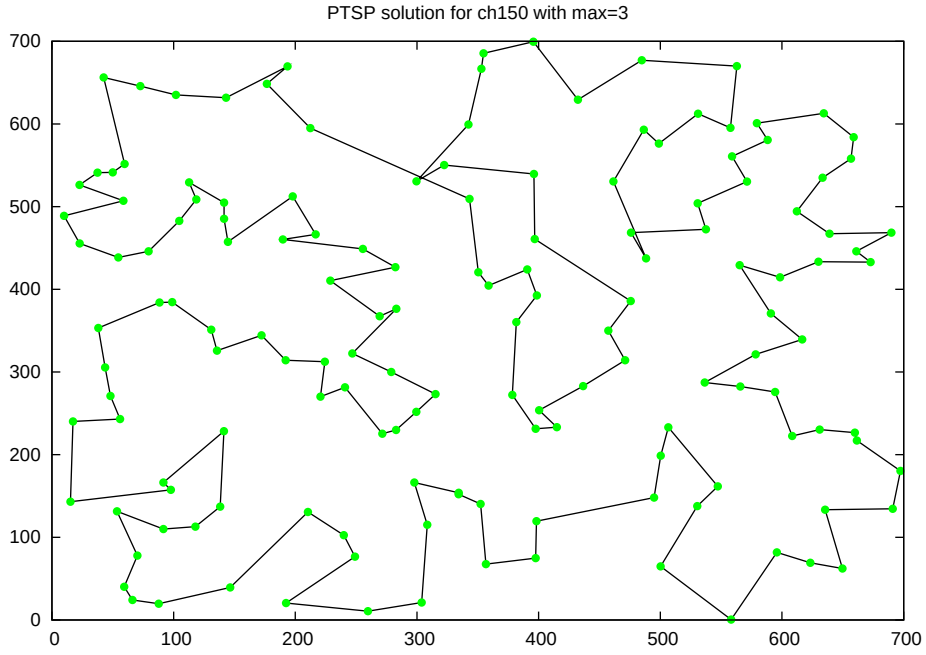


Figure 4.41: PTSP solution for Max = 3.

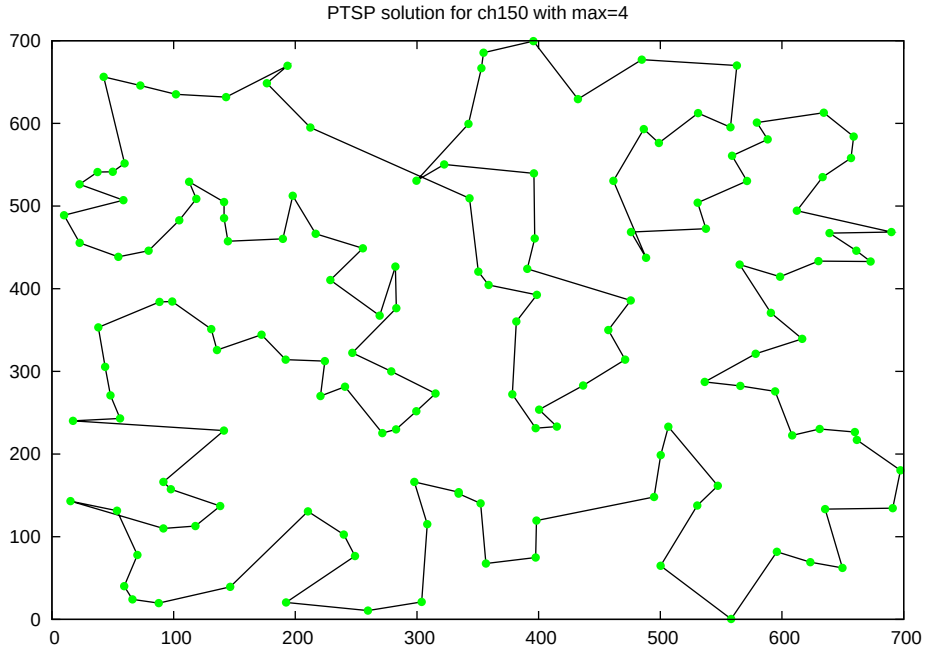


Figure 4.42: PTSP solution for Max = 4.

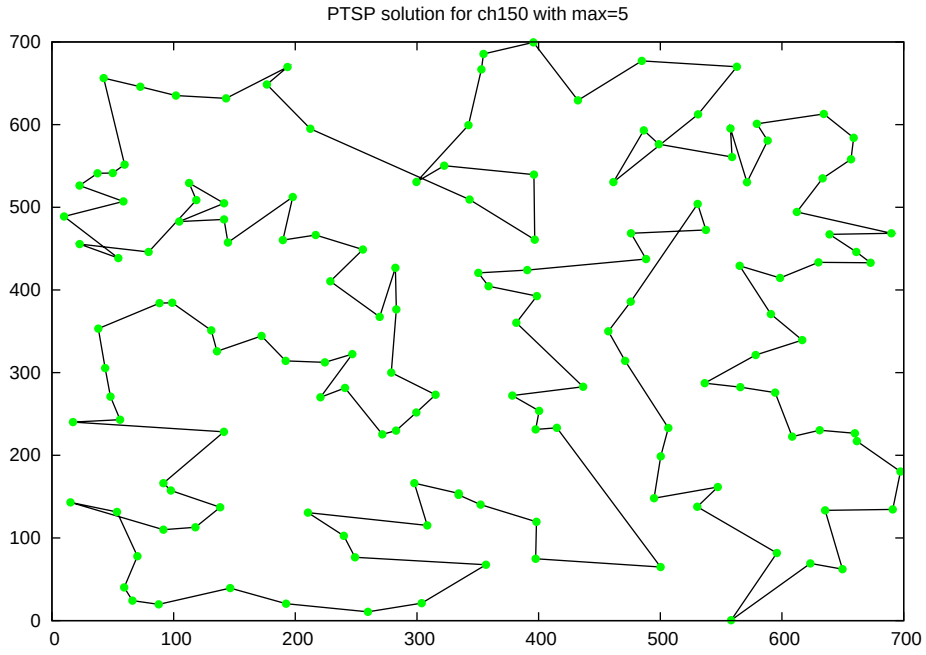


Figure 4.43: PTSP solution for Max = 5.

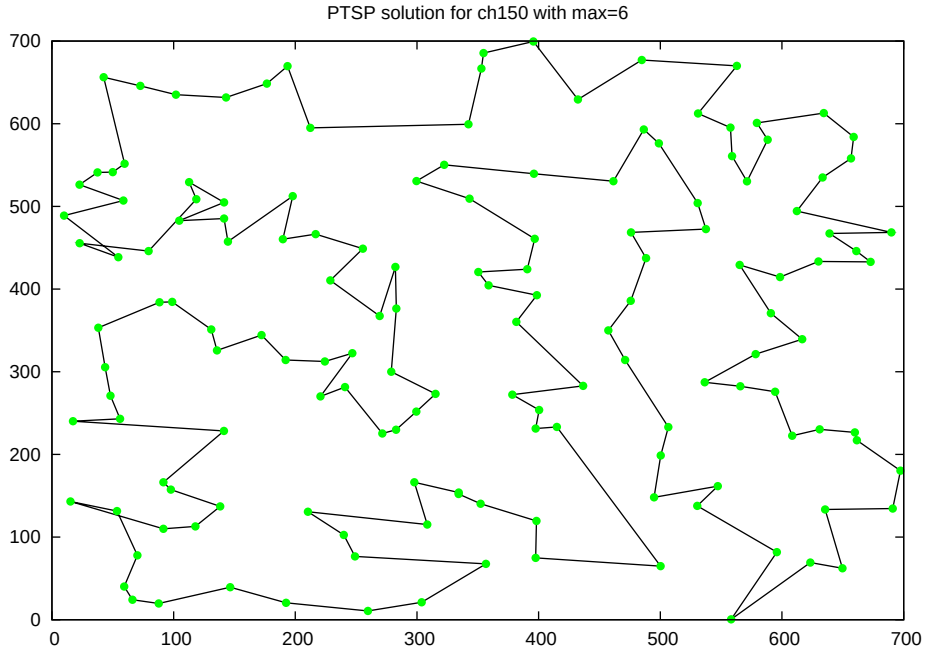


Figure 4.44: PTSP solution for Max = 6.

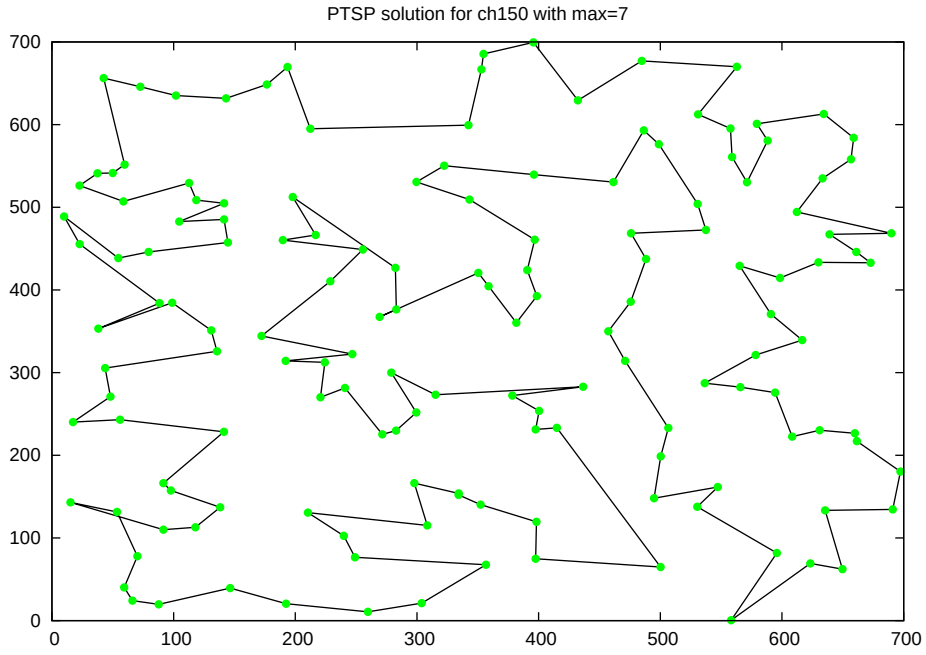


Figure 4.45: PTSP solution for Max = 7.

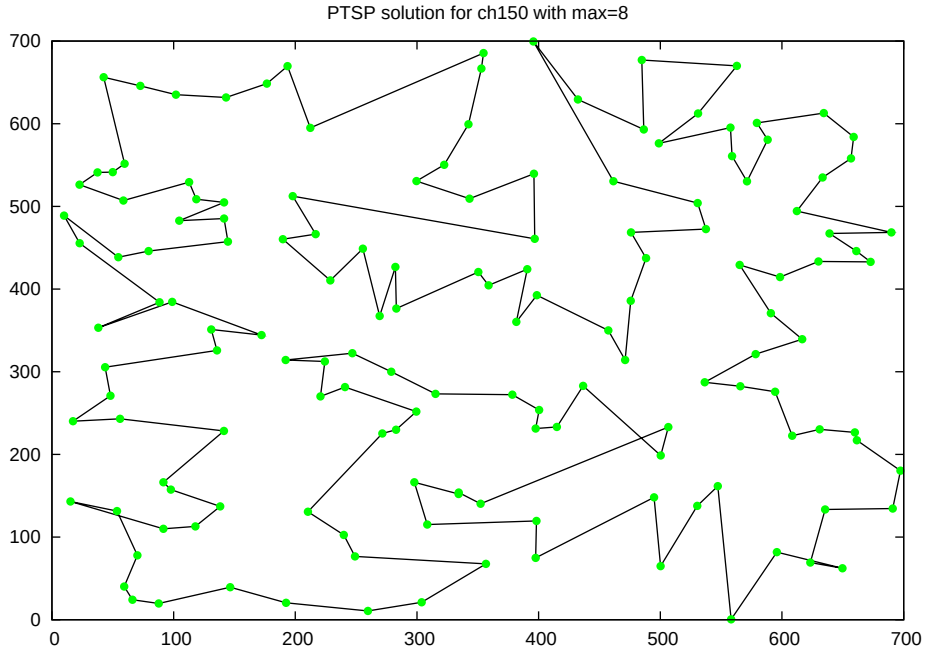


Figure 4.46: PTSP solution for Max = 8.

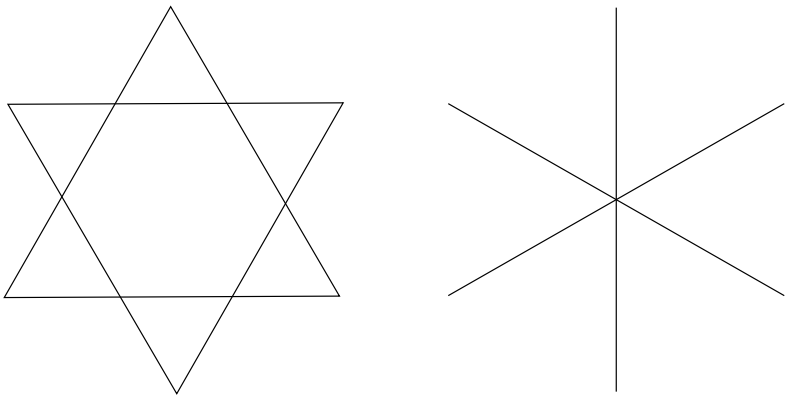


Figure 4.47: Interpretation of PTSP objective function (left figure $n = 6, r = 1$, right figure $n = 6, r = 2$).

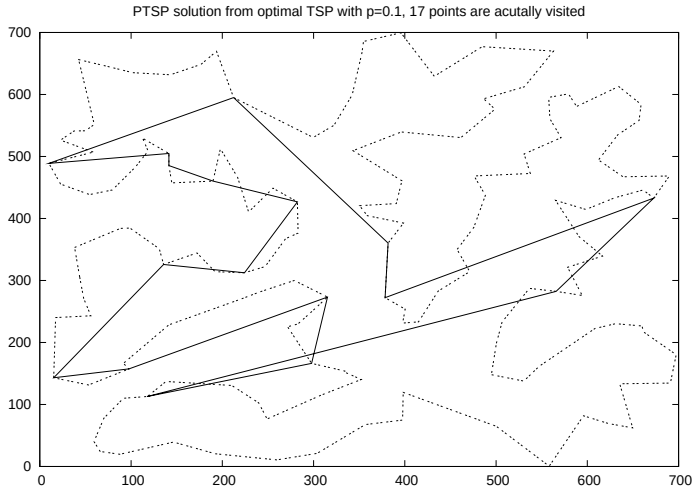


Figure 4.48: PTSP solution from optimal TSP with $p = 0.1$ (17 nodes are visited, solution value = 2645.156).

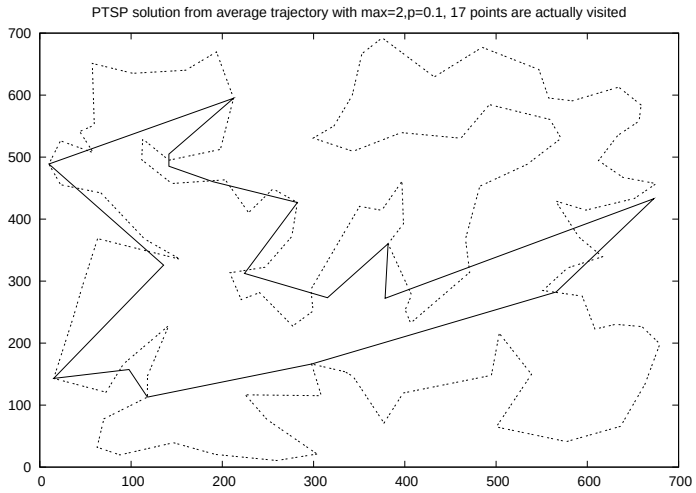


Figure 4.49: PTSP solution from quad tree with $p = 0.1, \text{Max} = 2$ (17 nodes are visited, solution value = 2497.538).

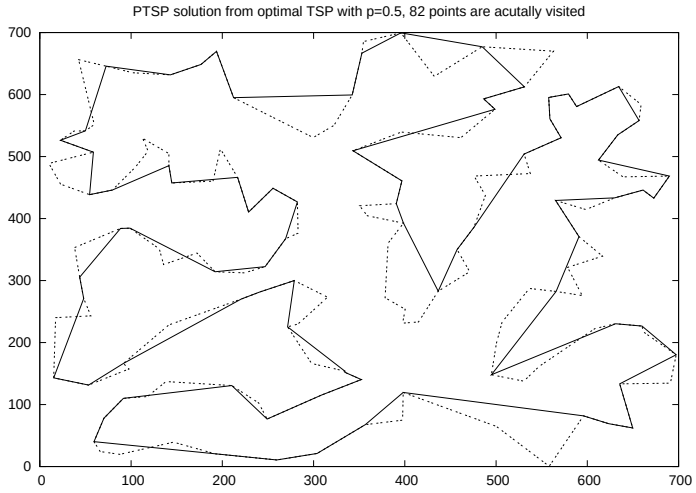


Figure 4.50: PTSP solution from optimal TSP with $p = 0.5$ (82 nodes are visited, solution value = 5477.999).

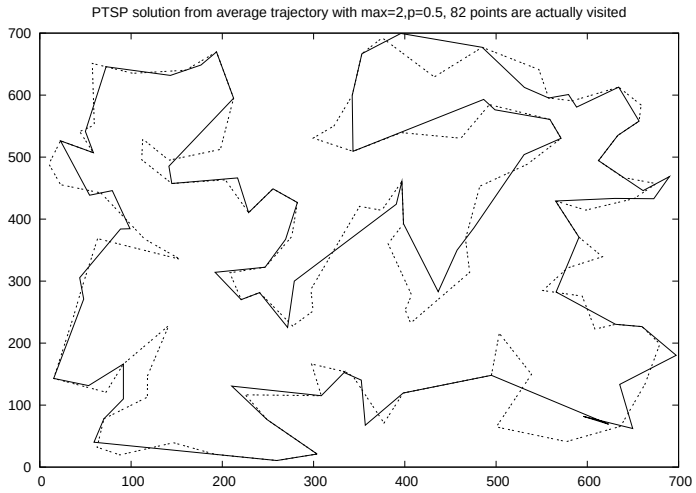


Figure 4.51: PTSP solution from quad tree with $p = 0.5, \text{Max} = 2$ (82 nodes are visited, solution value = 5208.998).

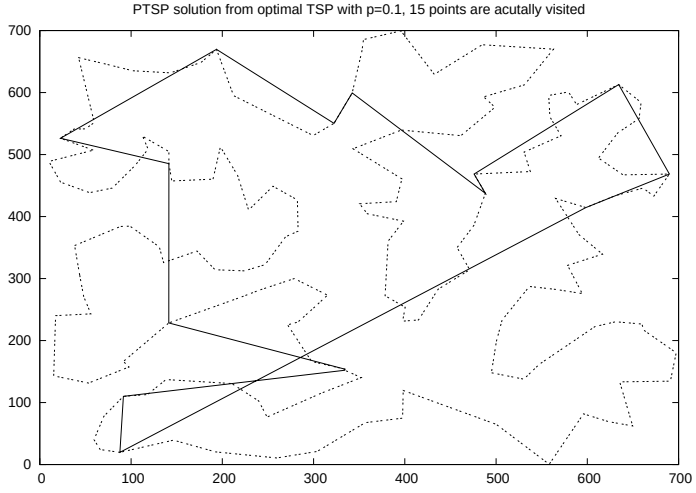


Figure 4.52: PTSP solution from optimal TSP with $p = 0.1$ (15 nodes are visited, solution value = 3550.883).

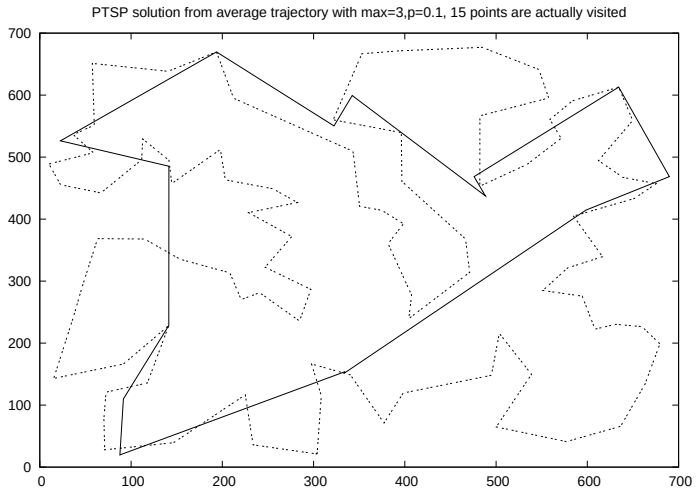


Figure 4.53: PTSP solution from quad tree with $p = 0.1, \text{Max} = 3$ (15 nodes are visited, solution value = 2434.599).

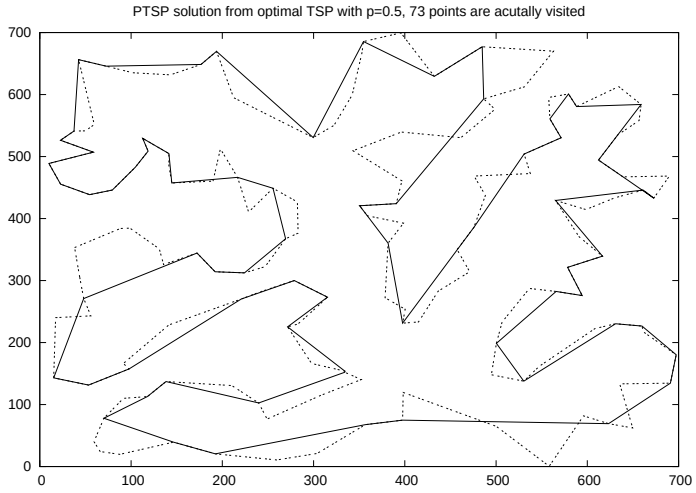


Figure 4.54: PTSP solution from optimal TSP with $p = 0.5$ (73 nodes are visited, solution value = 5047.560).

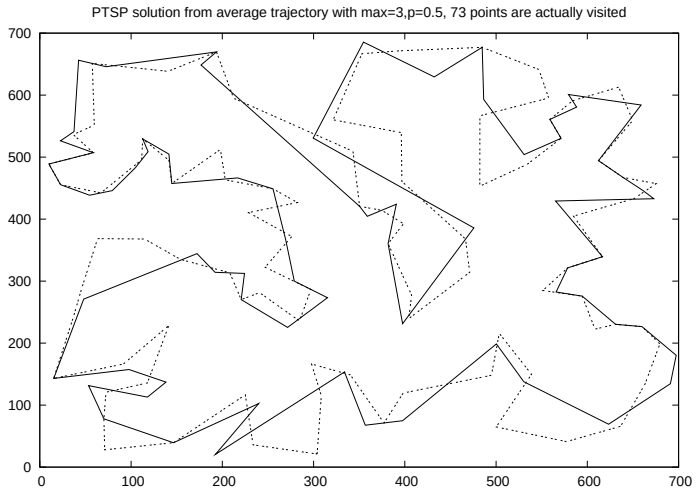


Figure 4.55: PTSP solution from quad tree with $p = 0.5, \text{Max} = 3$ (73 nodes are visited, solution value = 5239.254).

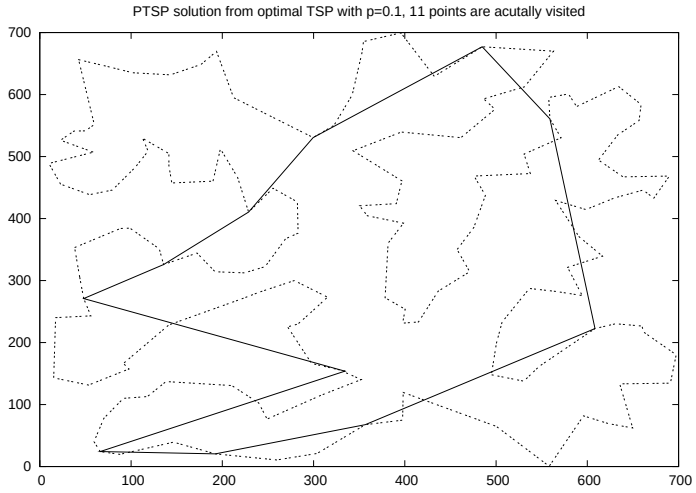


Figure 4.56: PTSP solution from optimal TSP with $p = 0.1$ (11 nodes are visited, solution value = 2295.000).

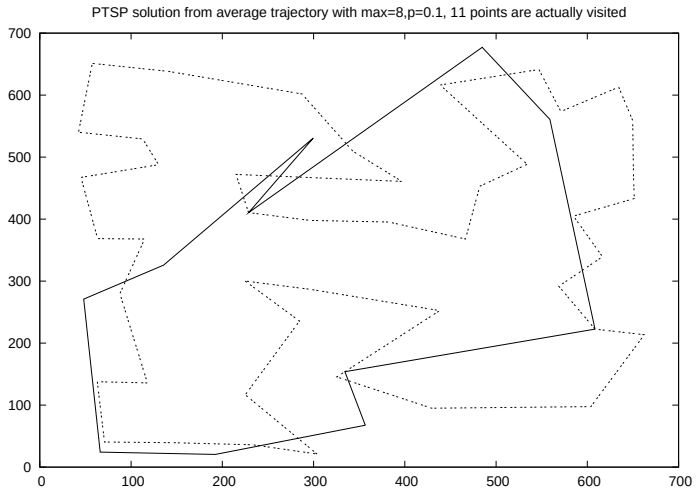


Figure 4.57: PTSP solution from quad tree with $p = 0.1, \text{Max} = 8$ (11 nodes are visited, solution value = 2270.721).

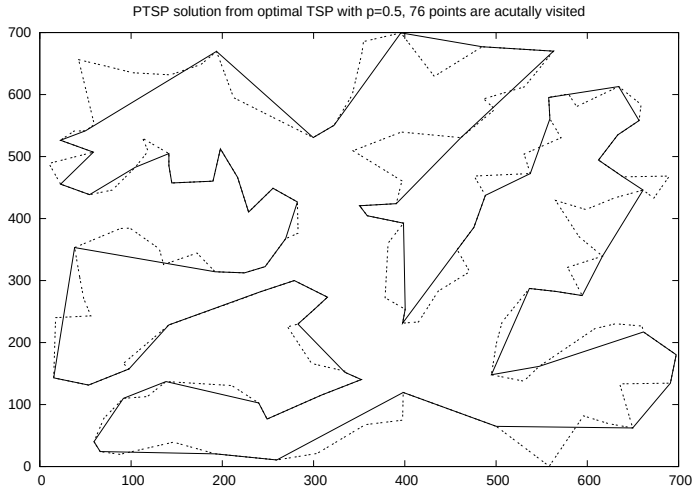


Figure 4.58: PTSP solution from optimal TSP with $p = 0.5$ (76 nodes are visited, solution value = 5204.818).

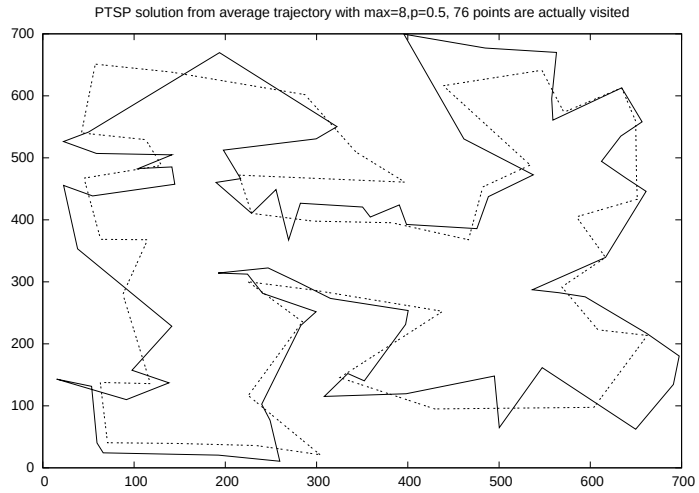


Figure 4.59: PTSP solution from quad tree with $p = 0.5, \text{Max} = 8$ (76 nodes are visited, solution value = 5171.242).

4.6 Conclusions and future work

In this chapter, we conducted extensive computational experiments using Braun and Buhmann's approach, a new quad tree approach that we developed, and a simple, convex hull cheapest insertion heuristic to generate average trajectories and final tours for the noisy traveling salesman problem. We used problems that had 100, 200, and 300 nodes. All three procedures quickly generated average trajectories (160 ms to 600 ms on average) and final tours (10 ms to 16 ms on average).

In Braun and Buhmann's average trajectory approach, we needed to set the values of three parameters, including the sample size and temperature. In the quad tree approach, we needed to set the value of one parameter (Max). When the value of Max was large, the quad tree approach generated final tours that were much lower in cost than the final tours generated by the average trajectory approach. When the problem variance was large, the average trajectory approach did not perform as well as the simple, convex hull cheapest insertion heuristic. For the most part, when the problem variability was small, the quad tree approach performed better than the average trajectory approach. We recommend using the quad tree approach to generate average trajectories and high-quality final tours for the noisy traveling salesman problem with small variability and using the convex hull, cheapest insertion heuristic for a problem with large variability. In other words, as long as variability is not too large, an average trajectory can be computed in advance and adapted each day, rather than having to solve a new routing problem from scratch each day.

Chapter 5

The Heterogeneous Vehicle Routing Problem

In the heterogeneous fleet vehicle routing problem (HVRP), several different types of vehicles can be used to service the customers. The types of vehicles differ with respect to capacity, fixed cost, and variable cost. We assume that the number of vehicles of each type is fixed and equal to a constant. We must decide how to make the best use of the fixed fleet of heterogeneous vehicles. In this chapter, we review methods for solving the HVRP, develop a variant of our record-to-record travel algorithm for the standard vehicle routing problem that takes a heterogeneous fleet into account, and report computational results on eight benchmark problems. Finally, we generate a new set of five test problems that have 200 to 360 customers and solve each new problem using our record-to-record travel algorithm.

5.1 Introduction

In the standard version of the vehicle routing problem (VRP), a fleet of homogeneous vehicles is based at a single depot. Each vehicle has the same capacity and must leave from and return to the depot. Each customer has a known demand and is serviced by exactly one visit of a single vehicle. A sequence of deliveries must be generated for each vehicle so that all customers are serviced

and the total distance traveled by the fleet is minimized.

In the *heterogeneous* fleet vehicle routing problem (HVRP), there are several different vehicle types. For vehicle type t, the capacity is Q_t, the fixed cost per vehicle is f_t, and the variable cost per unit distance is α_t. There are n_t vehicles of type t that are available for servicing the customers. Note that n_t might be very large or, essentially, unlimited.

In the *fixed* fleet version of the HVRP, the values of n_t are fixed. More specifically, the number of vehicles of type t is limited and the fleet composition is known in advance. We must decide how to make the best use of a fixed fleet of heterogeneous vehicles. This is the version of the HVRP studied in this chapter.

When the number of vehicles of type t is unlimited, we must determine the best composition of the fleet. We are not studying this version of the HVRP in this chapter and refer the reader to Gendreau et al. [17] for a literature review and computational results with a tabu search heuristic.

In Section 5.2, we review three algorithms that have been developed to solve the fixed fleet version of the HVRP. We develop a variant of our record-to-record travel algorithm (see Li et al. [28]) to handle a heterogeneous fixed fleet and report computational results for all four procedures on eight test problems taken from the literature.

In Section 5.3, we develop five new test problems that have 200 to 360 customers. We apply our record-to-record travel algorithm to these five problems and report our results. In Section 5.4, we give our conclusions.

5.2 Solving the fixed fleet version of the HVRP

5.2.1 Algorithms for the HVRP

Three algorithms have been developed to solve the fixed fleet version of the HVRP (see Table 5.1). One algorithm uses tabu search and integer linear programming (HCG) and two algorithms use threshold accepting (LBTA, BATA). No algorithm is guaranteed to produce an optimal solution. All three algorithms have been computationally tested on eight benchmark problems (more on this in the next two sections).

We adapted our record-to-record travel algorithm for the VRP (Li et al. [28]) to handle the HVRP and denote our algorithm by HRTR. The details of HRTR are given in Table 5.2.

We keep a record of the best solution (denoted by the Global Record) in our algorithm. In Step 5, we try to accept a solution that is slightly worse than the Global Record to avoid becoming trapped in a poor local minimum and we use it as the current solution in Step 2. The primary purpose is to diversify the search process.

5.2.2 Test problems

Golden et al. [19] developed eight test problems for the vehicle fleet size and mix routing problem which can be viewed as a special case of the HVRP where the travel costs are the same for all vehicle types and the number of vehicles of each type is unlimited. Taillard [38] adapted the Golden et al. problems to the HVRP by specifying the variable cost per unit distance for each type of vehicle and the number of vehicles of each type available. The specifications for the

Table 5.1: Three algorithms for solving the heterogeneous vehicle routing problem.

Authors	Algorithm	Comments
Taillard [38]	Heuristic Column Generation (HCG)	For each vehicle type and an unlimited number of vehicles, a homogeneous VRP is solved. A tabu search algorithm is used to generate a set of good solutions. Single vehicle routes are extracted and then combined into a partial solution using tabu search. The process is repeated and routes are "memorized" as candidates for the final solution to the HVRP. Once the homogeneous VRPs are solved for each vehicle type,"useless" routes are deleted leaving a set T of routes. T contains a small number of all the routes that can be generated for the homogeneous VRPs. The best solution to the HVRP is found by solving an integer linear problem (ILP). Each column in the ILP corresponds to a route in T.
Tarantilis, Kiranoudis, and Vassiliadis [44]	List-Based Threshold Accepting (LBTA)	Threshold accepting is a deterministic variant of. simulated annealing in which a threshold value T is specified as the upper bound on the amount of objective function increase allowed (uphill moves can be made). In the list-based algorithm, a list of values for T is used during the search.
Tarantilis, Kiranoudis, and Vassiliadis [45]	Backtracking Adaptive Threshold Accepting (BATA)	In the backtracking algorithm, T is allowed to increase during the search. The authors use two-opt moves, 1-1 exchanges (swap two customers from either the same or different routes), and 1-0 exchanges (move a customer from its position on one route to another position on either the same route or a different route) when performing local search.

HVRP problem set are given in Table 5.3. We use the numbering scheme (problem 13, ..., problem 20) given by Golden et al. [19].

The problems range in size from $n = 50$ to $n = 100$ customers. There are no route-length restrictions and no customer service times. The eight problems have between three and six types of vehicles available. In Table 5.3, in the right-most column labeled %, we compute the ratio of total customer demand to total capacity of all available vehicles. This provides an indication of how much potential flexibility we have in using the available vehicles. For six problems, we see that the percentages are greater than 94% and expect that nearly all of the available vehicles will be used in a solution.

5.2.3 Computational results

In Table 5.4, we present the solution values and the computation times for HCG, LBTA, BATA, and HRTR. The results for HCG, LBTA, and BATA are taken from the literature and were generated using a single set of parameter values. We see that HRTR generated six new best-known solutions (problems 13, 14, 16, 17, 18, 20) and produced the same best-known solution as BATA on problem 15. HCG generated the best-known solution to problem 19. LBTA did not generate any best-known solutions. Over all eight problems on different computing platforms, LBTA was the fastest procedure (1781 seconds) followed by HRTR (2286 seconds). In Figure 5.1 to Figure 5.8, we show the best-known solutions produced by HRTR to seven problems. In Table 5.5 to Table 5.12, we show the detailed routes for each problem.

Table 5.2: Record-to-record travel algorithm for the HVRP.

Step 0. Initialization Parameters are I, K, M and NBListSize. Set $I = 30, K = 5, M =$ Max(Number of nodes/2, 30) and NBListSize = 20. Set Global Record = ∞ Set Global Deviation = 0.01 $\times$ Global Record.
Step 1. Starting solution Generate a feasible solution using least cost insertion algorithm. Set Record = objective function value of the current solution. Set Deviation = 0.01 $\times$ Record.
Step 2. Improve the current solution. For $i = 1$ to I (I loop) Do One Point Move with record-to-record travel, Two Point Move with record-to-record travel between routes, and Two-opt Move with record-to-record travel. Feasibility must be maintained. If no feasible record-to-record move is made, go to Step 3. If a new record is produced, update Record and Deviation. End I loop.
Step 3. For the current solution, apply One Point Move (within and between routes), Two Point Move (between routes), Two-opt Move(within and between routes), and OR-opt move (within and between routes). Only downhill moves are allowed. If a new record is produced, update Record and Deviation.
Step 4. Repeat for K consecutive iterations. If no new record is produced, go to Step 5. Otherwise, go to Step 2.
Step 5. Update the Global Record and Global Deviation if the solution from Step 4 is better. Perturb the solution from Step 4 and repeat Step 2 to Step 4. Accept the perturbed solution if it is less than Global Record + Global Deviation.
Step 6. Repeat Step 2 to Step 5 until no better solution is found during M consecutive loops from Step 2 to Step 5.
Step 7. Report the best solution corresponding to the Global Record.

Table 5.3: Specifications for eight benchmark problems with six types of vehicles.

Problem		Vehicle Type																						
		A				B				C				D				E				F				
number	n	Q_A	f_A	α_A	n_A	Q_B	f_B	α_C	n_C	Q_C	f_C	α_C	n_C	Q_D	f_D	α_D	n_D	Q_E	f_E	α_E	n_E	Q_F	f_F	α_F	n_F	%
13	50	20	20	1.0	4	30	35	1.1	2	40	50	1.2	4	70	120	1.7	4	120	225	2.5	2	200	400	3.2	1	95.39
14	50	120	100	1.0	4	160	1500	1.1	2	300	3500	1.4	1													88.45
15	50	50	100	1.0	4	100	250	1.6	3	160	450	2.0	2													94.76
16	50	40	100	1.0	2	80	200	1.6	4	140	400	2.1	3													94.76
17	75	50	25	1.0	4	120	80	1.2	4	200	150	1.5	2	350	320	1.8	1									95.38
18	75	20	10	1.0	4	50	35	1.3	4	100	100	1.9	2	150	180	2.4	2	250	400	2.9	1	400	800	3.2	1	95.38
19	100	100	500	1.0	4	200	1200	1.4	3	300	2100	1.7	3													76.74
20	100	60	100	1.0	6	140	300	1.7	4	200	500	2.0	3													95.92

n number of customers
Q_t capacity of vehicle type t ($t = A, B, C, D, E, F$)
f_t fixed cost of vehicle type t
α_t variable cost per unit distance of vehicle type t
n_t number of vehicles of type t available
% 100× (total demand/total capacity)

Table 5.4: Computational results for HVRP algorithms on eight test problems.

	Taillard		Tarantilis et al.		Tarantilis et al.		Li et al.	
Problem	HCG	Time (s)	LBTA	Time (s)	BATA	Time (s)	HRTR	Time (s)
13	1518.05	473	1519.96	110	1519.96	843	**1517.84**	358
14	615.64	575	612.51	51	611.39	387	**607.53**	141
15	1016.86	335	1017.94	94	**1015.29**	368	**1015.29**	166
16	1154.05	350	1148.19	11	1145.52	341	**1144.94**	188
17	1071.79	2245	1071.67	221	1071.01	363	**1061.96**	216
18	1870.16	2876	1852.13	310	1846.35	971	**1823.58**	366
19	**1117.51**	5833	1125.64	309	1123.83	428	1120.34	404
20	1559.77	3402	1558.56	675	1556.35	1156	**1534.17**	447

Bold best-known solution
HCG heuristic column generation solution from Taillard [38]; Sun Sparc workstation, 50 MHz
LBTA list-based threshold accepting solution from Tarantilis et al. [44]; Pentium III, 500 MHz, 128 MB RAM
BATA backtracking adaptive threshold accepting solution from Tarantilis et al. [45]; Pentium II, 400 MHz, 128 MB RAM
HRTR record-to-record travel solution; Athlon 1 GHz, 256 MB RAM

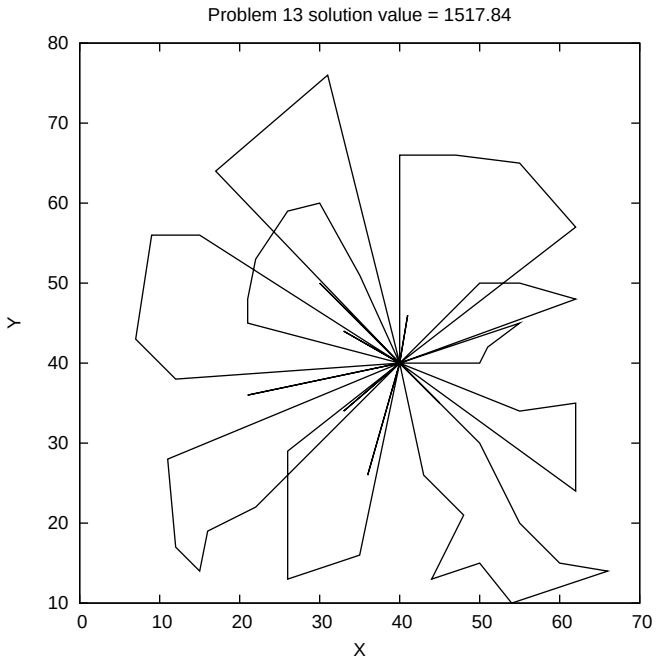


Figure 5.1: Problem 13.

Table 5.5: Routes for problem 13.

Problem 13, 50 customers, total load = 1020, total demand = 973				
Type	Capacity	Load	Distance	Sequence
A	20	19	18.44	0-6-0
A	20	18	12.17	0-26-0
A	20	19	38.83	0-16-0
A	20	20	16.12	0-17-0
B	30	30	15.56	0-4-0
B	30	26	32.03	0-2-0
C	40	33	33.94	0-40-0
C	40	37	73.72	0-27-13-15-0
C	40	39	106.55	0-25-31-0
C	40	40	59.80	0-7-35-19-0
D	70	62	56.18	0-34-46-8-0
D	70	68	115.27	0-33-28-22-0
D	70	67	142.76	0-49-24-18-50-0
D	70	68	142.69	0-23-41-42-43-1-0
E	120	117	153.65	0-12-39-9-32-44-3-0
E	120	118	198.74	0-10-38-11-14-0
F	200	192	301.40	0-45-29-5-37-20-36-47-21-48-30-0
Total Distance = 1517.84				

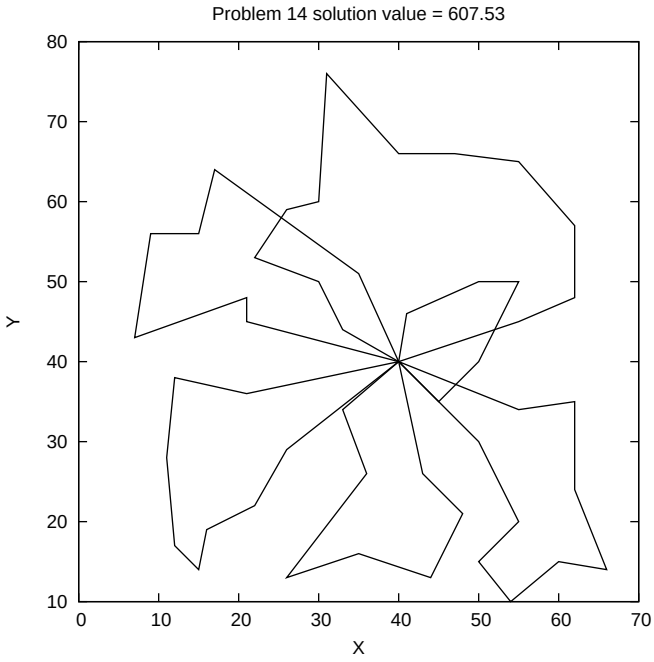


Figure 5.2: Problem 14.

Table 5.6: Routes for problem 14.

Problem 14, 50 customers, total load = 1100, total demand = 973				
Type	Capacity	Load	Distance	Sequence
A	120	119	91.65	0-16-49-23-41-42-43-1-33-0
A	120	119	46.25	0-4-34-46-35-7-26-0
A	120	120	99.20	0-12-25-50-18-24-44-3-0
B	160	156	91.82	0-30-48-21-28-22-2-6-0
B	160	159	107.46	0-27-13-15-20-37-36-47-5-29-45-0
C	300	300	171.15	0-17-39-32-9-39-31-10-38-11-14-19-8-0
Total Distance = 607.53				

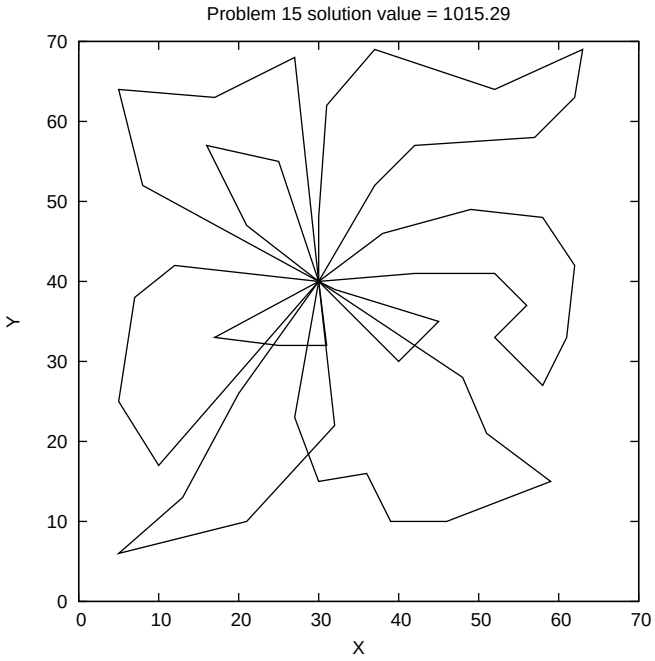


Figure 5.3: Problem 15.

Table 5.7: Routes for problem 15.

Problem 15, 50 customers, total load = 820, total demand = 777				
Type	Capacity	Load	Distance	Sequence
A	50	48	47.61	0-6-23-48-0
A	50	47	88.81	0-24-43-7-26-0
A	50	41	37.05	0-46-38-5-0
A	50	47	93.48	0-4-19-40-42-37-0
B	100	99	77.58	0-41-13-25-14-0
B	100	95	36.89	0-18-47-12-0
B	100	99	98.78	0-49-10-39-33-45-15-44-17-0
C	160	156	95.27	0-32-2-29-21-34-30-9-50-16-11-0
C	160	145	108.30	0-27-8-31-28-3-36-35-20-22-1-0
Total Distance = 1015.29				

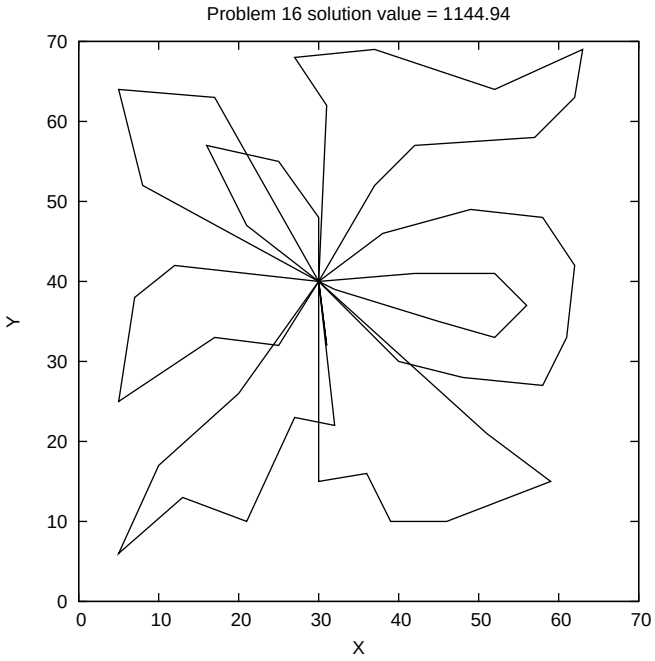


Figure 5.4: Problem 16.

Table 5.8: Routes for problem 16.

Problem 16, 50 customers, total load = 820, total demand = 777				
Type	Capacity	Load	Distance	Sequence
A	40	40	75.89	0-24-43-7-0
A	40	29	16.12	0-12-0
B	80	63	77.45	0-27-48-23-6-0
B	80	75	90.36	0-46-33-9-50-16-11-0
B	80	77	159.11	0-37-17-42-19-40-41-4-0
B	80	78	155.26	0-10-39-33-45-15-44-0
C	140	137	244.29	0-8-26-31-28-3-36-35-20-22-1-0
C	140	140	180.33	0-5-49-30-34-21-29-2-32-0
C	140	138	146.13	0-14-25-13-18-47-0
Total Distance = 1144.94				

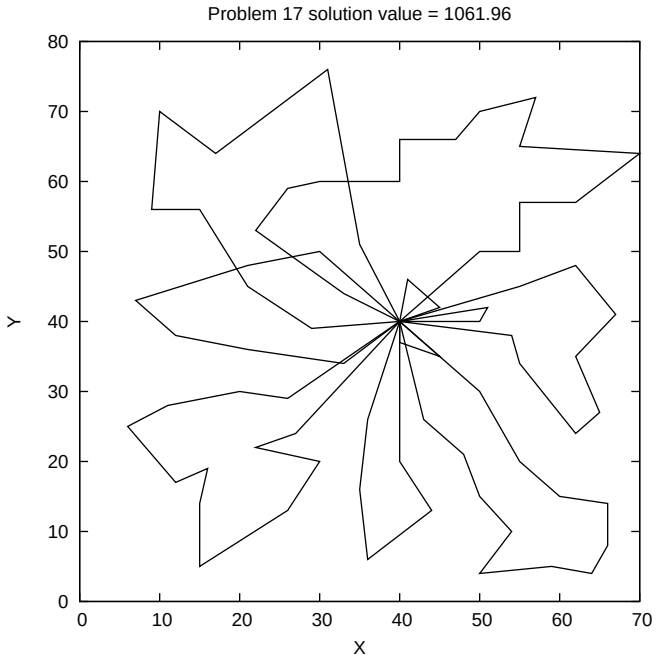


Figure 5.5: Problem 17.

Table 5.9: Routes for problem 17.

Problem 17, 75 customers, total load = 1430, total demand = 1364				
Type	Capacity	Load	Distance	Sequence
A	50	48	17.12	0-26-67-0
A	50	46	23.42	0-46-34-0
A	50	50	15.46	0-75-4-0
B	120	120	142.40	0-51-3-50-18-55-25-31-12-0
B	120	118	88.02	0-74-21-61-28-2-68-0
B	120	117	99.72	0-8-19-54-13-57-15-27-52-0
B	120	120	91.08	0-6-16-49-24-44-40-0
C	200	196	185.13	0-33-63-23-56-41-43-42-64-22-62-1-73-0
C	200	199	156.65	0-30-48-47-36-69-71-60-70-20-37-5-29-45-0
D	350	350	242.95	0-7-35-53-14-59-11-66-65-38-10-58-72-39-9-32-17-0
Total Distance = 1061.96				

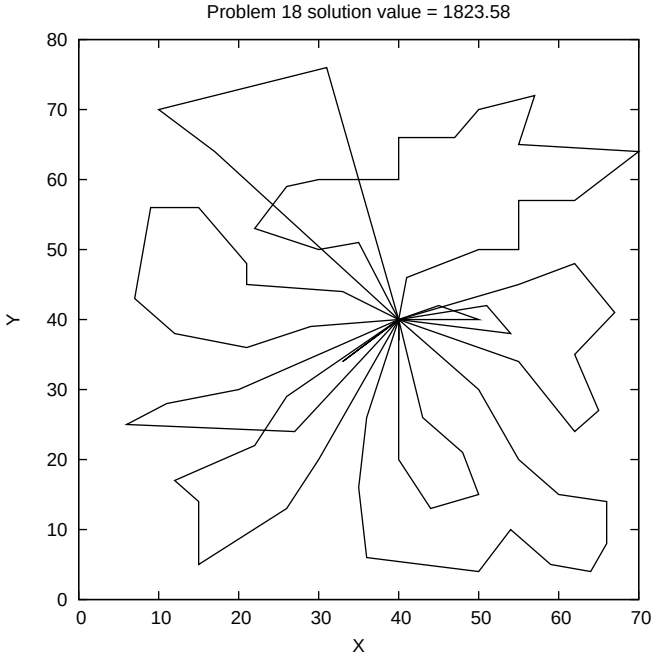


Figure 5.6: Problem 18.

Table 5.10: Routes for problem 18.

Problem 18, 75 customers, total load = 1430, total demand = 1364				
Type	Capacity	Load	Distance	Sequence
A	20	20	6.00	0-75-0
A	20	19	18.44	0-6-0
B	50	46	131.83	0-31-55-25-0
B	50	49	27.07	0-34-67-0
B	50	49	102.77	0-73-56-23-63-0
B	50	46	39.42	0-52-46-0
C	100	99	117.99	0-30-48-47-21-74-0
C	100	98	153.88	0-8-19-54-23-57-15-27-0
D	150	147	226.35	0-62-22-64-42-41-43-1-33-0
D	150	146	211.53	0-51-16-49-24-18-50-44-3-17-0
E	250	248	339.68	0-4-45-29-5-37-20-70-60-71-36-69-61-28-2-68-0
F	400	397	448.69	0-12-40-32-9-39-72-58-10-38-65-66-11-59-14-53-35-7-26-0
Total Distance = 1824.58				

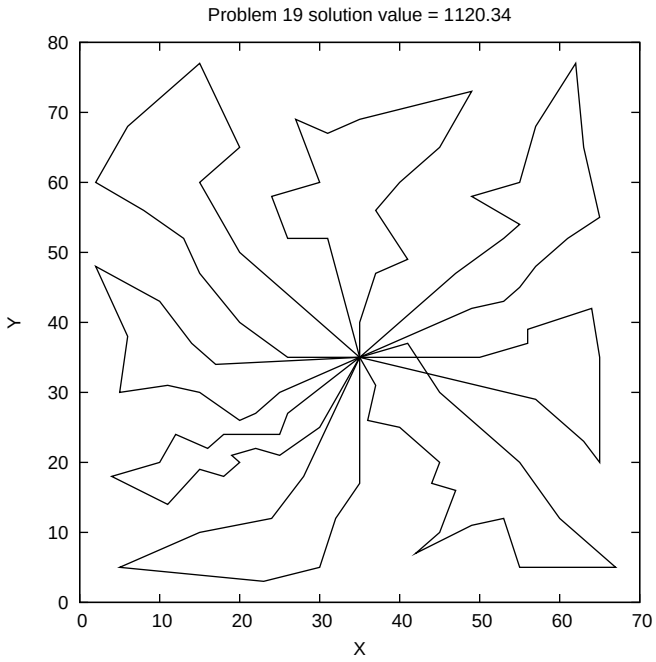


Figure 5.7: Problem 19.

Table 5.11: Routes for problem 19.

Problem 19, 100 customers, total load = 1900, total demand = 1458				
Type	Capacity	Load	Distance	Sequence
A	100	99	88.83	0-12-80-68-29-24-25-55-54-0
A	100	96	102.50	0-87-42-14-38-43-15-57-2-0
A	100	98	92.02	0-60-83-8-46-45-17-84-5-99-96-6-0
B	200	199	166.31	0-50-33-81-51-9-71-65-35-34-78-79-3-77-76-0
B	200	199	159.33	0-31-88-62–10-63-90-32-66-20-30-70-1-69-27-0
B	200	193	165.97	0-89-18-82-48-47-36-49-64-11-19-7-52-0
C	300	279	198.88	0-53-58-40-21-73-72-74-22-41-75-56-23-67-39-4-26-28-0
C	300	296	146.34	0-94-95-59-93-85-61-16-86-44-91-100-37-98-92-97-13-0
Total Distance = 1120.34				

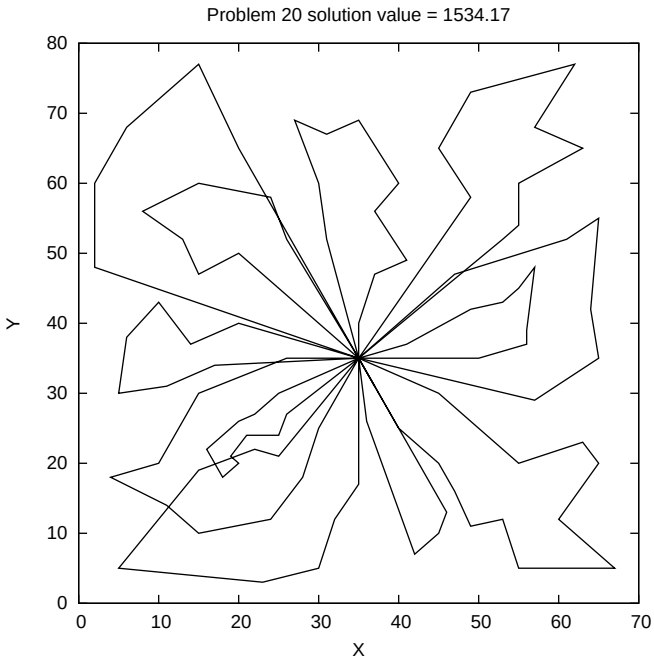


Figure 5.8: Problem 20.

Table 5.12: Routes for problem 20.

Problem 20, 100 customers, total load = 1520, total demand = 1458				
Type	Capacity	Load	Distance	Sequence
A	60	57	115.68	0-11-64-49-36-46-0
A	60	58	60.98	0-73-74-22-41-58-0
A	60	60	75.01	0-18-83-8-45-17-84-60-0
A	60	60	101.69	0-2-57-15-43-38-91-92-97-0
A	60	14	8.94	0-53-0
A	60	60	89.75	0-54-24-29-34-79-50-0
B	140	137	142.92	0-31-10-63-90-32-30-70-1-69-27-0
B	140	138	138.85	0-52-7-82-48-47-19-62-88-0
B	140	140	99.78	0-28-76-77-3-79-68-80-12-0
B	140	140	199.85	0-51-20-66-65-71-35-9-81-33-0
C	200	196	109.68	0-6-96-99-93-85-100-37-98-59-95-94-0
C	200	198	216.63	0-26-4-55-25-39-67-23-56-75-72-21-40-0
C	200	200	174.40	0-13-87-42-14-44-86-16-61-5-89-0
Total Distance = 1534.17				

5.3 New test problems and computational results

We selected five large-scale vehicle routing problems with 200 to 360 customers from Golden et al. [21] and adapted them to the HVRP. Each problem has a geometric symmetry with customers located in concentric circles around the depot. The problem generator is given in the Appendix. The specifications for the five new problems are given in Table 5.13.

For all five new problems, there are no route-length restrictions and no customer service times. One problem has four types of vehicles available, two problems have five types of vehicles available, and two problems have six types of vehicles available. For each of the five problems, we see that the ratio of total customer demand to the total capacity of all available vehicles is greater than 92% and expect that nearly all of the available vehicles will be used in a solution.

We applied HRTR with a single set of parameter values to the five new problems. In Table 5.14, we present the solution values and computation times. In Figure 5.9 to Figure 5.13, we show the routes. The routes are visually appealing and the computation times ranged from 11 minutes for the smallest problem ($n = 200$) to 54 minutes for the largest problem ($n = 360$).

Table 5.13: Specifications for five new problems with six types of vehicles.

Problem		Vehicle Type																						
		A				B				C				D				E				F				
number	n	Q_A	f_A	α_A	n_A	Q_B	f_B	α_C	n_C	Q_C	f_C	α_C	n_C	Q_D	f_D	α_D	n_D	Q_E	f_E	α_E	n_E	Q_F	f_F	α_F	n_F	%
H1	200	50	20	1.0	8	100	35	1.1	6	200	50	1.2	4	500	120	1.7	3	1000	225	2.5	1					93.02
H2	240	50	100	1.0	10	100	1500	1.1	5	200	3500	1.2	5	500	120	1.7	4									96.00
H3	280	50	100	1.0	10	100	250	1.1	5	200	50	1.2	5	500	120	1.7	4	1000	225	2.5	2					94.76
H4	320	50	100	1.0	10	100	200	1.1	8	200	400	1.2	5	500	120	1.7	2	1000	225	2.5	2	1500	250	3	1	94.12
H5	360	50	25	1.0	10	100	80	1.2	8	200	150	1.5	5	500	320	1.8	1	1500	225	2.5	2	2000	250	3	1	92.31

n number of customers
Q_t capacity of vehicle type t (t = A,B,C,D,E,F)
f_t fixed cost of vehicle type t
α_t variable cost per unit distance of vehicle type t
n_t number of vehicles of type t available
% 100× (total demand/total capacity)

Table 5.14: Computational results for HVRP algorithms on five new test problems.

		Li et al.	
Problem	n	HVRP	Time(s)
H1	200	12067.65	687.82
H2	240	10234.40	995.27
H3	280	16231.80	1437.56
H4	320	17576.10	2256.35
H5	360	21850.41	3276.91

HRTR Record-to-record travel solution; Athlon 1 GHz, 256 MB RAM

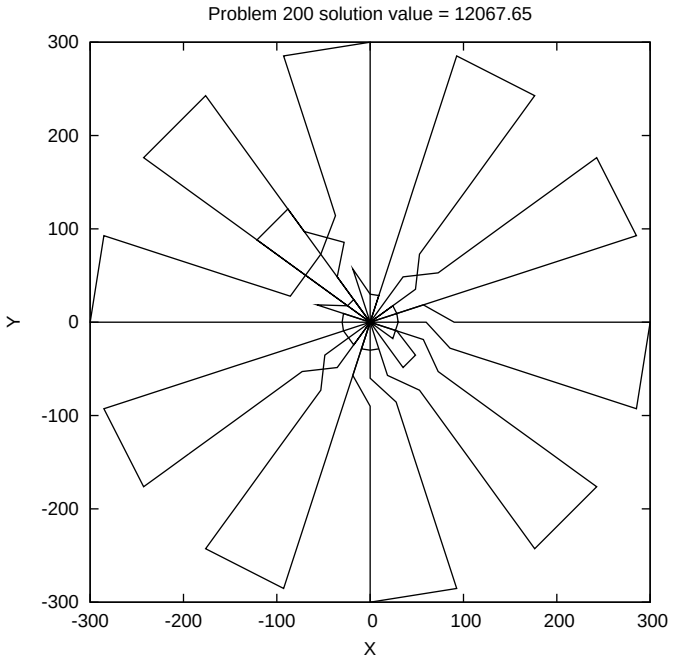


Figure 5.9: Problem H1.

5.4 Conclusions

We described three algorithms (HCG, LBTA, BATA) that have been proposed in the vehicle routing literature over the last six years to solve the HVRP with a fixed fleet. We presented our solution procedure (HRTR) that is based on the record-to-record travel algorithm and compared the results of HRTR to the results of the three algorithms on eight benchmark problems. We found that HRTR produced six new best-known solutions and was reasonably fast. Finally, we developed five new HVRPs and solved them with HRTR.

5.5 HVRP generator

$(x(i), y(i))$ is the coordinate of customer i, where $i = 0$ is the depot
$q(i)$ is the demand of customer i
A and B are parameters that determine the number of customers n, where $n = A \times B$
All data recorded to four decimal places

```
begin
      ω = 0
      x(ω) = 0, y(ω) = 0, q(ω) = 0
      for k := 1 to B do
      begin
   γ = 30k
   for i := 1 to A do
   begin
         ω = ω + 1
         x(ω) = γ cos [2(i − 1)π/A]
         y(ω) = γ sin [2(i − 1)π/A]
         if mod(i, 4) = 2 or 3
         then q(ω) = 30
         else q(ω) = 10
         end
   end
      end
end
```

Problem	A	B	n
H1	20	10	200
H2	40	6	240
H3	28	10	280
H4	40	8	320
H5	36	10	360

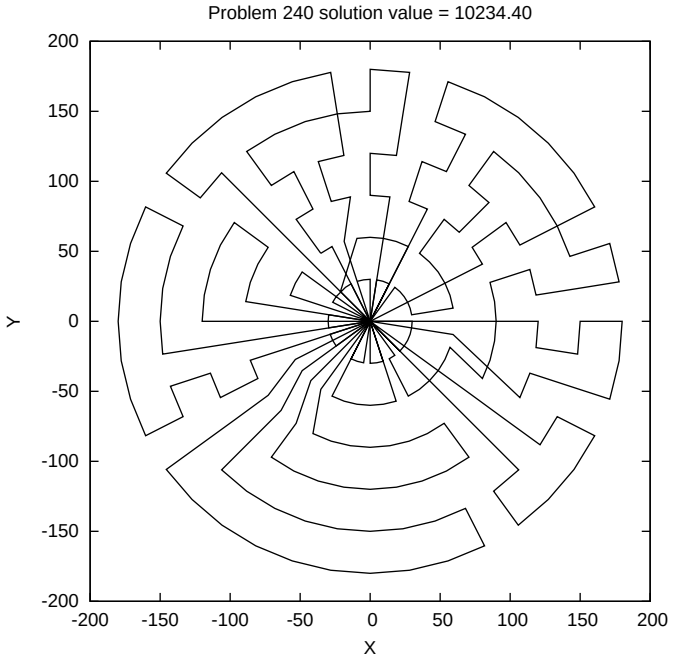


Figure 5.10: Problem H2.

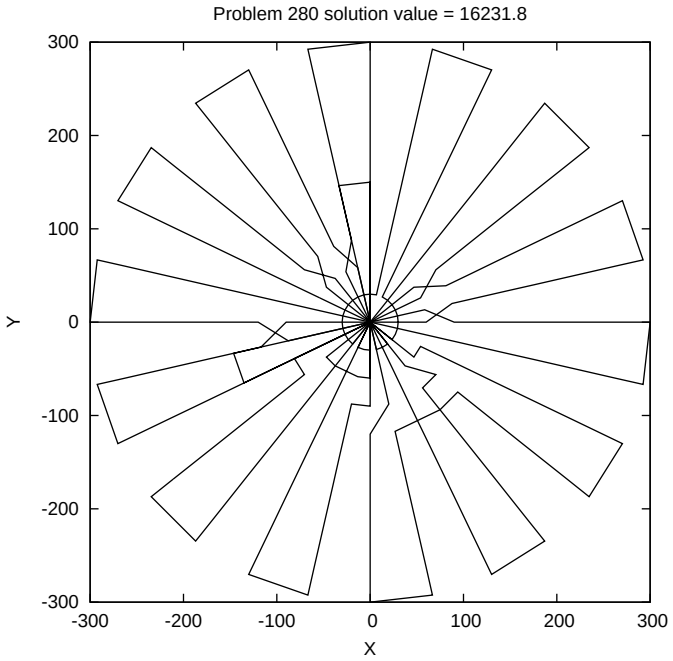


Figure 5.11: Problem H3.

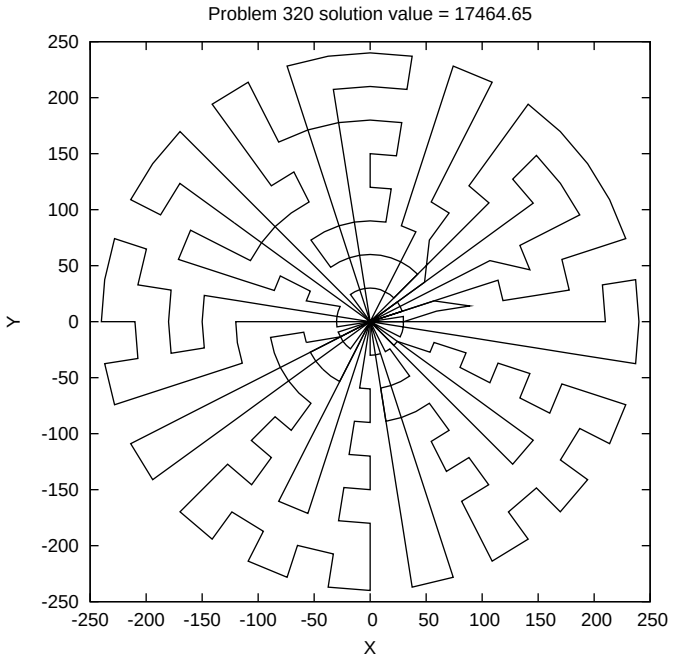


Figure 5.12: Problem H4.

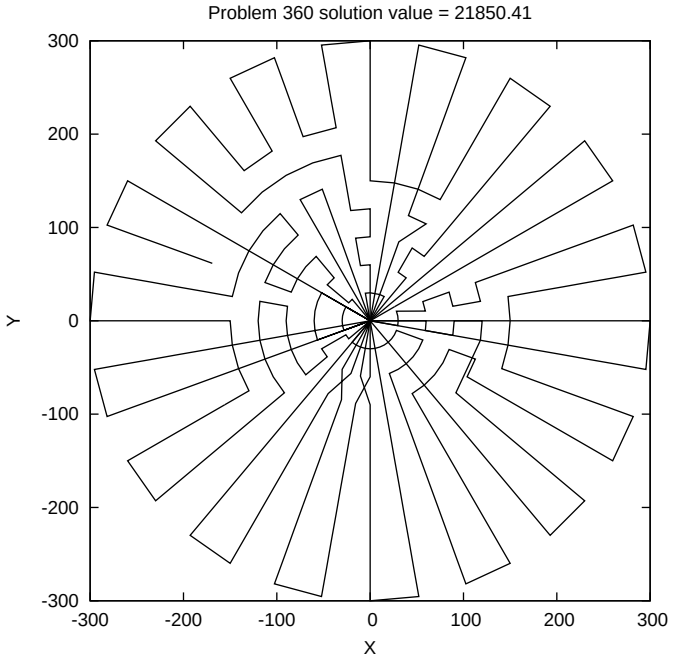


Figure 5.13: Problem H5.

Chapter 6

The Open Vehicle Routing Problem

In the open vehicle routing problem (OVRP), a vehicle does not return to the depot after servicing the last customer on a route. The description of this variant of the standard vehicle routing problem appeared in the literature over 20 years ago, but has just recently attracted the attention of researchers and practitioners. In the last five years, tabu search and deterministic annealing have been applied to the OVRP with some success. In this chapter, we review OVRP algorithms, develop a variant of our record-to-record travel algorithm for the standard vehicle routing problem to handle open routes, and report computational results on test problems taken from the literature. Finally, we develop a new set of eight test problems that range in size from 200 to 480 nodes and report solutions generated by our record-to-record travel algorithm on the new test set.

6.1 Introduction

In the standard version of the vehicle routing problem (VRP), we generate a sequence of deliveries for each vehicle in a homogeneous fleet based at a single depot so that all customers are serviced and the total distance traveled by the fleet is minimized. Each vehicle has a fixed capacity and perhaps a route-length restriction that limits the maximum distance it can travel. Each customer has a

known demand and is serviced by exactly one visit of a single vehicle. Each vehicle must leave from and return to the depot.

In the open vehicle routing problem (OVRP), a vehicle does not return to the depot after servicing the last customer on a route. Each route in the OVRP is a Hamiltonian path over the subset of customers visited on the route. (A Hamiltonian path is a path which visits each customer in the subset precisely once.) In addition, we need to find the minimum number of vehicles that are required to service all of the customers.

One of the earliest descriptions of the OVRP was by Schrage [37] in a paper that tried to classify the key features of VRPs found in practice.

> "A vehicle can be characterized by at least the following three characteristics: its (multidimensional) capacity, cost rate, and whether it makes open or closed trips. In a closed trip, a vehicle returns to its starting location; in an open trip, it may not. For example, relative to private vehicles, common carrier vehicles tend to have a higher cost/kilometer; however, they make open rather than closed trips. An air express courier which has planes depart from a single depot city early in the morning making deliveries and then has each plane retrace its route late in the evening making pickups effectively has open routes."

Bodin et al. [6] describe the OVRP encountered by FedEx in generating "incomplete" delivery routes for airplanes. An airplane leaves Memphis, makes deliveries to several cities, and does not return to Memphis. The airplane lays over at the last city on the delivery route and then starts its pickups from the layover city. Bodin et al. describe a variant of the Clarke and Wright algorithm that was used by FedEx to develop an open route for each airplane.

Currently, FedEx experiences the OVRP in its Home Delivery service to residential-only customers (Levy [27]). FedEx contracts couriers with vehicles who drive into the FedEX depot each morning, load packages, and then make deliveries to residences. The couriers and vehicles do not return to the FedEx

depot after their last deliveries of the day.

From the early 1980s to the late 1990s, the OVRP received very little attention in the operations research literature. However, since 2000, several researchers have used tabu search and deterministic annealing to solve the OVRP with some success. In Section 6.2, we review seven algorithms that have been used recently to solve the OVRP. We develop a variant of our record-to-record travel algorithm (see Li et al. [28]) to handle open routes and report computational results for all seven procedures on 16 test problems taken from the literature. In Section 6.3, we develop eight new OVRPs that have 200 to 480 customers. We apply our record-to-record travel algorithm to these eight problems and report our results. In Section 6.4, we summarize our contributions.

6.2 Solving the open vehicle routing problem

6.2.1 Algorithms for the OVRP

Since 2000, seven algorithms have been developed to solve the OVRP. Two algorithms use threshold accepting, three use tabu search, one uses large neighborhood search, and one uses the minimum spanning tree. All of the algorithms are based on heuristic methods, so that an optimal solution is not guaranteed. In Table 6.1 we summarize the seven OVRP algorithms. We point out that Brandao [7] generates a starting solution in two different ways, so that there are two variants of his TSA procedure (denoted by TSAK and TSAN). This same holds for Fu et al. [16], so that there are two variants of their TS procedure (denoted by TSF and TSR). Pisinger and Ropke [32] run ALNS for 25,000 iterations and 50,000 iterations (these variants are denoted by ALNS 25K

and ALNS 50K). All ten procedures (CFRS, TSAK, TSAN, BR, BATA, LBTA, TSF, TSR, ALNS 25K, ALNS 50K) have been computationally tested on benchmark problems (more about this later).

We adapted our record-to-record travel algorithm that was developed to handle very large instances of the standard VRP (Li et al. [28]) to solve the OVRP (we denote our algorithm by ORTR). The details of ORTR are given in Table 6.2.

In ORTR, we use a fixed-length neighbor list with 20 customers. We generate an initial feasible solution using a sweep algorithm. Each customer is a starting point in the sweep algorithm. We compare the number of vehicles used with the minimum number of vehicles needed to service the customers. If they are equal, we stop and use that solution as our initial solution. Otherwise, we select the first solution we find with the minimum number of vehicles.

In Step 4 of ORTR, we try to combine routes by merging two routes. For example, routes A and B can be merged if the combined demand does not exceed vehicle capacity and the total distance traveled does not exceed the maximum route length. Of course, it is not always possible to find the best way to merge two routes quickly and efficiently. In our implementation, we try to insert all of route A between each link of route B.

Table 6.1: Seven algorithms for solving the open vehicle routing problem.

Authors	Algorithm	Comments
Sariklis and Powell [35]	Cluster First, Route Second (CFRS)	In the first phase of their heuristic, the authors form clusters of customers taking into acount vehicle capacity and then balance the clusters by reassigning customers. In the second phase, the authors generate open routes by solving a minimum spanning tree problem (MSTP). They use penalties to modify the MSTP solution and iteratively conver infeasible solutions to feasible solutions.
Brandao [7]	Tabu Search Algorithm (TSA)	The authors generate an initial solution using a variety of methods including a nearest neighbor heuristic and the K-tree method. The initial solution is submitted to either a nearest neighbor method or an unstringing and stringing procedure to improve each route. In the tabu search algorithm, there are only two types of trial moves: (1) an insert move (take a customer from one route and insert it onto another route), and (2) a swap move that exchanges two customers on two different routes.
Tarantilis, Diakoulaki, and Kiranoudis [40]	Adaptive Memory-Based Tabu Search (BR)	The authors extract a sequence of points (called bones) from a set of solutions and generate a route using adaptive memory. If a large number of routes in the set of solutions contains a specific bone, then the authors argue that this bone should be included in a route that appears in a high-quality solution. The BoneRoute algorithm has two phases. In Phase 1 (pool generation phase), an initial pool of routes is generated using weighted savings. The solutions are then improved using a standard tabu search algorithm. In Phase 2 (pool exploitation phase), promising bones are extracted, a solution is generated and improved using tabu search, and the set of solutions is updated.

Table 6.1 (continued)

Authors	Algorithm	Comments
Tarantilis, Ioannou, Kiranoudis, and Prastacos [41]	Backtracking Adaptive Threshold Accepting (BATA)	Threshold accepting is a deterministic variant of simulated annealing in which a threshold value T is specified as the upper bound on the amount of objective function increase allowed (uphill moves can be made). In the backtracking algorithm, T is allowed to increase during the search. In the list-based algorithm, a list of values of T is used during the search. The authors use two-opt moves, 1-1 exchanges (swap two customers from either the same or different routes), and 1-0 exchanges (move a customer from its position on one route to another position on the same route or a different route) when performing local search.
Tarantilis, Ioannou, Kiranoudis, and Prastacos [42]	List-Based Threshold Accepting (LBTA)	
Fu, Eglese, and Li [16]	Tabu Search Heuristic (TS)	A farthest first heuristic (FFH) is developed to generate an initial solution. FFH starts a new route with the farthest unrouted customer from the depot and tries to add customers to the route until the vehicle is sufficiently full. In the tabu search heuristic, two different nodes (customer or depot, on the same route or different routes) are selected at random and one of the four types of neighborhood moves is performed at random: (1) node reassignment, (2) node swap, (3) two-opt move, and (4) tails swap (select two customers and swap the tails, that is, perform an exchange from the customer to the end of the route).
Pisinger and Ropke [32]	Adaptive Large Neighborhood Search (ALNS)	In the ALNS framework, a feasible solution is constructed and then modified. In each iteration, an algorithm is selected to "destroy" the current solution and an algorithm is selected to "repair" the current solution. For example, customers can be removed at random from the solution and then reinserted in the cheapest possible route. Several removal and insertion heuristic can be used to diversify and intensify the search. The new solution is accepted if it satisfies the criteria defined by the local search procedure (the authors use simulated annealing).

Table 6.2: Record-to-record travel algorithm for the OVRP.

Step 0. Initialization Parameters are I, K, M and NBListSize. Set $I = 2, K = 5, M =$ Max(Number of nodes/2, 30) and NBListSize = 20.
Step 1. Starting solution Generate a feasible solution using the sweep algorithm. Set Record = objective function value of the current solution. Set Deviation = 0.01 $\times$ Record.
Step 2. Improve the current solution. For $i = 1$ to I (I loop) Do One Point Move with record-to-record travel, Two Point Move with record-to-record travel between routes, and Two-opt Move with record-to-record travel. Feasibility must be maintained. If no feasible record-to-record move is made, go to Step 3. If a new record is produced, update Record and Deviation. End I loop.
Step 3. For the current solution, apply One Point Move (within and between routes), Two Point Move (between routes), Two-opt Move(within and between routes), and Three-opt move (within routes). Only downhill moves are allowed. If a new record is produced, update Record and Deviation.
Step 4. Combining routes Try to insert each route between each pair of consecutive nodes of another route. Feasibility must be maintained. If it is possible, we make the insertion even if the total distance traveled increases since we want to use as few vehicles as possible.
Step 5. Repeat for K consecutive iterations. If no new record is produced, go to Step 5. Otherwise, go to Step 2.
Step 6. Perturb the solution from Step 5. Repeat Step 2 to Step 5 and keep the better solution
Step 7. Repeat Step 2 to Step 6 until no better solution is found during M consecutive loops.

6.2.2 Test Problems

There are 16 test problems available in the literature and they are summarized in Table 6.3. The fourteen problems denoted C1 to C14 are from Christofides et al. [10] and the two problems denoted F11 and F12 are from Fisher [15]. All 16 problems are available on line at either www.branchandcut.org/VRP/data/ or http://people.brunel.ac.uk/mastjjb/jeb/info.html.

The test problems range in size from $n = 50$ to $n = 199$ customers. We let K_{min} denote the minimum number of vehicles needed to service all of the customers ($K_{min} = \frac{\sum_{i=1}^{n} q_i}{Q}$) and L denote the maximum route length (this is the original VRP route length restriction multiplied by 0.9, since the original VRP restrictions on route length are loose and not binding for many solutions). Seven of the problems have a route-length restriction.

6.2.3 Computational results

In Table 6.3, we present the results (solution values and running times) generated by OVRP algorithms on the test problems. With the exception of ORTR, all results are taken from the literature. Seven algorithms were applied to all 16 problems: two variants of Brandao's tabu search algorithm (TSAK and TSAN), two variants of Fu, Eglese, and Li's tabu search heuristic (TSF and TSR), two variants of Pisinger and Ropke's adaptive large neighborhood search procedure (ALNS 20K and ALNS 50K), and ORTR. The remaining four procedures (CFRS, BR, BATA, LBTA) were applied to the seven problems that did not have a route-length restriction. The results for the 11 algorithms (CFRS, TSAK, TSAN, BR, BATA, LBTA, TSF, TSR, ALNS 25K, ALNS 50K, ORTR) were generated using a single set of parameter values. We point out that

Brandao [7] also provides the best solutions he encountered in all of his computational experiments (we do not give his best solutions in this paper).

There are a few points that we need to make with respect to Table 6.3. When generating a solution to the OVRP, there are two objectives that need to be considered: (1) minimize the total number of vehicles that are required to service all of the customers and (2) minimize the total distance traveled by the vehicles. Most researchers (e.g., Fu et al. [16]) assume that the cost of an additional vehicle far exceeds the reduction in distance that can be achieved by an additional route (so do we). In Table 6.3, we highlight in bold the solution for each problem that minimizes the total number of vehicles. For example, for problem C2, ALNS 25K, ALNS 50K, and ORTR generated a total distance of 567.14 with 10 vehicles and we highlight this in bold. BR, BATA, and LBTA each generated a slightly lower total distance of 564.06, but required 11 vehicles. Overall, we prefer solutions with the total number of vehicles as close to K_{min} as possible, even if the total distance is somewhat higher. For six problems, no solution used K_{min} vehicles (these six problems had route-length restrictions, so that the value of K_{min} was too low; K_{min} appears to be a good lower bound when only capacity constraints are considered). For example, for problem C6, TSF generated a total distance of 400.6 with 6 vehicles $> K_{min} = 5$ vehicles. Finally, we note that, on problem C4, the solution generated by BATA is infeasible; the solution reported by Tarantilis et al. [41] services only 138 of the 150 customers.

In Table 6.4, we compare the results of the 11 OVRP algorithms on the test set of problems with respect to both objectives (minimize total number of vehicles and minimize total distance traveled). When the number of vehicles is minimized, ALNS 50K generated the best solutions to nine problems, followed by ALNS 25K with the best solutions to seven problems, and ORTR with the best

solutions to five problems. When total distance traveled is minimized, TSR generated the best solutions to five problems and ORTR generated the best solution to four problems.

In Table 6.4, if we aggregate the best results across all 16 problems on both dimensions, we have the following observations: (1) the solutions that minimize the total number of vehicles use 156 vehicles (the minimum number is 147) and have a total distance traveled of 10,233.33; (2) the solutions that minimize the total distance traveled have total distance of 9,943.90 and use 165 vehicles.

In Figure 6.1 to Figure 6.5, we show the routes produced by ORTR for five problems (C2, C4, C12, C14, F12). For each problem, ORTR (along with other algorithms such as ALNS 25K and ALNS 50K) produced the best solutions with respect to minimizing the number of vehicles.

In Table 6.5, we provide aggregate statistics for 10 OVRP algorithms. All 10 algorithms solve all seven problems without route-length constraint. We did not include BATA in Table 6.5 because it solved only six problems (recall the solution to C4 was infeasible), thereby making comparisons using aggregate statistics across algorithms that solved seven problems difficult. When we sum the values of K_{min} for all seven problems, a minimum of 68 vehicles is required to service all of the customers. Eight of the 10 algorithms (CFRS, TSAK, TSAN, TSF, TSR, ALNS 25K, ALNS 50K, ORTR) generated solutions using 68 vehicles, while BR and LBTA needed substantially more vehicles (75). The distances produced by six algorithms that used 68 vehicles (TSAK, TSF, TSR, ALNS 25K, ALNS 50K, ORTR) ranged from 4,470 (ALNS 50K) to 4,557 (TSF) and these distances compare favorably to 4,433 (LBTA) and 4,445 (BR) using 75 vehicles.

In Table 6.5, we provide aggregate statistics for seven OVRP algorithms that solve all 16 problems. We see that five algorithms – TSAK, TSAN, ALNS 25K,

ALNS 50K, ORTR – performed very well using 156 or 159 vehicles and generating distances that ranged from 10,191 to 10,776. These results compare very favorably to the aggregate statistics of the best solutions which used 156 vehicles and generated a total distance of 10,233.33 (these values are given in the Total row in Table 6.4). Finally, over all 16 problems, TSAN was the fastest procedure (405 seconds), followed by ORTR (1,756 seconds). In contrast, on the set of seven problems, CFRS was extremely fast, using a total of only seven seconds, but it generated low-quality solutions with respect to distance. On all 16 problems, ALNS 25 and ALNS 50K were very slow using 13,350 seconds and 22,200 seconds, respectively for 10 runs of each algorithm on a fast computer.

Table 6.3: Computational results for OVRP algorithms on sixteen test problems.

				Sariklis and Powell		Brandao		Brandao		Tarantilis et al.	
Problem	n	L	K_{min}	CFRS	Time(s)	TSAK	Time(s)	TSAN	Time(s)	BR	Time(s)
C1	50		5	488.204	0.22	416.1	88.8	438.2	1.7	(6)412.96	7.2
C2	75		10	795.334	0.16	574.5	167.5	584.7	4.9	(11)564.06	25.8
C3	100		8	815.042	0.94	641.6	325.3	643.4	12.3	(9)641.77	28.8
C4	150		12	1034.139	0.88	740.8	870.2	767.4	33.2	735.47	75
C5	199		16	1349.709	2.20	953.4	1415.0	1010.9	116.9	(17)877.13	225.6
C6	50	180	5			(6)412.96	55.8	(6)416.00	1.4		
C7	75	144	10			634.54	123.7	(11)580.97	3.4		
C8	100	207	8			(9)644.63	351.7	(9)652.09	10.4		
C9	150	180	12			(13)785.2	622.2	(14)827.54	25.2		
C10	199	180	16			(17)884.63	2060.3	(17)946.84	100.1		
C11	120		7	828.254	1.54	683.4	696.0	713.3	15.7	(10)679.38	29.4
C12	100		10	882.265	0.76	535.1	233.6	543.2	7.8	**534.24**	14.4
C13	120	648	7			(11)943.66	401.9	(11)994.26	25.8		
C14	100	936	10			(11)597.31	419.8	(12)651.92	8.1		
F11	71		4			177.40	398.1	179.45	5.7		
F12	134		7			781.21	1000.2	825.9	32.7		

( )	Number of vehicles required if diffrent from K_{min}
Bold	Indicates the solution that minimizes the number of vehicles
CFRS	Cluster first, route second solution from Sariklis and Powell [35]; Pentium 133 MHz, 16 MB RAM
TSAK	Tabu search algorithm solution (initial solution given by K-tree with unstringing and stringing) from Brandao [7]; Pentium III, 500 MHz
TSAN	Tabu search algorithm solution (initial solution given by nearest neighbor heuristic) from Brandao [7]; Pentium III, 500 MHz
BR	Solution found by BoneRoute algorithm from Tarantilis et al. [40]; Pentium II, 400 MHz, 128 MB RAM

Table 6.3 (continued)

				Tarantilis et al.		Tarantilis et al.		Fu et al.		Fu et al.	
Problem	n	L	K_{min}	BATA	Time(s)	LBTA	Time(s)	TSF	Time(s)	TSR	Time(s)
C1	50		5	(6)412.96	38.62	(6)412.96	28.75	**408.5**	170.2	413.3	65.0
C2	75		10	(11)564.06	68.89	(11)564.06	61.21	587.8	202.1	570.6	197.8
C3	100		8	642.42	56.54	(9)639.57	53.78	644.3	719.9	**617.0**	367.6
C4	150		12	(11)736.89	81.69	733.68	84.13	734.5	1610.3	741.1	1094.0
C5	199		16	**879.37**	98.13	(17)870.26	96.47	(17)878.0	2060.6	(17)886.6	1279.9
C6	50	180	5					(6)**400.6**	128.0	(6)409.7	184.9
C7	75	144	10					(11)565.7	292.4	(11)560.4	231.8
C8	100	207	8					(9)**638.2**	987.8	(9)647.7	321.0
C9	150	180	12					(14)758.9	1635.2	(14)752.0	799.5
C10	199	180	16					(18)891.3	1922.2	(18)898.2	1218.2
C11	120		7	(9)679.60	37.67	(10)678.54	25.36	753.8	735.8	716.5	88.9
C12	100		10	**534.24**	84.54	**534.24**	64.59	549.9	413.4	534.8	30.9
C13	120	648	7					(12)943.0	741.1	(12)952.4	35.0
C14	100	936	10					(12)586.8	463.2	(12)469.3	155.5
F11	71		4					178.0	256.0	**175.0**	151.7
F12	134		7					789.7	1044.8	778.5	125.4

( )	Number of vehicles required if diffrent from K_{min}
Bold	Indicates the solution that minimizes the number of vehicles
BATA	Backtracking adaptive threshold accepting solution from Tarantilis et al. [41]; Pentium II, 400 MHz, 128 MB RAM
LBTA	List-based threshold accepting solution from Tarantilis et al. [42]; Pentium II, 400 MHz, 128 MB RAM
TSF	Tabu search solution (initial solution generated by farthest first heuristic) from Fu et al. [16]; Pentium II, 600 MHz, 128 MB RAM
TSR	Tabu search solution (initial solution generated randomly) from Fu et al. [16]; Pentium II, 600 MHz, 128 MB RAM

Table 6.3 (continued)

				Pisinger and Ropke		Pisinger and Ropke		Li et al.	
Problem	n	L	K_{min}	ALNS 25K	Time(s)	ALNS 50K	Time(s)	ORTR	Time(s)
C1	50		5	416.06	120	416.06	230	416.06	6.2
C2	75		10	**567.14**	360	**567.14**	530	**567.14**	31.3
C3	100		8	641.76	850	641.76	1280	639.74	39.5
C4	150		12	**733.13**	1790	**733.13**	2790	**733.13**	128.6
C5	199		16	897.93	1240	896.08	2370	924.96	380.6
C6	50	180	5	(6)412.96	200	(6)412.96	310	(6)412.96	10.3
C7	75	144	10	584.15	180	**583.19**	330	(11)568.49	32.2
C8	100	207	8	(9)645.31	730	(9)645.16	1140	(9)644.63	53.2
C9	150	180	12	(13)759.35	1080	(13)**757.84**	1850	(14)756.38	195.1
C10	199	180	16	(17)**875.67**	1200	(17)**875.67**	2240	(17)876.02	363.5
C11	120		7	**682.12**	730	**682.12**	1410	682.54	121.6
C12	100		10	**534.24**	800	**534.24**	1180	**534.24**	32.9
C13	120	648	7	(11)**909.80**	610	(11)**909.80**	1160	(12)896.50	120.3
C14	100	936	10	(11)**591.87**	400	(11)**591.87**	750	(11)**591.87**	62.9
F11	71		4	177.00	690	177.00	1040	177.00	19.5
F12	134		7	770.17	2370	770.17	3590	**769.66**	158.2

( )	Number of vehicles required if diffrent from K_{min}
Bold	Indicates the solution that minimizes the number of vehicles
ALNS25K	Adaptive large neighborhood search solution (25,000 iterations) from Pisinger and Ropke [32]; Pentium IV, 3GHz
ALNS50K	Adaptive large neighborhood search solution (50,000 iterations) from Pisinger and Ropke [32]; Pentium IV, 3GHz
ORTR	Record-to-record travel solution; Athlon 1GHz, 256 MB RAM

6.3 New test problems and computational results

We now consider the large-scale vehicle routing problems developed by Golden et al. [21]. We selected eight problems with 200 to 480 customers that do not have route-length restrictions. Each problem has a geometric symmetry with customers located in concentric circles around the depot. The problem generator is given in the Appendix.

Each problem exhibits a geometric symmetry that allows us to visually estimate a solution. In Figure 6.19 to Figure 6.13, we show our estimated solutions for all eight problems (problem O1 with $n = 200$ to problem O8 with $n = 480$). The routes are visually appealing. We applied ORTR with a single set of parameter values to the eight new problems. In Figure 6.14 to Figure 6.21, we show the routes produced by ORTR. In Table 6.6, we present the solution values and the running times generated by ORTR and the estimated solution values. For each problem, ORTR used the minimum number of vehicles as specified by the lower bound of K_{min}. ORTR used a total of 68 vehicles, while the estimated solutions used a total of 72 vehicles. For all eight problems, ORTR produced solution values that were less than the estimated solution values. On average, ORTR generated solution values that were 1.57% less than the values of the estimated solutions. ORTR was reasonably fast with computation times that ranged from 365.3 seconds (6 minutes) for 200 customers to 1,126.8 seconds (19 minutes) for 480 customers.

Table 6.4: Comparing the results of OVRP algorithms on sixteen test problems.

Problem	K_{min}	Minimize Vehicles		Minimize Distance	
C1	5	408.5	TSF	408.5	TSF
C2	10	567.14	ALNS 25K, ALNS 50K, ORTR	(11)564.06	BR, BATA, LBTA
C3	8	617.0	TSR	617.0	TSR
C4	12	733.13	ALNS 25K, ALNS 50K, ORTR	733.13	ALNS 25K, ALNS 50K, ORTR
C5	16	879.37	BATA	(17)870.26	LBTA
C6	5	(6)400.6	TSF	(6)400.6	TSF
C7	10	583.19	ALNS 50K	(11)560.4	TSR
C8	8	(9)638.2	TSF	(9)638.2	TSF
C9	12	(13)757.84	ALNS 50K	(14)752.0	TSR
C10	16	(17)875.67	ALNS 25K, ALNS 50K	(17)875.67	ALNS 25K, ALNS 50K
C11	7	682.12	ALNS 25K, ALNS 50K	(10)679.38	BR
C12	10	534.24	BR, BATA, LBTA, ALNS 25K, ALNS 50K, ORTR	534.24	BR, BATA, LBTA ALNS 25K, ALNS 50K, ORTR
C13	7	(11)909.80	ALNS 25K, ALNS 50K	(12)896.50	ORTR
C14	10	(11)591.87	ALNS 25K, ALNS 50K, ORTR	(12)469.3	TSR
F11	4	175.0	TSR	175.0	TSR
F12	7	769.66	ORTR	769.66	ORTR
Total	147	(156)10233.33		(165)9943.90	

	Number of times an algorithm generated the minimum number of vehicles		Number of times an algorithm generated the the minimum distance	
	ALNS 50K	9	TSR	5
	ALNS 25K	7	ORTR	4
	ORTR	5	BR*	3
	TSF	3	LBTA*	3
	BATA*	2	TSF	3
	TSR	2	ALNS 25K	3
	BR*	1	ALNS 50K	3
	LBTA*	1	BATA*	2
	CFRS*	0	CFRS*	0
	TSAK	0	TSAK	0
	TSAN	0	TSAN	0

* algorithm applied to seven problems

Table 6.5: Aggregate statistics on OVRP algorithms.

	CFRS	TSAK	TSAN	BR	LBTA	TSF	TSR	ALNS 25K	ALNS 50K	ORTR
Comparison on seven problems										
Vehicles	68	68	68	75	75	68	68	68	68	68
Distance	6192	4545	4701	4445	4433	4557	4480	4472	4470	4498
Time (s)	7	3796	193	406	414	5912	3124	5890	8790	741
Comparison on sixteen problems										
Vehicles		156	159			162	162	156	156	159
Distance		10406	10776			10309	10123	10199	10194	10191
Time (s)		9230	405			13383	6347	13350	22200	1756

Table 6.6: Computational results for ORTR on eight new test problems.

Problem	n	C	K_{min}	Estimated Solution	Li et al. ORTR	Li et al. Time (s)	Percent Improvement
O1	200	900	5	6151.77	6018.52	365.3	2.17
O2	240	550	9	(10)4785.75	4584.55	439.6	4.20
O3	280	900	7	7833.15	7732.85	492.8	1.28
O4	320	700	10	7338.51	7291.89	573.6	0.64
O5	360	900	8	(9)9303.85	9197.61	766.5	1.14
O6	400	900	9	(10)9924.81	9803.80	977.2	1.22
O7	440	900	10	(11)10507.13	10374.97	935.4	1.26
O8	480	1000	10	12513.11	12429.56	1126.8	0.67
Total Vehicles			68	72	68		

C — Vehicle capacity

( ) — Number of vehicles required if different from K_{min}

Percent Improvement — $100\frac{\text{(Estimated solution distance - ORTR distance)}}{\text{Estimated solution distance}}$

6.4 Conclusions

One of the first descriptions of the OVRP appeared in the operations research literature nearly 25 years ago, but the problem remained dormant computationally until recently. Over the last five years or so, a variety of heuristics including several that use tabu search and deterministic annealing were developed to solve 16 benchmark problems and we have summarized the details of these heuristics. We presented a new solution procedure (ORTR) that is based

on the record-to-record travel algorithm. We compared the results of 11 algorithms that solve the OVRP and found that the adaptive large neighborhood search procedures (ALNS 25K, ALNS 50K) performed very well but were very slow, and that the record-to-record travel procedure (ORTR) performed nearly as well but was fast. Procedures based on tabu search (TSAK, TSAN, TSF, TSR) were very competitive. Finally, we applied ORTR to eight new large-scale OVRPs and found that, in several minutes, it generated visually appealing solutions that were better than the estimated solutions.

6.5 OVRP generator

$(x(i), y(i))$ is the coordinate of customer i, where $i = 0$ is the depot
$q(i)$ is the demand of customer i
A and B are parameters that determine the number of customers n, where $n = A \times B$
All data recorded to four decimal places

```
begin
      ω = 0
      x(ω) = 0, y(ω) = 0, q(ω) = 0
      for k := 1 to B do
      begin
   γ = 30k
   for i := 1 to A do
   begin
         ω = ω + 1
         x(ω) = γ cos [2(i − 1)π/A]
         y(ω) = γ sin [2(i − 1)π/A]
         if mod(i ,4) = 2 or 3
         then q(ω) = 30
         else q(ω) = 10
         end
   end
      end
end
```

Problem	A	B	n	Vehicle Capacity
O1	20	10	200	900
O2	40	6	240	550
O3	28	10	280	900
O4	40	8	320	700
O5	36	10	360	900
O6	40	10	400	900
O7	44	10	440	900
O8	40	12	480	1000

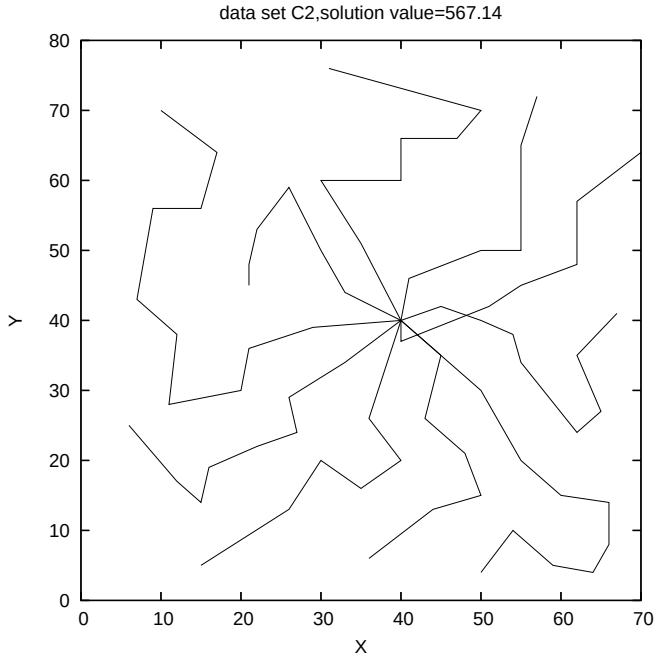


Figure 6.1: Problem C2.

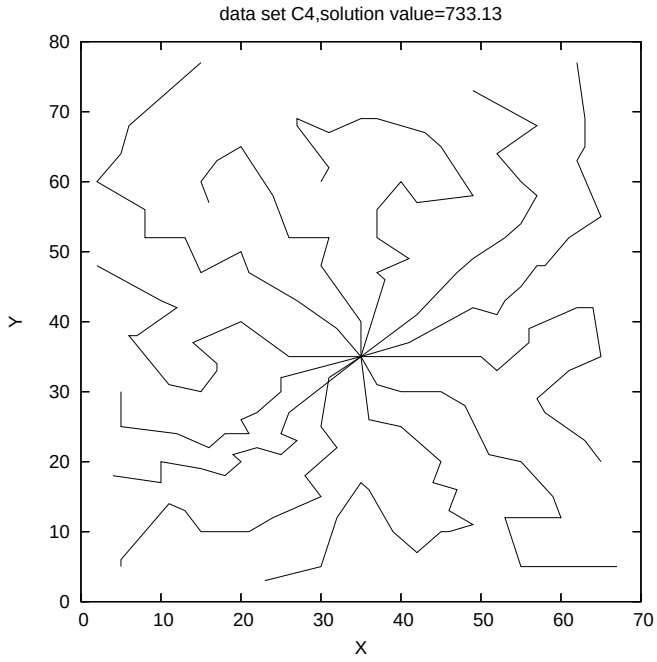


Figure 6.2: Problem C4.

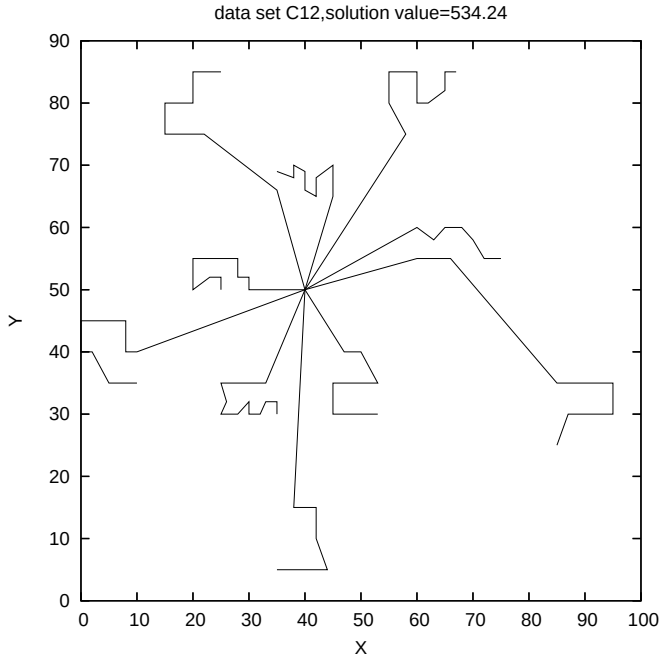


Figure 6.3: Problem C12.

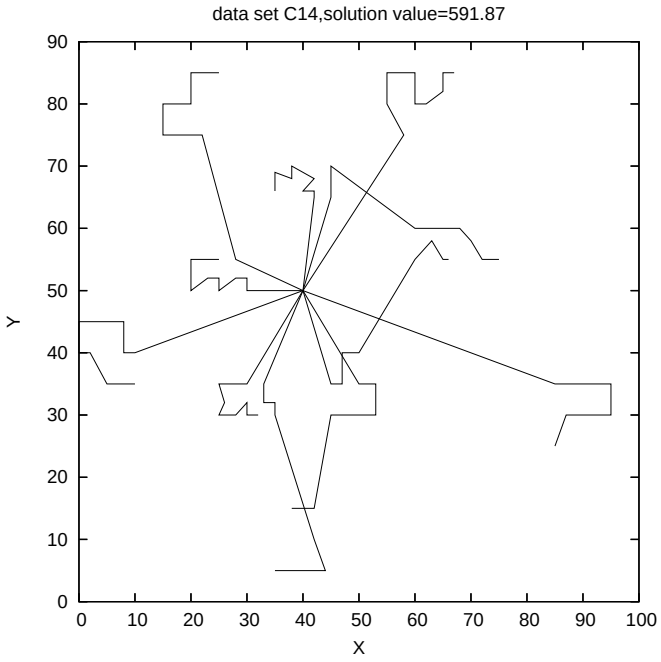


Figure 6.4: Problem C14.

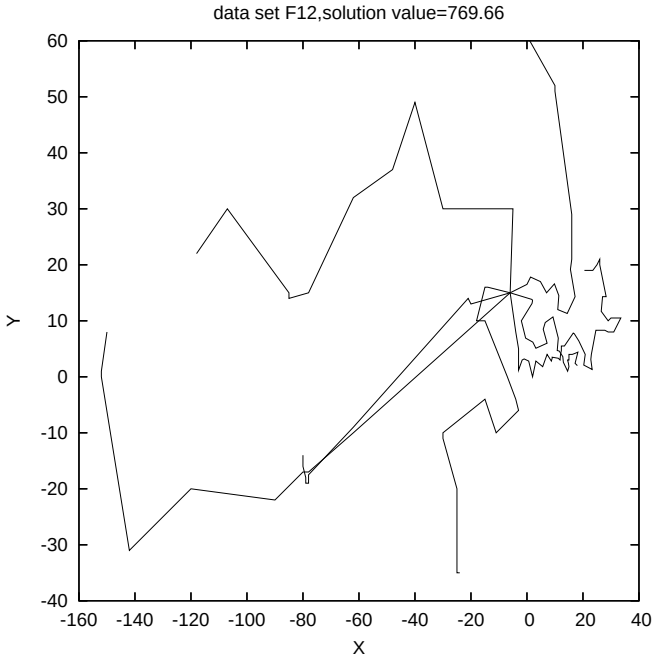


Figure 6.5: Problem F12.

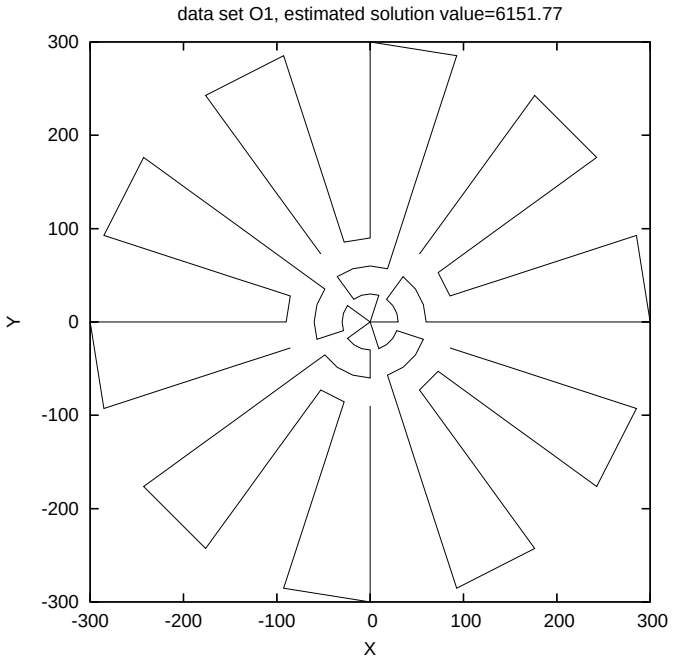


Figure 6.6: Estimated solution for O1.

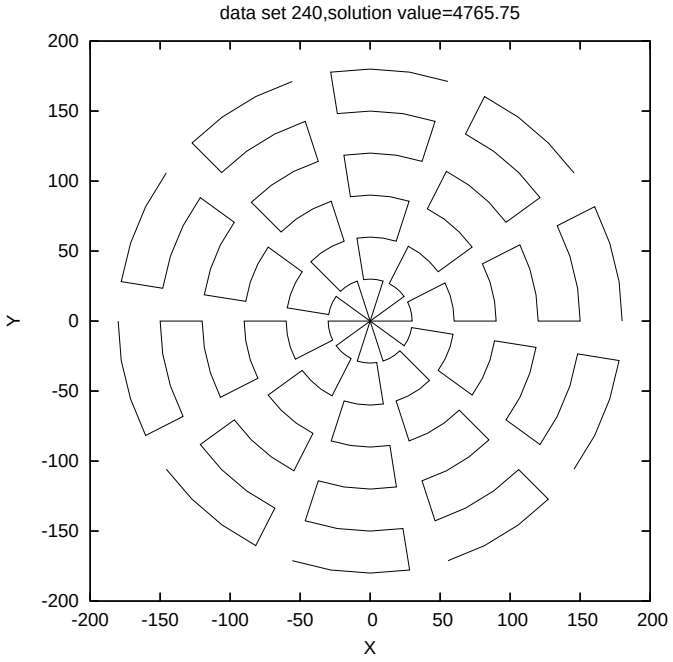


Figure 6.7: Estimated solution for O2.

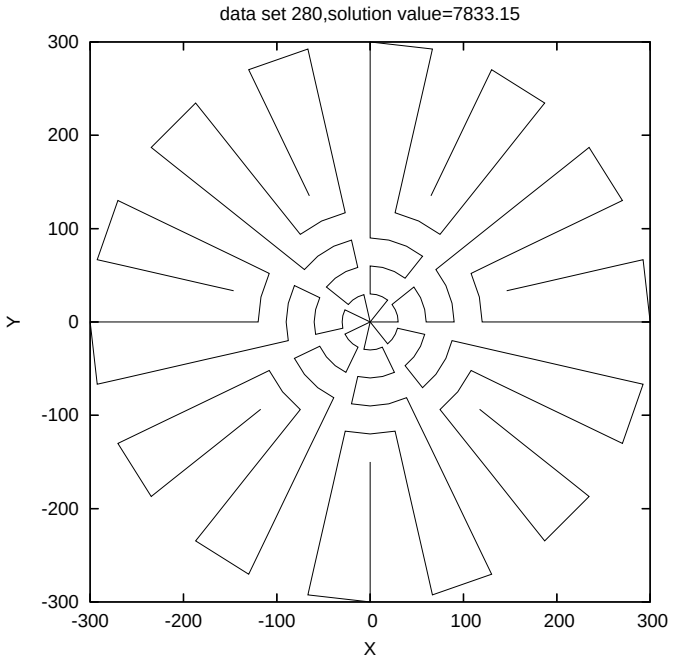


Figure 6.8: Estimated solution for O3.

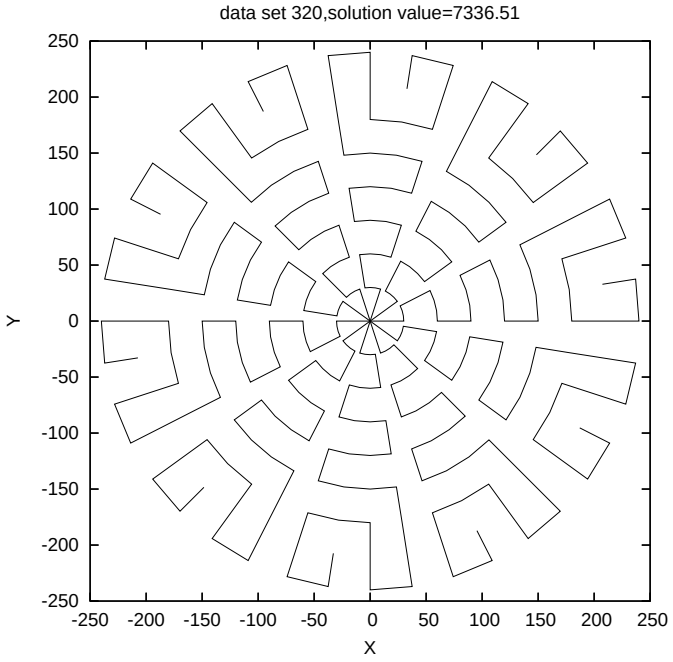


Figure 6.9: Estimated solution for O4.

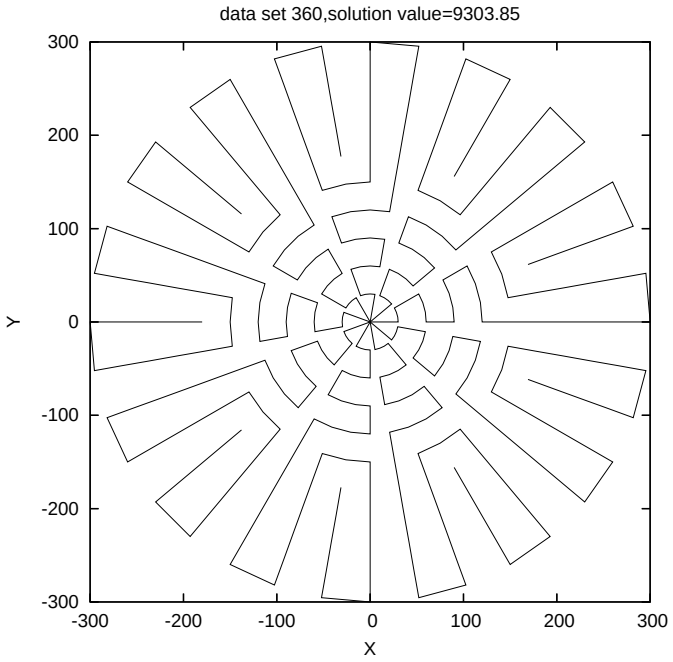


Figure 6.10: Estimated solution for O5.

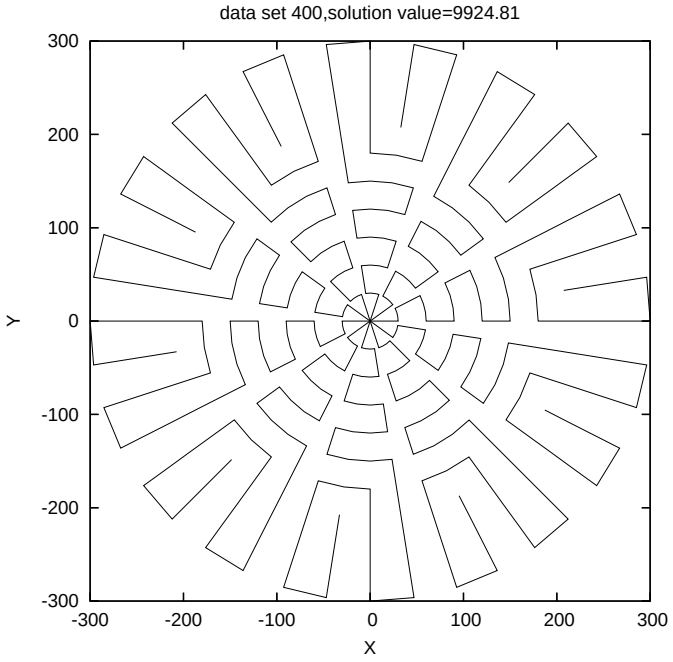


Figure 6.11: Estimated solution for O6.

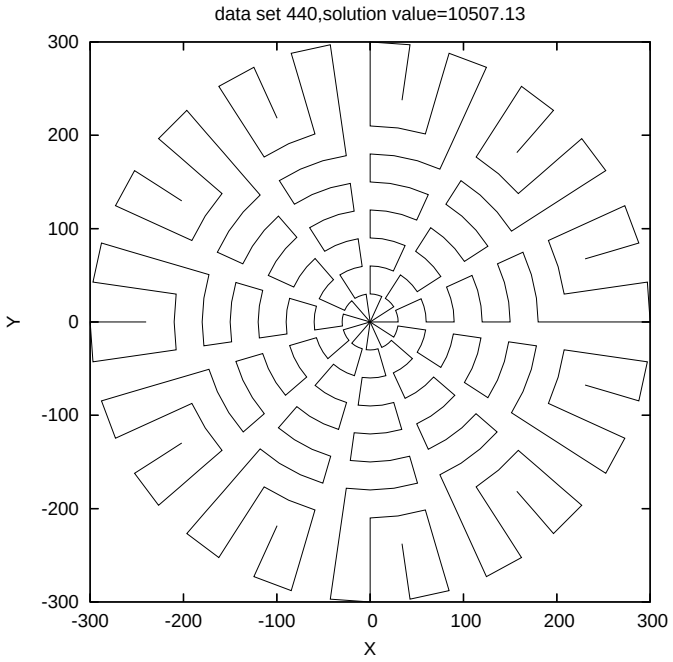


Figure 6.12: Estimated solution for O7.

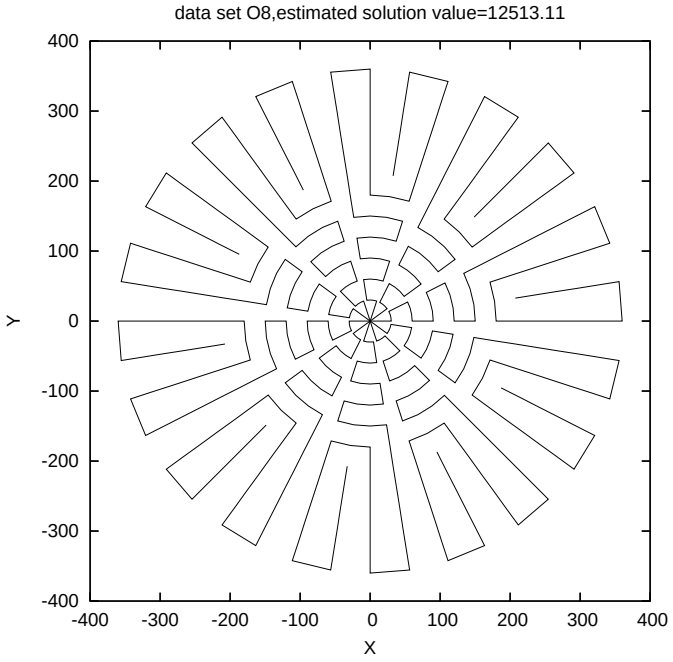


Figure 6.13: Estimated solution for O8.

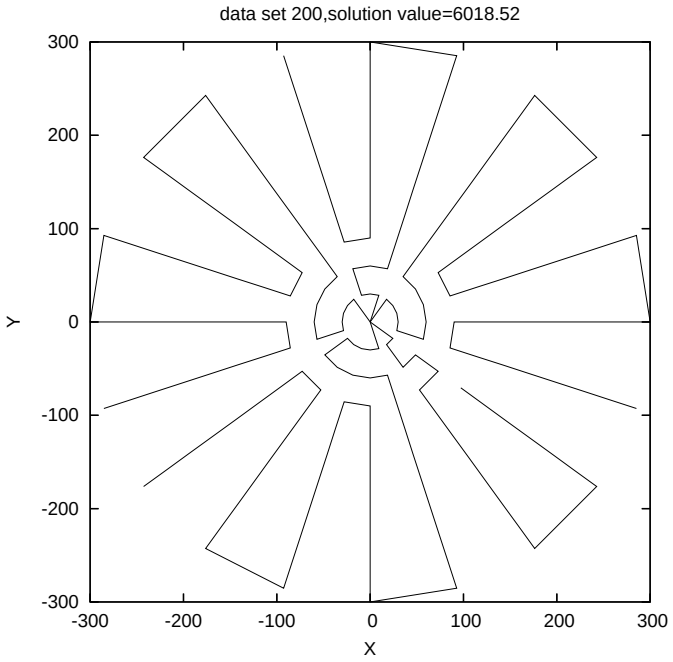


Figure 6.14: OVRP solution for O1.

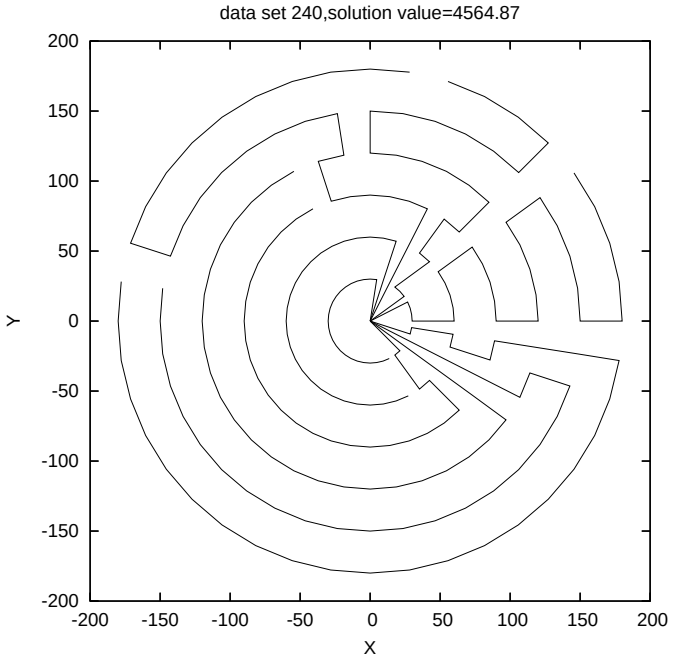


Figure 6.15: OVRP solution for O2.

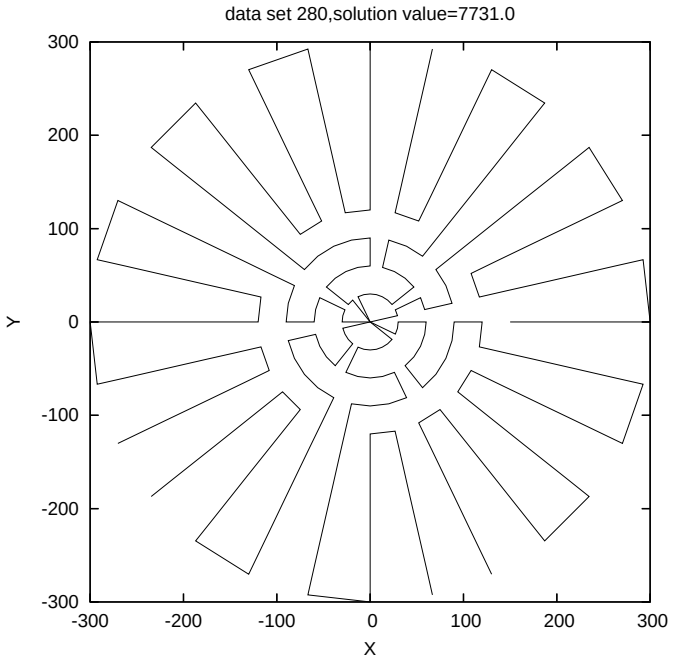


Figure 6.16: OVRP solution for O3

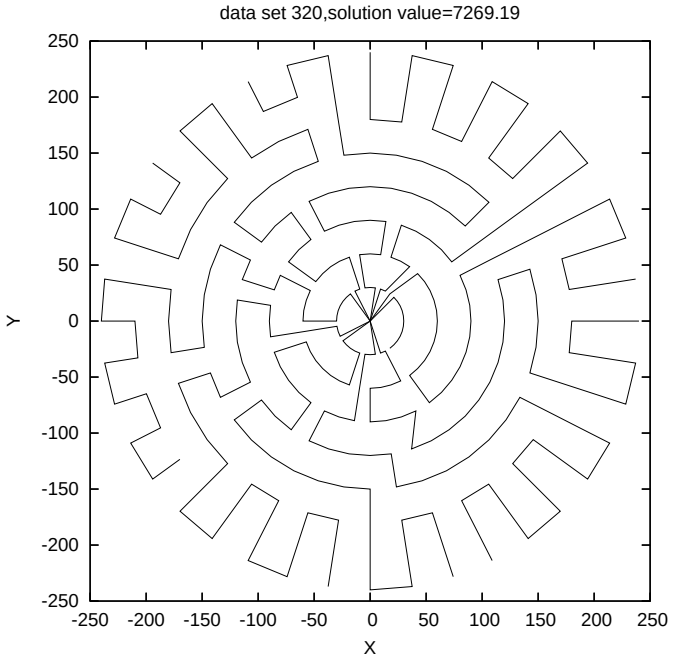


Figure 6.17: OVRP solution for O4.

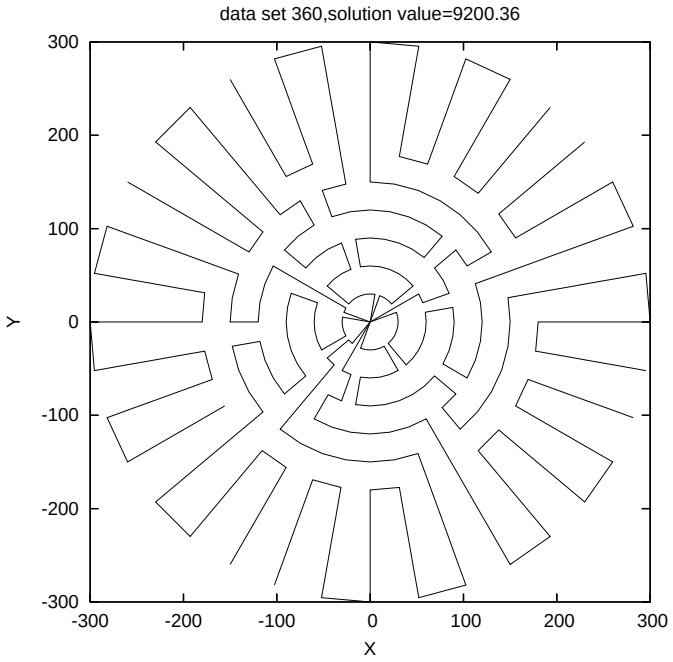


Figure 6.18: OVRP solution for O5.

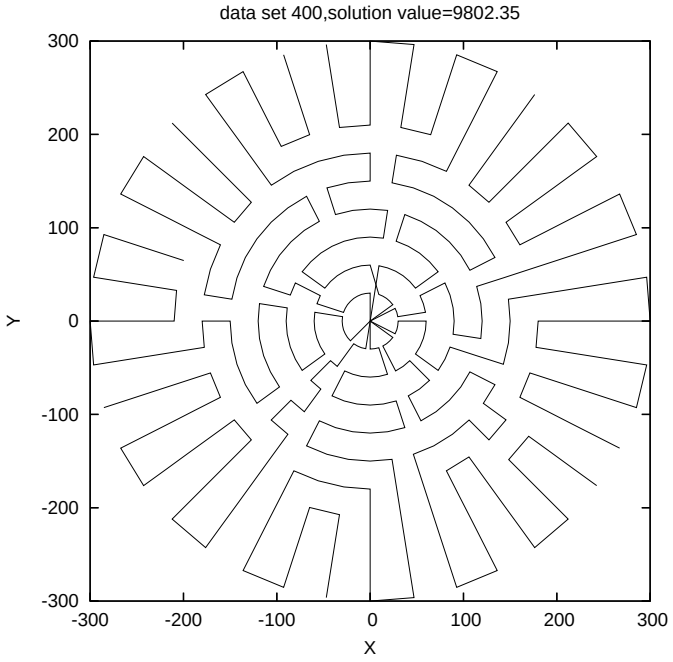


Figure 6.19: OVRP solution for O6.

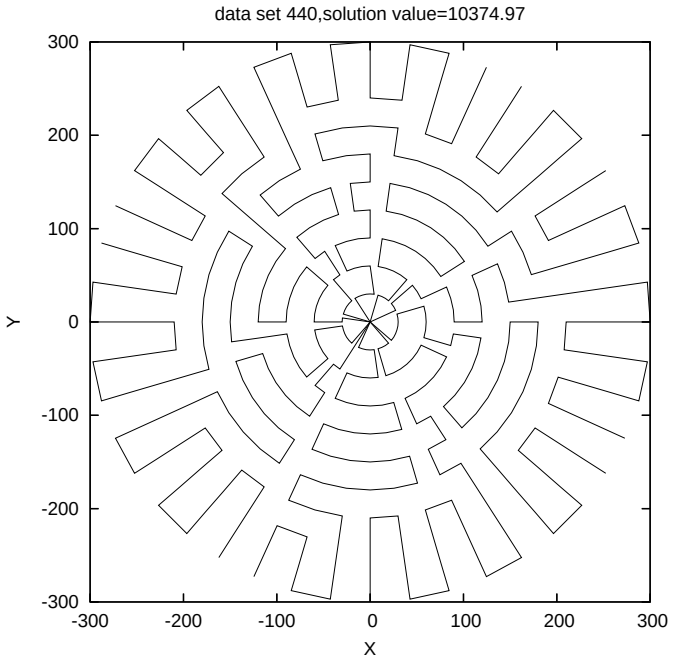


Figure 6.20: OVRP solution for O7.

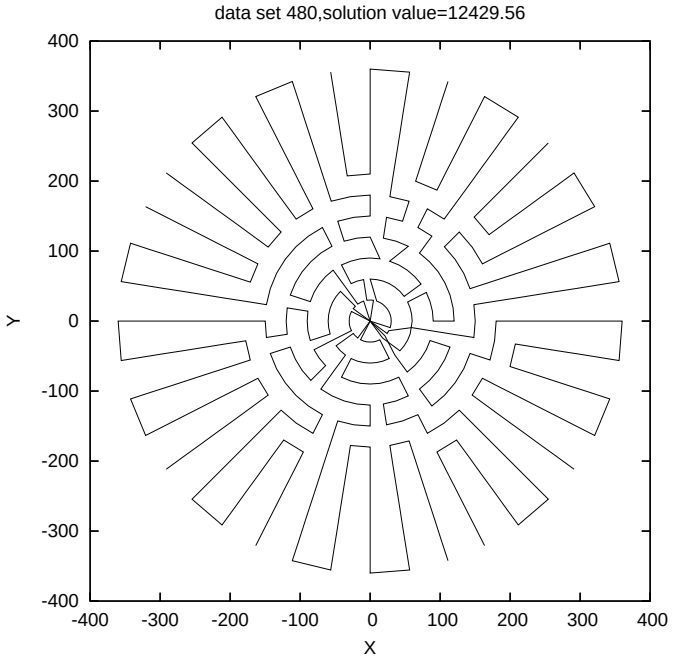


Figure 6.21: OVRP solution for O8.

Chapter 7

The Landfill Routing Problem

In this chapter, we consider a real-world waste collection problem. We describe the problem, discuss the underlying assumptions, sketch a solution procedure based on record-to-record travel, and present computational results for a problem extracted from an actual data set. We point out that much of the work in this chapter is preliminary and at the formulation stage.

7.1 Problem description

We consider a waste collection company that collects trash from commercial customers on a daily basis and dumps the trash into several landfills using a homogeneous fleet of vehicles. The objective is to minimize the total distance traveled by the fleet to service all of the customers. We call this the landfill routing problem and denote it by LRP.

In Figure 7.1, we show the locations of customers and landfills that were extracted from an actual data set provided by Levy [27]. We note this is a *node-routing* problem (commercial customers are the nodes and vehicles must service only the nodes) in contrast to an *arc-routing* problem encountered in residential trash collection where all residences on a street must be serviced by a vehicle.

The waste collection application of vehicle routing has been studied in the operations research literature for more than 30 years, starting with the classic paper by Beltrami and Bodin [4]. Recently, Amponsah and Salhi [1] modeled the garbage collection problem as an arc-routing problem. They assume the arcs covered by each vehicle are already determined. The objective function takes into account deadheading and the environmental impact of garbage in warm weather. A look-ahead strategy was used during the route-construction phase and a refinement scheme was proposed to improve the solution.

Sahoo et al. [34] studied the routing of vehicles for solid-waste collection for Waste Management, Inc (WM). They developed a geographical information system called WasteRoute for routing vehicles to service commercial and residential customers. By using WasteRoute, WM significantly reduced the number of collection routes and achieved savings of $18 million in 2003 and $44 million in 2004.

7.2 Assumptions

Each vehicle has a common capacity C and a route-length constraint L (or equivalently a working time). Each customer has a demand d and a service time t. Each customer must be fully serviced by one vehicle (i.e., there is no splitting of demand).

Each vehicle starts from the depot and visits each customer exactly once. Each vehicle must dump all of the trash it collects at a landfill before it is out of capacity. Here we assume all landfills are homogeneous. In future work, we may consider the fact that different landfills charge different fees for a unit size of trash. Also, if the vehicle company owns some landfills, then company vehicles

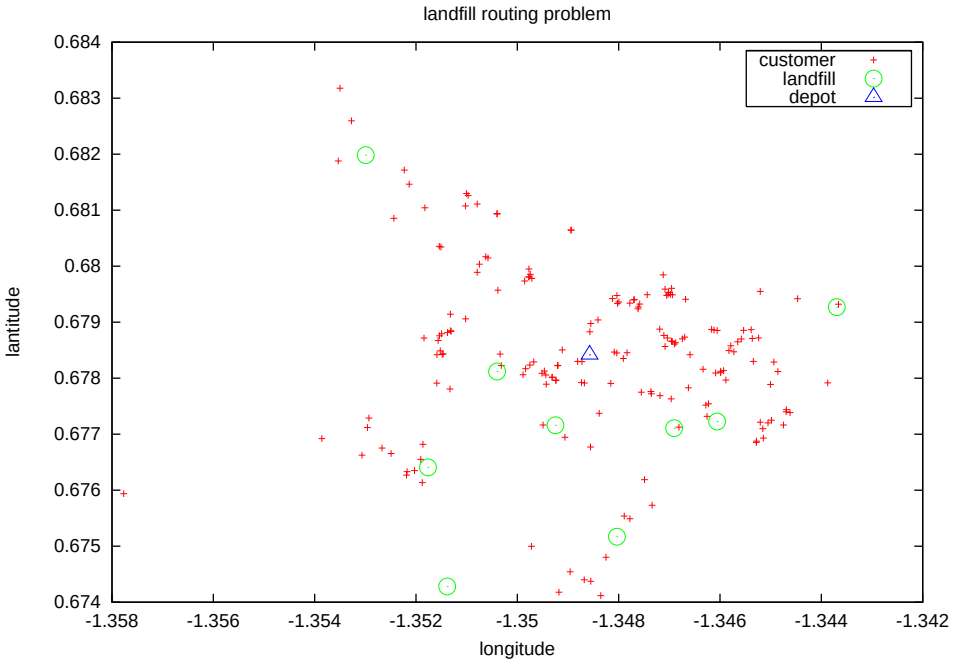


Figure 7.1: 178 customers and 9 landfills.

will prefer to dump the trash at those landfills if they are not too far away from the last stop of the current route. We assume that a landfill has unlimited input capacity. Finally, we assume a landfill is open 24 hours so that a vehicle can visit any landfill at any time. This assumption is reasonable because a route-length constraint will ensure that a vehicle will not visit a landfill too late in the day. Our objective function is to minimize the total distance traveled by the fleet of vehicles to service all the customers.

7.3 List of candidate landfills

For each customer, we build a list of candidate landfills at which we can dump a customer's trash. First, we sort the distance between the customer i and all the landfills ℓ_k. Second, we compute the ratio $r_k = \frac{d_{i1}}{d_{ik}}$ and we only keep the

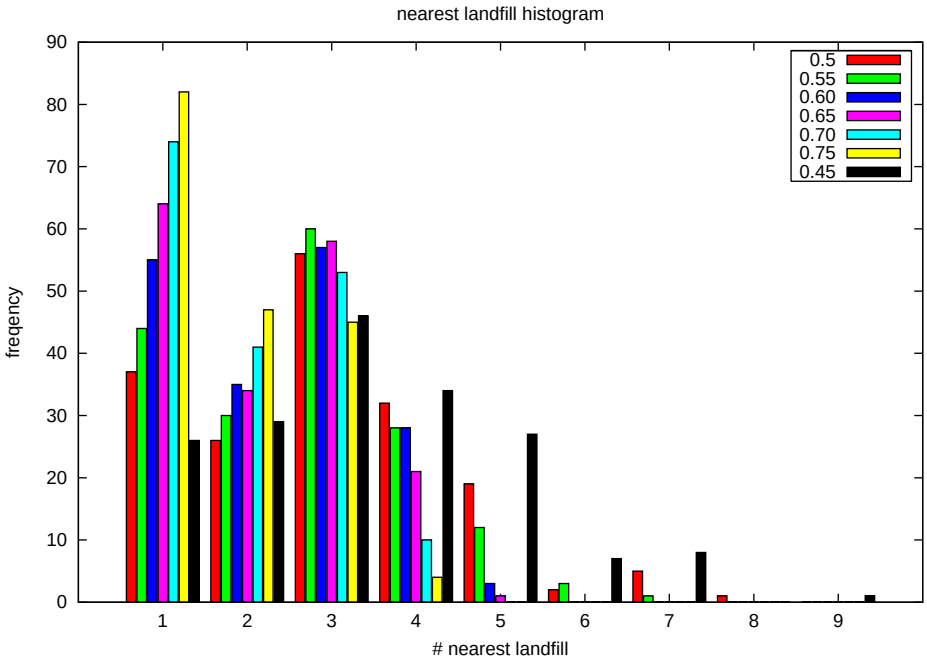


Figure 7.2: Histogram of candidate list.

landfills whose ratio $r_k >$ threshold, where d_{i1} is the distance to the nearest landfill. The rationale behind this is we want to consider only a small set of landfills for each customer. In Figure 7.2, we show the histogram for different values of the threshold. We want to choose a such a value that on average each customer does not have too many or too few candidate landfills. It appears that 0.6 is a good value to use.

7.4 Initial solution

We need to generate a good initial feasible quickly. Here are two ideas.

(1) Cluster. We first assign customers to the nearest landfill, then build routes individually. This seems easy to implement but we are not sure the solution is of high quality.

(2) Savings. First, we assign each customer to one vehicle, then compute the savings by merging two customers into one route. We sort these savings and apply ideas that are similar to those in Clarke and Wright's algorithm. Here are some details. Let 0 denote the depot, 1 and 2 are two customers, and x is the landfill. The cost of serving 1 and 2 is

$$c_1 = d_{01} + d_{1x} + d_{x0}$$

$$c_2 = d_{02} + d_{2x} + d_{x0}.$$

The cost of serving 1 and 2 in one route is

$$c_{12} = d_{01} + d_{12} + d_{2x} + d_{x0},$$

so the savings is

$$s_{12} = d_{02} + d_{1x} - d_{12} + d_{x0}.$$

The resulting tour is

$$0 - 1 - 2 - x - 0.$$

If we consider the reverse order

$$0 - 2 - 1 - x - 0,$$

the another possible savings is

$$s_{21} = d_{01} + d_{2x} - d_{12} + d_{x0}.$$

So the savings is not symmetric as it is in the traditional VRP.

7.5 Last customer in the tour

We define two terms. A *trip* consists of a sequence of nodes starting from one landfill and ending at another landfill. A tour consists of one or more trips in which two consecutive trips share exactly one landfill and the starting landfill in the first trip is the depot (if we treat the depot as a special landfill).

The last customer i in a tour is very special in the sense that the landfill following it is uniquely determined. It is the landfill that minimizes $d_{i,k} + f_k$ for all landfills k, where $d_{i,k}$ is the distance between customer i and landfill k, and f_k is the distance between landfill k and the depot. In general, this landfill is not the same as the nearest landfill to customer i. In Figure 7.3, l is the nearest landfill to customer i, where i is the last customer in the trip. It is clear from the figure that we would choose to dump trash at landfill k instead of landfill l since the total distance traveled in the solid lines is shorter than the distance traveled in the dashed lines.

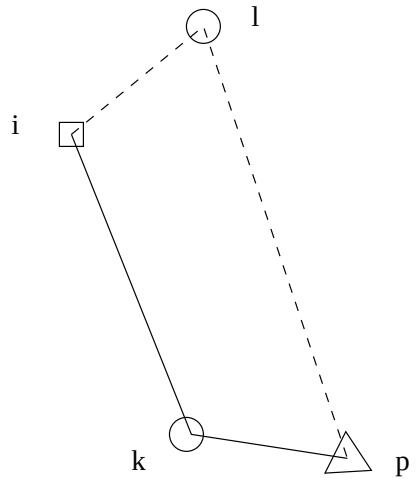


Figure 7.3: Nearest landfill is not always preferable for the last customer in a tour.

7.6 Savings heuristic for the initial solution

7.6.1 Saving from two customers

We use ideas that are similar to Clarke and Wright's savings method. First, we only consider linking two customers i and j and neither of them are interior to the tour. Let D denote the depot, x the customer, and o the landfill. In the following diagram we link j to i and the result is one giant tour.

Let $t_1 = Dix \dots xo \dots oD$ and $t_2 = Dx \dots o \dots joD$. If we join t_1 and t_2, the resulting tour is $t_1 \cup t_2 = Dx \dots o \dots j - ix \dots xo \dots oD$.

The savings s_{ij} associated with above linking is $s_{ij} = d_i + d^j - d_{ij}$, where d_i is the distance from the depot to customer i and d^j is the distance from customer j back to depot via a landfill uniquely determined by j.

The above savings can be viewed as linking two customers directly. We can also link the last customer j in one trip with the first customer i in the second trip via a landfill k, and we denote it by s_{ij}^k.

Let $t_1 = Dix \dots xo.....oD$ and $t_2 = Dx \dots o \dots joD$. The resulting tour is $t_1 \cup t_2 = Dx \dots o \dots j - k - ix \dots xo \dots oD$. The savings associated with above linking is

$$s_{ij}^k = d_i + d^j - d_{ik} - d_{kj},$$

where d_i is the distance from the depot to customer i, and d^j is the distance from the landfill uniquely determined by j back to the depot, and d_{ik} is the distance between customer i and the landfill k, and similarly for d_{kj}. We notice that the savings s_{ij} is not symmetric, i.e., $s_{ij} \neq s_{ji}$.

Table 7.1: Comparison of different implementations of the savings method.

	Straightforward implementation	Candidate landfill	Grids (8x8)
Saving list size	174798	41941	29621
Objective value	30490.1629	30490.8989	30596.9581
Running time (sec)	1.38	0.71	0.62

7.6.2 Reducing the size of the savings list

In the straightforward implementation of savings algorithm, we first compute savings s_{ij} for all pair of nodes (i,j) and s_{ij}^k for all triples (i,j,k). So, if n is the number of nodes and m is the number of landfills, the total number of savings is on the order of $O(n^2m)$, which is a big number when n is large. In Golden et.al [20], the authors first superimpose a grid on the graph, and only considers savings for two nodes lying in the adjacent grids. By doing this, lots of uninteresting savings will be removed at the cost of two extra parameters, the horizontal grid size and vertical grid size.

We now propose another way of reducing the size of the savings list. We consider savings s_{ij} when i and j have at least one common candidate landfill, and s_{ij}^k when k belongs to both i's candidate landfills and j's candidate landfills. The computational experiment shows this works very well. In Table 7.1, we show the size of the savings list for different implementations as well as the running time. In Figure 7.4, we show the solution given by the savings method. In Figure 7.5, we show the solution after applying two-opt to individual trips.

7.7 Global two-opt does not work

When we say global two-opt, we mean applying a two-opt procedure on a giant tour from the solution. For this problem, the global two-opt does not work

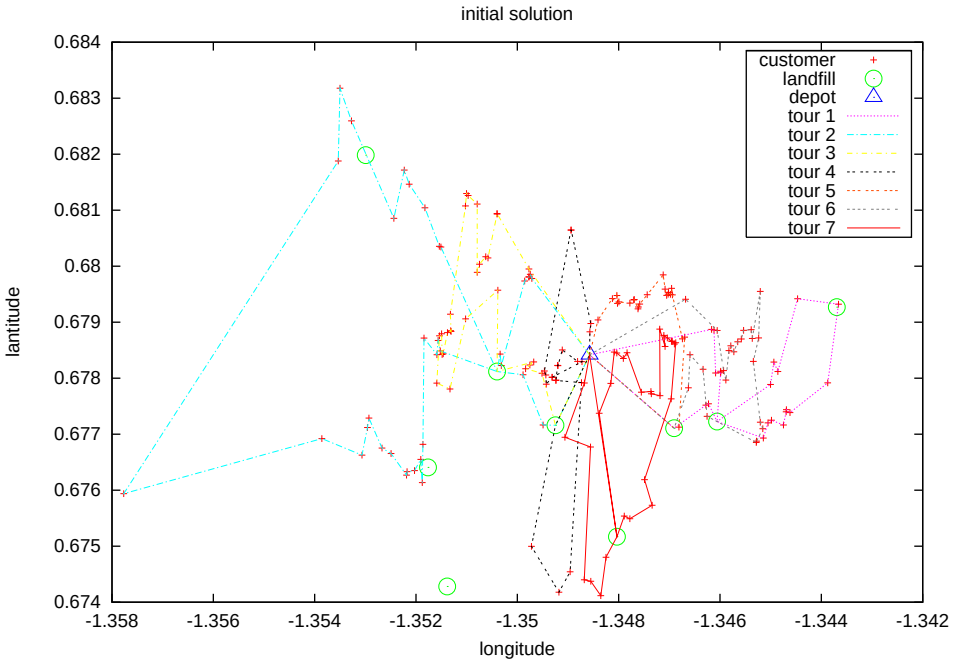


Figure 7.4: Initial solution generated by the savings method (solution value is 30491.06).

because the distance matrix is asymmetric. The distance from the depot to a customer and from a customer to the depot is different due to the fact that we must dump all trash at a landfill before we can return to the depot. So, any procedure that alters the orientation of a subtour will not work because the savings is more complicated.

7.8 Record-to-record travel

In record-to-record travel, we use two-opt with record-to-record travel within a tour, one-point move with record-to-record travel, two-points exchange move with record-to-record travel. In Figure 7.6, we give the solution obtained by record-to-record-travel with spherical distances. In Figure 7.7 we give the solution using Euclidean distances.

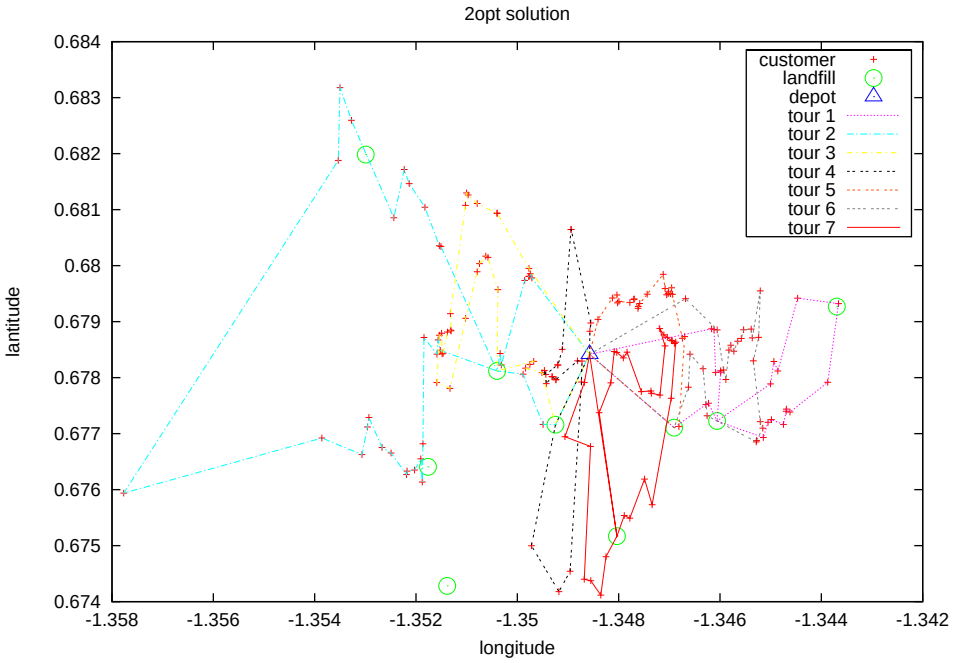


Figure 7.5: Two-opt solution (solution value is 30483.56).

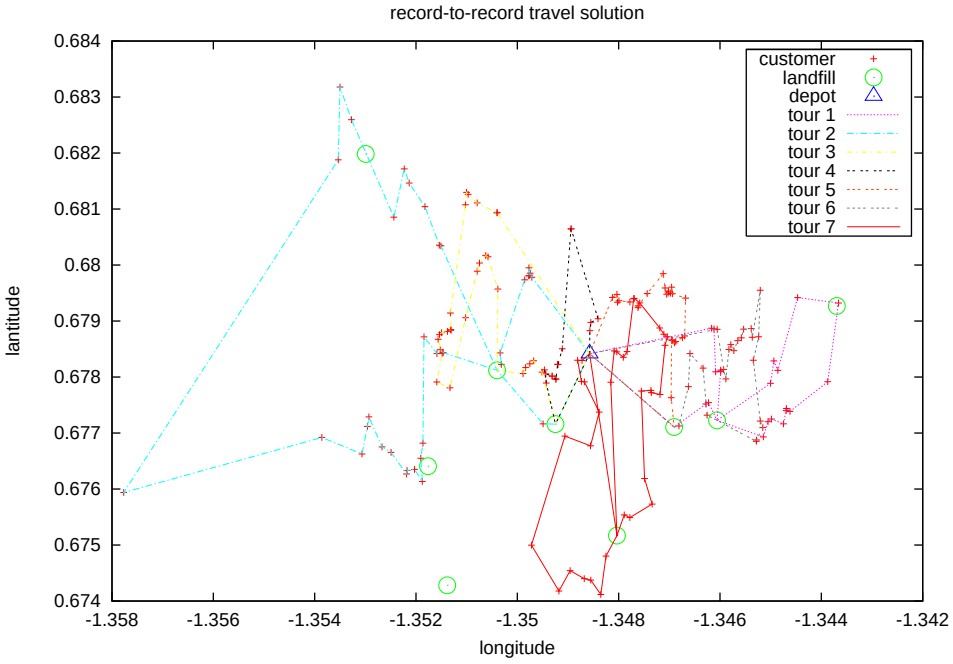


Figure 7.6: Record-to-record travel solution (solution value 30473.16, spherical distances).

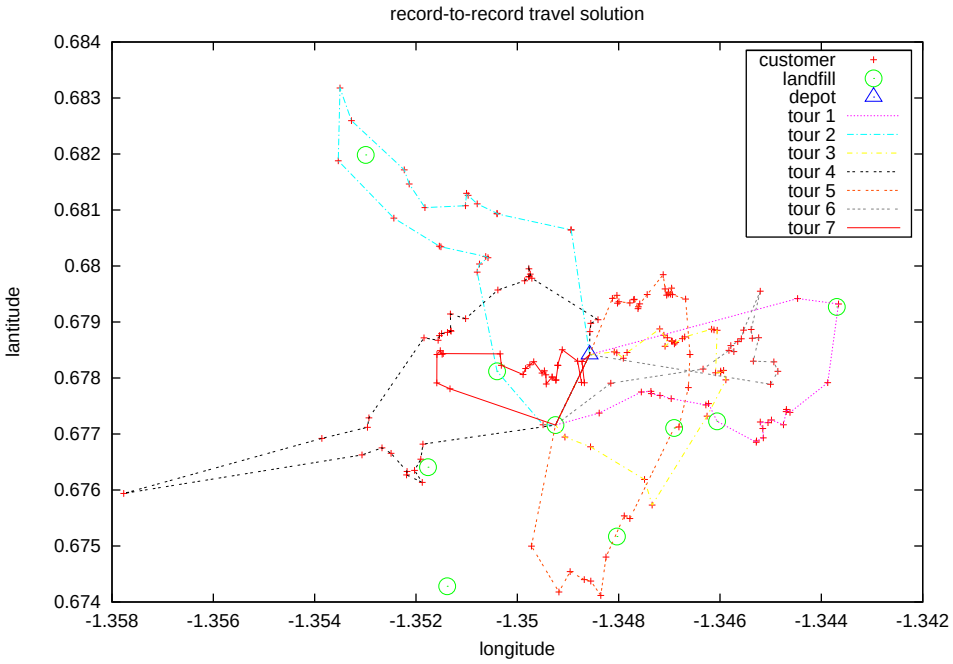


Figure 7.7: Record-to-record travel solution (solution value 30107.58, Euclidean distances).

Chapter 8

Conclusions

In this dissertation, we modeled and solved six variants of the vehicle routing problem – large-scale vehicle routing problem (LSVRP), time dependent traveling salesman problem (TDTSP), noisy traveling salesman problem (NTSP), open vehicle routing problem (OVRP), heterogeneous vehicle routing problem (HVRP), and landfill routing problem (LRP).

We developed a single heuristic – record-to-record travel (RTR) – to solve five of the six variants (the exception is the NTSP, where we developed a new solution approach based on the quad tree). RTR is a variant of deterministic annealing. It was fast and required few parameters. RTR was easy to implement, easy to extend, and produced very good computational results.

We generated new large-scale data sets for the LSVRP, NTSP, HVRP, and OVRP. Our data sets ranged from 560 nodes to 1200 nodes. We used the geometric symmetry in problem instances to generate estimated solutions for the LSVRP and the OVRP. Our hope is that these data sets will motivate the operations research community to develop new, more powerful heuristics for different variants of the standard vehicle routing problem.

We provided a problem generator for the LSVRP, HVRP, and OVRP. Our generator can be used by researchers to generate new problem instances.

Researchers can then compare the results of different heuristics on these new instances.

During our research effort, we incorporated simplicity and flexibility into the design of our heuristics. Simplicity made our heuristics easy to understand and easy to implement. Simplicity also made it easier to incorporate additional features. Flexibility enabled the heuristic to easily accommodate side constraints, so a set of related problems could be solved. The flexibility of RTR enabled us to solve the LSVRP, TDTSP, OVRP, HVRP, and LRP. Even though computers are becoming more and more powerful, simplicity and flexibility should always be kept in mind when designing heuristics for difficult combinatorial optimization problems.

Appendix A

Large-scale VRP & Time dependent TSP Code

A.1 Graph.java

```
/**
   a graph is a nxn double matrix
*/
public class Graph implements Cloneable{
    protected int N;
    public double dist[][];
    final static double INFTY = 1e15;
    public Graph() {}
    public Graph(int n) {
  N=n;
  dist = new double[n][n];
  for(int i=0;i<n;i++) {
      connect(i,i,0.0);
      for(int j=i+1;j<n;j++) {
    connect(i,j,INFTY);
    connect(j,i,INFTY);
      }
  }
    }

    public Graph(double [][] d)
    {
  this(d.length);
  for(int i=0;i<N;i++)
      for(int j=0;j<N;j++)
    dist[i][j] = d[i][j];
    }
```

```
    public  void connect(int i,int j,double d) { dist[i][j]=d;
   }
    public final int size() { return N; }
    final double distance(int i,int j) {
  return dist[i][j];
    }

    public Object clone()
    {
  Graph g = null;
  try {
      g = (Graph)super.clone();
  }
  catch(CloneNotSupportedException e){}
  g.dist = new double[N][N];
  for(int i=0;i<N;i++)
      for(int j=0;j<N;j++)
    g.dist[i][j] = dist[i][j];
  return g;
    }

    public String toString()
    {
  StringBuffer sb = new StringBuffer();
  for(int i=0;i<N;i++) {
      for(int j=0;j<N;j++)
    sb.append(dist[i][j]+" ");
      sb.append("\n");
  }
  return sb.toString();
    }

}
```

A.2 VRPGraph.java

```
//package vrp;

import java.awt.Point;
import java.util.Random;
import java.io.*;
import java.util.*;

public class VRPGraph extends Graph {
    final double sqr(double x) { return x*x;}
```

```
 protected VRPNode [] nodes;
 /**
    capacity and distance constraints
 */
 public int capacity;
 public double disCons;

 public int NBList[][];
 public int NBCount[];

 final int MAX_NB = 25;
 public int numRoutes;
 public Route routes[];
 final int MAX_ROUTES=100;
 final double EPS=1e-10;
 boolean ovrp;
 public void setOVRP(boolean flag) { ovrp = flag; }
 int Load[];
 /**
    for distance constraints
 */
 private double length[];
 /**
    dummy depots and path
 */
 VRPNode depots[];
 VRPNode path[];

 public double cutoff;
 public static double beta ;

 public SimpleTabuList tabulist;
 ////////////////////////////////////
 public TrafficJamRegion tjr;
 public double jamFactor;

 public void setTrafficJamRegion(TrafficJamRegion t) { tjr =
 t; }
 public void setJamFactor(double f) { jamFactor = f; tjr.
setJamFactor(f); }
 final double START_TIME = 8D;
 final double END_TIME = 17D;
 final double TOTAL_DISTANCE = 118.52;
 final double VELOCITY = TOTAL_DISTANCE/(END_TIME -
START_TIME);
```

```
  final double JAMTIME = 12D;
  public double getTimeStart() { return START_TIME; }
  public double getVelocity() { return VELOCITY; }
  public double getJamTime() { return JAMTIME; }
  public double getJamFactor() { return jamFactor; }

  // including the # of dummy depot in the graph
  public int psize(){
return size()+ numRoutes -1 ;
  }

  public boolean insideTrafficJamRegion(Edge e) {
int p = e.from>=size()?0:e.from;
int q = e.to>=size()?0:e.to;
return tjr.inside(nodes[p]) & tjr.inside(nodes[q]);
  }

  public boolean withinTrafficJamRegion(Edge e) {
int p = e.from>=size()?0:e.from;
int q = e.to>=size()?0:e.to;
return tjr.inside(nodes[p]) | tjr.inside(nodes[q]);
  }

  public boolean inside(int n) {
int p = n>=size()?0:n;
return tjr.inside(nodes[p]);
  }

  public double getPartialLength(int inside,int outside,
 double len) {
int p = inside>=size()?0:inside;
int q = outside>=size()?0:outside;
return tjr.getPartialLength(nodes[p],nodes[q],len);
  }

  public double getPartialLength(int from, int to,double len,
  double currentTime,double jamTime,double v) {
int p = from>=size()?0:from;
int q = to>=size()?0:to;
return tjr.getPartialLength(nodes[p],nodes[q],len,currentTime
 ,jamTime,v);
  }

  ////////////////////////////////
  public VRPGraph(VRPNode [] n,int cap, double dist) {
```

```
super(n.length);
capacity = cap;
disCons = dist;
nodes = n;
NBList = new int[N][MAX_NB];
NBCount = new int[N];
Load = new int[N];
length = new double[N];
for(int i=0;i<N;i++)
    for(int j=i+1;j<N;j++) {
  double d = Math.sqrt(sqr(n[i].x-n[j].x)+sqr(n[i].y-n[j].y))
 ;
  connect(i,j,d);
  connect(j,i,d);
    }
buildNBList();
routes = new Route[MAX_ROUTES];
for(int i=0;i<MAX_ROUTES;i++) routes[i] = new Route();
  }

  public int getCapacity() { return capacity; }
  public double getDisConstraint() { return disCons; }
  public String toString() {
return new String("The graph has " + N +" nodes and with
 capacity "+
  capacity + ",distance constraint "+ disCons + "\n"+tjr + ",
 "+jamFactor);
  }

  public void dump() {
for(int i=0; i<size(); i++)
    System.out.println(nodes[i]);
  }

  public double getDistance(int C1,int C2) {
int n1 = C1>=N?0:C1;
int n2 = C2>=N?0:C2;
if(n1==n2) return 0;
else return dist[n1][n2];
  }

  private void buildNBList() {
double t1[] = new double[N];
double t2[] = new double[N];
for(int i=0; i<N; i++) {
```

```
    for(int k=0; k<N;k++) {
  t1[k] = distance(i,k);
  t2[k] = t1[k];
    }
    Arrays.sort(t1);
    for(int j=0; j<MAX_NB;j++) {
  for(int k=0;k<N;k++) {
      if(t2[k]==t1[j+1]) {
    NBList[i][j] = nodes[k].getID();
    t2[k]=0.0;
    break;
      }
  }
    }
}


TrimNBList();
  }

  private void TrimNBList()
  {
// start from 1 b/c 0 is the depot and we want to keep all
 the link from
// the depot to the nodes.
for(int i=0;i<N;i++)
    if(cutoff<distance(i,NBList[i][MAX_NB-1]))
  cutoff = distance(i,NBList[i][MAX_NB-1]);
cutoff *= beta;
//System.out.println("cutoff="+cutoff);
int remain = 0;
//NBCount = new int[N];
//NBCount[0] = MAX_NB;
for(int i=0;i<N;i++)
    for(int j=0;j<MAX_NB;j++)
  if(distance(i,NBList[i][j])<cutoff)
      NBCount[i]++;

int minNB=MAX_NB,maxNB=0;
for(int i=0;i<N;i++){
    if(minNB>NBCount[i]) minNB = NBCount[i];
    if(maxNB<NBCount[i]) maxNB = NBCount[i];
}
for(int i=0;i<N;i++) remain+=NBCount[i];
```

```
System.out.println("orignial NBList size ="+MAX_NB*N);
System.out.println("after trimming NBlist size ="+ remain+",
 fraction="+remain/(double)(MAX_NB*N));
System.out.println("min = "+ minNB+",max="+maxNB);

  }

  private void updateTour(VRPNode n1,VRPNode n2,int load,
 double len) {
VRPNode now = n2;
VRPNode next, prev;
while( now.getID()!= 0 ) {
    next = now.getNext();
    Load[now.getID()] = load;
    length[now.getID()] = len;
    now = next;
}
now = n1;
while(now.getID() != 0 ){
    prev = now.getPrior();
    Load[now.getID()] = load;
    length[now.getID()] = len;
    now = prev;
}
  }

  //used in CWParallel to check if two nodes from
  //the saving list are already in the same route

  private boolean inTheSameRoute(VRPNode n1,VRPNode n2) {
if(n1.getID()==0 | n2.getID()==0) return true;
else {
    boolean reachable = false;
    for(int j=0; j<2; j++) {
  VRPNode now = n1;
  VRPNode next;
  while( (next=now.get(j)).getID() != 0 ) {
      if( next.getID() == n2.getID() ) {
    reachable = true;
    break;
      }
      else
    now = next;
  }
  if(reachable) break;
```

```
    }
    return reachable;
}
  }

  /**
     Change the orienation of a path which n belongs to
  */
  private void reversePath(VRPNode n) {
if(n.isDepot()) return;
else {
    VRPNode reverse,save;
    for(reverse = n; !reverse.isDepot(); reverse = save) {
  save = reverse.getNext();
  reverse.setNext(reverse.getPrior());
  reverse.setPrior(save);
    }
    for(reverse=n.getNext(); !reverse.isDepot(); reverse=save
 ) {
  save = reverse.getPrior();
  reverse.setPrior(reverse.getNext());
  reverse.setNext(save);
    }
}
  }

  public Path CWParallel(double lamda) {
class CWSaving implements Comparable{
    public double saving;
    public VRPNode i,j;
    public CWSaving(VRPNode n1,VRPNode n2,double s) {
  i = n1;
  j= n2;
  saving = s;
    }
    public int compareTo(Object o) {
  if (saving < ((CWSaving) o).saving)
        return -1;
  else {
      if (saving > ((CWSaving) o).saving)
        return 1;
      else
    return 0;
  }
    }
```

```
}

for(int i=1; i<N; i++) {
    nodes[i].setPrior(nodes[0]);
    nodes[i].setNext(nodes[0]);
    Load[i] = nodes[i].getDemand();
    length[i] = 2*dist[0][i];
}

int n = (N-1)*(N-2)/2;
CWSaving [] savings = new CWSaving[n];
// compute Clarke & Wright's saving and sort it in ascending
 order
for(int i=1; i<N-1; i++)
    for(int j=i+1; j<N; j++) {
  double s =distance(i,0)+distance(j,0)- lamda*distance(i,j);
  savings[(i-1)*N-(i-1)*(i+2)/2+j-i-1] = new CWSaving(nodes[i
 ],nodes[j],s);
    }
Arrays.sort(savings);
for(int i=0;i<n/2;i++) {
    CWSaving temp = savings[i];
    savings[i] = savings[n-i-1];
    savings[n-i-1] = temp;
}
for(int i=0; i<n; i++) {
    CWSaving top = savings[i];
    VRPNode n1 = top.i;
    VRPNode n2 = top.j;
    boolean feasible = true;
    if (n1.adjacentToDepot() & n2.adjacentToDepot()) {
  if(!inTheSameRoute(n1,n2)) {
      int load = Load[n1.getID()] + Load[n2.getID()] ;
      double dist = length[n1.getID()]+length[n2.getID()]
    +distance(n1.getID(),n2.getID())-distance(n1.getID(),0)-
 distance(n2.getID(),0);
      if ( (load <= capacity) & (dist<=disCons) ) {
    if (n1.getNext().isDepot()) {
        if(n2.getNext().isDepot())
      reversePath(n2);
        n1.setNext(n2);
        n2.setPrior(n1);
        updateTour(n1,n2,load,dist);
    }
```

```
    else {
        if(n2.getPrior().isDepot())
      reversePath(n2);
        n1.setPrior(n2);
        n2.setNext(n1);
        updateTour(n2,n1,load,dist);
    }
      }
  }
    }
}
buildRoutes();
buildPath();
Path p = new Path(this);
Edge e[] = new Edge[N+numRoutes-1];
for(int i=0;i<N+numRoutes-2;i++) {
    e[i] = new Edge(path[i].getID(),path[i+1].getID(),
 getDistance(path[i].getID(),path[i+1].getID()));
    if(path[i].getRoute()>0) e[i].setRoute(path[i].getRoute()
 );
    else e[i].setRoute(path[i+1].getRoute());
}
e[N+numRoutes-2] = new Edge(path[N+numRoutes-2].getID(),path
 [0].getID(),getDistance(path[N+numRoutes-2].getID(),path[0].
 getID()));
e[N+numRoutes-2].setRoute(path[N+numRoutes-2].getRoute());
p.setEdge(e);
//p.setNode(path);
p.updatePosition();
return p;
  }

  private void buildRoutes() {
int[] tour_id = new int[N];
int cur_route_id = 0;
for(int i=0;i<N; i++) tour_id[i] = 0;
for(int i=1; i<N; i++) {
    if(tour_id[i]==0) {
  VRPNode start = nodes[i];
  VRPNode now = start;
  while(now.getID()!=0) {
      now.setRoute(cur_route_id+1);
      tour_id[now.getID()] = 1;
      now = now.getNext();
  }
```

```
  now = start.getPrior();
  while(now.getID()!=0) {
       now.setRoute(cur_route_id+1);
       tour_id[now.getID()] = 1;
       now = now.getPrior();
  }
  cur_route_id++;
    }
}
numRoutes = cur_route_id;
for(int i=0;i<MAX_ROUTES;i++){
    routes[i].setLoad(0);
    routes[i].setDistance(0.0);
}
numRoutes = cur_route_id;
//System.out.println("There are " + numRoutes +" route in the
  map");
int loads[] = new int[numRoutes];
for(int i=0; i<numRoutes; i++) loads[i] = 0;
for(int i=1;i<N;i++)
    loads[nodes[i].getRoute()-1]+=nodes[i].getDemand();
for(int i=0; i<numRoutes;i++)
    routes[i].setLoad(loads[i]);
for(int i=1;i<N;i++)
    routes[nodes[i].getRoute()-1].setDistance(length[i]);
  }

  /**
     we insert some dummy depot to get a giant TSP path
  */
  public void buildPath() {
path = new VRPNode[N+numRoutes-1];
int bitset[] = new int[N];
for(int i=0; i< N; i++) bitset[i] = 1;
int cur_pos = 0; int cur_depot = 0;
depots = new VRPNode[numRoutes];
VRPNode end[] = new VRPNode[numRoutes];//the last node in
 each route
depots[0] = nodes[0];
// setup the dummy depots
for(int i=1;i<numRoutes;i++) {
    depots[i] = new VRPNode(nodes[0].x,nodes[0].y,N+i-1);
    depots[i].setDemand(0);
}
VRPNode next, prior;
```

```
  for(int i=1; i< N; i++) {
      if(bitset[i]>0) {
    VRPNode start;
    VRPNode now = nodes[i];
    bitset[i] = 0;
    while( (next=now.getNext()).getID()!=0) {
        now = next;
        bitset[now.getID()]=0;
    }
    end[cur_depot] = now;
    now = nodes[i];
    while( (prior=now.getPrior()).getID()!=0 ) {
        now = prior;
        bitset[now.getID()] = 0;
    }
    start = now;
    start.setPrior(depots[cur_depot]);
    path[cur_pos] = depots[cur_depot];
    depots[cur_depot].setNext(start);
    path[cur_pos].setPosition(cur_pos);
    while( now.getID()!=0 ) {
        cur_pos++;
        path[cur_pos] = now;
        now.setPosition(cur_pos);
        prior = now;
        now = now.getNext();
    }
    prior.setNext(depots[(++cur_depot)%numRoutes]);
    cur_pos++;
      }
  }
  for(int i=0; i<numRoutes; i++)
      depots[i].setPrior(end[(i+numRoutes-1)%numRoutes]);
    }

    public void createTabuList() {
  tabulist = new SimpleTabuList(20);
  tabulist.setSize(N);
    }
}
```

A.3 Node.java

```
// a general node class
// only x,y coordinate and a demand is associated with it
```

```
public class Node
{
    // asume the depot is node 0
    public Node() {
  x = 0;
  y = 0;
  demand = 0;
    }

    public Node(double i, double j)
    {
        x = i;
        y = j;
  demand = 0;
    }

    public Node(double i, double j, int q)
    {
        x = i;
        y = j;
  demand = q;
    }

    public String toString()
    {
        return new String("("+demand+")[" + x + "," + y + "]");
    }

    public int getDemand() {
  return demand;
    }

    public void setDemand(int q) { demand = q; }
    public double x,y; // the x,y coordinates for the node
    public int demand;
}
```

A.4 VRPNode.java

```
public class VRPNode extends Node
{

    public VRPNode(double x,double y) {
  super(x,y);
  prior = null;
```

```
    next = null;
    route = 0;
      }

      public VRPNode(double x,double y, int i) {
    super(x,y); prior = null; next=null;
    id = i; route = 0;
      }

      public VRPNode(VRPNode v) {
    super(v.x,v.y);
    prior = v.prior;
    next = v.next;
    id = v.id;
    pos = v.pos;
    route = v.route;
      }
      public void setRoute(int r) { route = r; }
      public int getRoute() { return route; }
      public int getID() { return id; }
      public boolean isDepot() { return id==0; }
      public VRPNode getPrior() { return prior;}
      public VRPNode getNext() { return next; }
      public void setPrior(VRPNode p) { prior = p;}
      public void setNext(VRPNode n) { next = n; }
      public VRPNode get(int i) { return i==0?next:prior; }
      public String toString() { return super.toString()+"("+id+"
     ,"+pos+")"; }
      public boolean adjacentToDepot() { return prior.getID()==0
     | next.getID()==0; }
      public void setPosition(int p) { pos = p; }
      public int getPosition() { return pos; }
      public void setId(int i) { id=i; }
      /**
         the demand associated with the node
      */
      VRPNode prior;
      VRPNode next;
      private int id;
      private int route;
      private int pos;
}
```

A.5 VRPFileReader.java

```
import java.io.*;
import java.util.*;

public class VRPFileReader {
    String data;
    public VRPFileReader(String fileName) {
  data = new String(fileName);
    }

    /*
     the data file should follow some standard as described in
   TSPLIB
    */
    public VRPGraph readData() throws IOException,
   FileNotFoundException
    {
  BufferedReader in = new BufferedReader(new FileReader(data));
  String content;
  int capacity=0, numNodes=0;
  double disCons = 1e10;
  double x_min,x_max,y_min,y_max;
  boolean ovrp = false;
  //for traffic jam region
  x_min = 1e10; y_min=1e10; x_max=-1e10; y_max=-1e10;
  TrafficJamRegion tjr = null;
  double jamFactor = 1.0;
  VRPNode nodes[]=null;
  while( ( content = in.readLine() ) != null ) {
      int i = content.indexOf(':');
      if(i>0) {
   String keyword = content.substring(0,i).trim();
   String value = content.substring(i+1,content.length()).trim
   ();
    if (keyword.equals("DIMENSION")) {
        numNodes = Integer.parseInt(value);
        nodes = new VRPNode[numNodes];
    }
    if (keyword.equals("CAPACITY"))  capacity = Integer.
   parseInt(value);
    if (keyword.equals("DISTANCE"))  disCons = Double.
   parseDouble(value);
    if (keyword.equals("TYPE") && value.equals("OVRP")) ovrp =
   true;
      }
      else {
```

```
  if(content.equals("NODE_COORD_SECTION")) {
      for(i=0 ; i<numNodes ; i++) {
    String coord = in.readLine();
    StringTokenizer st = new StringTokenizer(coord.trim());
    double [] data = new double[3];
    int k=0;
    while(st.hasMoreTokens()) {
        data[k++] = Double.parseDouble(st.nextToken());
    }
    nodes[(int)data[0]-1] = new VRPNode(data[1],data[2],(int)
 data[0]-1);
    if(data[1]>x_max) x_max = data[1];
    if(data[1]<x_min) x_min = data[1];
    if(data[2]>y_max) y_max = data[2];
    if(data[2]<y_min) y_min = data[2];
      }
  }
  else {
      if(content.equals("DEMAND_SECTION")) {
    for(i=0; i<numNodes; i++) {
        String demand = in.readLine();
        StringTokenizer st = new StringTokenizer(demand.trim
 ());
        int n = Integer.parseInt(st.nextToken());
        int q = Integer.parseInt(st.nextToken());
        nodes[n-1].setDemand(q);
    }
      }
      else
    if(content.equals("TRAFFIC_JAM_SECTION")) {
        String jam = in.readLine();
        StringTokenizer st = new StringTokenizer(jam.trim());
        double data[] = new double[5]; int k=0;
        while(st.hasMoreTokens()) {
      data[k++] = Double.parseDouble(st.nextToken());
        }
        tjr = new TrafficJamRegion(data);
        jamFactor = data[4];
    }
  }
    }
}
VRPGraph g =  new VRPGraph(nodes,capacity,disCons);
g.setOVRP(ovrp);
System.out.println("this is a ovrp problem");
```

```
  if(tjr!=null){
      g.setTrafficJamRegion(tjr);
      g.setJamFactor(jamFactor);
  }
  return g;
    }
}
```

A.6 Path.java

```
import java.util.*;
/**
 *  a path is here is a closed tour
 *  in a graph
 *  everything begin with TD_ means for
 * Time-dependent VRP
*/
public class Path implements Cloneable {
    private VRPGraph G;
    public double length;
    public Edge[] e;
    boolean debug = false;
    boolean debug1 = false;
    private int pos[];
    public int[] routes;
    final int MAX_ROUTES=100;
    static boolean TIME_DEPENDENT_VRP = false;
    // load for each route
    private int load[];
    // route length for each route
    private double dist[];
    static int nc=0;
    boolean useTabu;
    boolean do2OPTBetweenRoutes = true;
    Random rand = new Random();
    public Path(VRPGraph g) {
  G = g;
  length = 0.0;
  pos = new int[G.size()];
  load = new int[MAX_ROUTES];
  dist = new double[MAX_ROUTES];
  for(int i=0;i<MAX_ROUTES;i++) {
      load[i]=G.routes[i].getLoad();
      dist[i]=G.routes[i].getDistance();
  }
```

```
  }

  public Object clone() {
nc++;
Path p=null;
try {
    p = (Path) super.clone();
} catch(CloneNotSupportedException e) { System.out.println(e)
 ; }
//Path p = new Path(G);
p.length = length;
p.G = G;
p.pos = (int []) this.pos.clone();
//  for(int i=0;i<G.size();i++) p.pos=(int []) this.pos.clone
 (); //p.pos[i]=pos[i];
p.e = (Edge []) this.e.clone();
for(int i=0;i<e.length;i++) {
    p.e[i] = new Edge(e[i]);
}
p.load = (int []) this.load.clone();
p.dist = (double []) this.dist.clone();
p.tour = (int []) this.tour.clone();
p.inverse = (int []) this.inverse.clone();
p.reverse = this.reverse;
p.routes = (int []) this.routes.clone();
return p;
  }

  public void setEdge(Edge[] edge) {
length = 0.0;
e = new Edge[edge.length];
for(int i=0; i<edge.length; i++) {
    e[i] = new Edge(edge[i]);
    length+=e[i].length();
}
init_tour(e.length);
  }

  public void setPath(int tour[]) {
if(e==null)
    e=new Edge[tour.length-1];
for(int i=0; i<tour.length-1;i++)
    e[i] = new Edge(tour[i],tour[i+1],G.getDistance(tour[i],
 tour[i+1]));
int rid=0;
```

```
for(int i=0;i<tour.length-1;i++) {
    if(tour[i]==0){
  rid++;
    }
    load[rid-1]+=(G.nodes[tour[i]]).getDemand();
    dist[rid-1]+=e[i].length();
    e[i].setRoute(rid);
}
  }

  public void dump() {
for(int i=0;i<e.length;i++)
    System.out.println(e[i]);
for(int i=0;i<MAX_ROUTES;i++)
    if(load[i]>0)
  System.out.println("route["+(i+1)+"]="+dist[i]+",computed="
 +getRouteLength(i));
  }
  public String toString() { return new String("tour length
 is" + length); }
  public double getLength() { return length; }

  /*
   * pre-condition: i < j
   */

  private void swapEdge(int i,int j) {
Edge e1 = new Edge(e[i].from,e[j].from,G.getDistance(e[i].
 from,e[j].from));
Edge e2 = new Edge(e[i].to,e[j].to, G.getDistance(e[i].to,e[j
 ].to));
e1.setRoute(e[i].getRoute()); e2.setRoute(e[j].getRoute());
e[i] = e1; e[j] = e2;

for(int k=i+1;k<j;k++) e[k].swap();
int n=j-i-1;
for(int t=1;t<=n/2;t++) {
    Edge temp=e[i+t];
    e[i+t]=e[j-t];
    e[j-t]=temp;
}
  }

  public void dump_edge(int i,int j) {
for(int k=i;k<=j;k++) System.out.println(e[k]);
```

```
  }

  private boolean seek(int i,int dir) {
boolean improved = false;
Edge e1 = e[i];
int C1 = e1.from;
int C2 = e1.to;
int id2;
if(dir==0)
    id2 = C2<G.size()?C2:0;
else
    id2 = C1<G.size()?C1:0;

for(int m=0;m<G.NBCount[id2];m++) {
    int C3 = pos[G.NBList[id2][m]]; int C4=-1;
    int j=0;
    if(dir==0) {
  if(C3!=i+1) {
      if(C3>0) j=C3-1;
      else j=e.length-1;
      C4=e[j].from;
  }
  if(j==i+1 | j==i-1) continue;
    }
    else {
  if(C3==i) continue;
  else j=C3;
  if(j==i+1 | j==i-1) continue;
  C4=e[j].to;
    }
    Edge e2 = e[j];
    double d12 = e1.length();
    double d23;
    if(dir==0)
  d23 = G.getDistance(e1.to,e2.to);
    else
  d23 = G.getDistance(e1.from,e2.from);
    if(d23>=d12) break;
    else {
  double d14;
  if(dir==0)
      d14 = G.getDistance(e1.from,e2.from);
  else
      d14 = G.getDistance(e1.to,e2.to);
  double d34 = e2.length();
```

```
  double saving = d12+d34-d23-d14;
  if(saving>G.EPS && (i!=j) ) {
      if(e1.inTheSameRoute(e2)) {
    if(i<j)
        swapEdge(i,j);
    else
        swapEdge(j,i);
    improved = true;
    length-=saving;
    /**
       update the length for that route
    */
    dist[e1.getRoute()-1]-=saving;
      }
      else {
    //System.out.println("(between route)"+e1+","+e2);
    if(i<j) {
        if(checkFeasibility(i,j,true)){
      swapEdge(i,j);
      improved=true;
      length-=saving;
        }
    }
    else{
        if(i>j) {
      if(checkFeasibility(j,i,true)){
          swapEdge(j,i);
          improved=true;
          length-=saving;
      }
        }
    }
      }
  }
    }
    if(improved) {
  updatePosition();
  break;
    }
}
return improved;
  }

  public void updatePosition(){
for(int k=0;k<e.length;k++) {
```

```
    int r = e[k].from;
    if(r<G.size()) pos[r]=k;
}
  }

  /**
    check the feasibility of doing a 20PT move between
    link i and j.
     i<j
  */

  private boolean checkFeasibility(int i,int j,boolean flag)
 {
//System.out.println("i="+i+",j="+j+"flag="+flag);
int r1 = e[i].getRoute();
int r2 = e[j].getRoute();
int p1=0; int p2=0;
double d1=0.0; double d2=0.0;
int k=i+1;
final int M = e.length;
while(k<M & e[k].getRoute()==r1){
    p1+=G.nodes[e[k].from].getDemand();
    d1+=e[k].length();
    k++;
}
k=j-1;
while(k>=0 & e[k].getRoute()==r2){
    p2+=G.nodes[e[k].to].getDemand();
    d2+=e[k].length();
    k--;
}
int n1 = load[r1-1]-p1+p2;
int n2 = load[r2-1]-p2+p1;
double len1 = dist[r1-1]-d1+d2+G.getDistance(e[i].from,e[j].
 from)-e[i].length();
double len2 = dist[r2-1]-d2+d1+G.getDistance(e[i].to,e[j].to)
 -e[j].length();
if(n1<=G.getCapacity() & n2<=G.getCapacity() & len1<=G.
 getDisConstraint() & len2<=G.getDisConstraint()) {
    if(flag) {
  k=i+1;
  while(k<e.length & e[k].getRoute()==r1){
      e[k].setRoute(r2);
      k++;
  }
```

```
  k=j-1;
  while(k>=0 & e[k].to<G.size()){
      e[k].setRoute(r1);
      k--;
  }
  load[r1-1]=n1;
  load[r2-1]=n2;
  dist[r1-1]=len1;
  dist[r2-1]=len2;
    }
    return true;
}
else return false;
  }

  public boolean twOptStep() {
//  System.out.println("Improving.....");
boolean improved = false;
for(int i=0;i<e.length;i++) {
    improved = seek(i,0);
    if(!improved) improved=seek(i,1);
    if(improved) break;
}
return improved;
  }

  public void dumpPath() {
System.out.print("tour=[");
for(int i=0;i<e.length;i++) {
    if(e[i].from>=G.size()) System.out.print("0;");
    else System.out.print(e[i].from+";");
}
System.out.println(e[0].from+"];");
for(int i=0;i<MAX_ROUTES;i++) {
    if(load[i]>0) {
  System.out.println("load="+load[i]+",distance="+dist[i]);
  if(TIME_DEPENDENT_VRP) {
      if(getClockwiseRouteLength(i)<
 getAntiClockwiseRouteLength(i))
    System.out.println("direction -->");
      else
    System.out.println("direction <--");
  }

    }
```

```
}

System.out.println("clone is called "+ nc + " times");
  }

  public double twOpt() {
//   System.out.println("Doing twOpt....");
while(twOptStep());
return length;
  }

  public void clean() {
twOpt();
OPM();
TPM();
//ThreeOPT();
  }

  public void cleanUp() {
double d1,d2,d3,d4;
do {
    d1 = twOpt();
    d2 = OPM();
    d3 = TPM();
    d4 = ThreeOPT();
} while(d1!=d2 | d2!=d3| d2!=d4 | d3!=d4);
  }
  /*
  private void updateDeviation() {
deviation = 0.005*length;
  }
  */
  /**
     twOpt record to record travel
  */

  public void dump_edge()
  {
dump_edge(0,e.length-1);
  }

  boolean twOpt_RRT(double best_record, double deviation) {
//checkDistance("before 2opt-RRT");
//dump_edge();
boolean moved=false;
```

```
int size = G.size() + G.numRoutes-1;
for(int i=0;i<size;i++) {
    //       if(e[i].selfLoop()) continue;
    double saving;
    boolean swap=false;
    boolean done_for_this_node=false;
    Edge e1 = e[i];
    double d12 = e1.length();
    Edge e2;
    int C1 = e1.from;
    int C2 = e1.to;
    double best_saving = -1e10;
    int best_j = -1;
    for(int dir=0;dir<2;dir++) {
  int id2;
  if(dir==0)
      id2 = C2<G.size()?C2:0;
  else
      id2 = C1<G.size()?C1:0;
  for(int m=0;m<G.NBCount[id2];m++) {
      int C3 = pos[G.NBList[id2][m]];
      int j=0;
      if(dir==0) {
    if(C3!=i+1) {
        if(C3>0) j=C3-1;
        else j=e.length-1;
    }
    if(j==i+1 | j==i-1) continue;
      }

      else {
    if(C3==i) continue;
    else j=C3;
    if(j==i+1|j==i-1) continue;
      }
      e2 = e[j];
      double d23;
      if(dir==0)
    d23 = G.getDistance(e1.to,e2.to);
      else
    d23 = G.getDistance(e1.from,e2.from);
      double d14;
      if(dir==0)
    d14 = G.getDistance(e1.from,e2.from);
      else
```

```
    d14 = G.getDistance(e1.to,e2.to);
      double d34 = e2.length();
      saving = d12 + d34 - d23 - d14;
      if(e1.inTheSameRoute(e2)) {
    if(saving>G.EPS) {
        best_saving = saving;
        best_j=j;
        done_for_this_node=true;
        swap=true;
        break;

    }
    else {
        if(saving>best_saving) {
      best_saving = saving;
      best_j = j;
      swap=true;
        }
    }
      }
      else {
    if(saving>G.EPS) {
        if(checkFeasibility(Math.min(i,j),Math.max(i,j),false
)) {
      best_saving = saving;
      best_j=j;
      done_for_this_node=true;
      swap=true;
      break;
        }
    }
    else {
        if(saving>best_saving) {
      if(checkFeasibility(Math.min(i,j),Math.max(i,j),false))
 {
          best_saving = saving;
          best_j = j;
          swap=true;
      }
        }
    }
      }
      if(done_for_this_node) break;
 }
 if(done_for_this_node) break;
```

```
    }
    if(swap && (i!=best_j)) {
 e2 = e[best_j];
 //System.out.println("[debug]i="+i+"best_j="+best_j+"
best_saving="+best_saving);
 if(length-best_saving<best_record+deviation) {
      if(e1.inTheSameRoute(e2)) {
    //check route length feasibility
    if(dist[e1.getRoute()-1]-best_saving<=G.getDisConstraint
()) {
        //System.out.println("[debug]within route 2opt("+i
+","+best_j+")");
        if(i<best_j)
      swapEdge(i,best_j);
        else
      swapEdge(best_j,i);
        updatePosition();
        length-=best_saving;
        //update route distance
        dist[e1.getRoute()-1]-=best_saving;
        //checkDistance("after swapEdge("+i+","+best_j+")");
        moved=true;
    }
      }
      else {
    //System.out.println("[debug]2opt between routes");
    if(i<best_j) {
        if(checkFeasibility(i,best_j,true)){
      swapEdge(i,best_j);
      updatePosition();
      length-=best_saving;
      //checkDistance("after swapEdge("+i+","+best_j+")");
      moved=true;
        }
    }
    else{
        if(checkFeasibility(best_j,i,true)){
      swapEdge(best_j,i);
      updatePosition();
      length-=best_saving;
      //checkDistance("after swapEdge("+i+","+best_j+")");
      moved=true;
        }
    }
      }
```

```
  }
    }
}
//checkDistance("after 2opt-RRT");
return moved;
  }

  boolean twOpt_RRT(Record r) {
boolean moved=false;
int size = G.size() + G.numRoutes-1;
for(int i=0;i<size;i++) {
    //       if(e[i].selfLoop()) continue;
    double saving;
    boolean swap=false;
    boolean done_for_this_node=false;
    Edge e1 = e[i];
    double d12 = e1.length();
    Edge e2;
    int C1 = e1.from;
    int C2 = e1.to;
    double best_saving = -1e10;
    int best_j = -1;
    for(int dir=0;dir<2;dir++) {
  int id2;
  if(dir==0)
      id2 = C2<G.size()?C2:0;
  else
      id2 = C1<G.size()?C1:0;
  for(int m=0;m<G.MAX_NB;m++) {
      int C3 = pos[G.NBList[id2][m]];
      int j=0;
      if(dir==0) {
    if(C3!=i+1) {
        if(C3>0) j=C3-1;
        else j=e.length-1;
    }
    if(j==i+1 | j==i-1) continue;
      }

      else {
    if(C3==i) continue;
    else j=C3;
    if(j==i+1|j==i-1) continue;
      }
      e2 = e[j];
```

```
      double d23;
      if(dir==0)
    d23 = G.getDistance(e1.to,e2.to);
      else
    d23 = G.getDistance(e1.from,e2.from);
      double d14;
      if(dir==0)
    d14 = G.getDistance(e1.from,e2.from);
      else
    d14 = G.getDistance(e1.to,e2.to);
      double d34 = e2.length();
      saving = d12 + d34 - d23 - d14;
      if(e1.inTheSameRoute(e2)) {
    if(saving>G.EPS) {
        best_saving = saving;
        best_j=j;
        done_for_this_node=true;
        swap=true;
        break;

    }
    else {
        if(saving>best_saving) {
      best_saving = saving;
      best_j = j;
      swap=true;
        }
    }
      }
      else {
    if(saving>G.EPS) {
        if(checkFeasibility(Math.min(i,j),Math.max(i,j),false
)) {
      best_saving = saving;
      best_j=j;
      done_for_this_node=true;
      swap=true;
      break;
        }
    }
    else {
        if(saving>best_saving) {
      if(checkFeasibility(Math.min(i,j),Math.max(i,j),false))
 {
          best_saving = saving;
```

```
            best_j = j;
            swap=true;
        }
          }
    }
        }
        if(done_for_this_node) break;
  }
  if(done_for_this_node) break;
    }
    if(swap) {
  e2 = e[best_j];
  if(r.accept(length-best_saving)) {
        if(e1.inTheSameRoute(e2)) {
    //check route length feasibility
    if(dist[e1.getRoute()-1]-best_saving<=G.getDisConstraint
()) {
          if(i<best_j)
        swapEdge(i,best_j);
          else
        swapEdge(best_j,i);
          updatePosition();
          length-=best_saving;
          //update route distance
          dist[e1.getRoute()-1]-=best_saving;
          moved=true;
    }
        }
        else {
    if(i<best_j) {
          if(checkFeasibility(i,best_j,true)){
        swapEdge(i,best_j);
        updatePosition();
        length-=best_saving;
        moved=true;
          }
    }
    else{
          if(checkFeasibility(best_j,i,true)){
        swapEdge(best_j,i);
        updatePosition();
        length-=best_saving;
        moved=true;
          }
    }
```

```
      }
  }
    }
    if(length<r.length) {
  r.length=length; r.deviation=0.01*length;
    }
}
return moved;
  }

  private void updateCurrentSolution(Path best) {
length = best.length;
for(int i=0;i<pos.length;i++) pos[i]=best.pos[i];
for(int i=0;i<e.length;i++) e[i] = new Edge(best.e[i]);
for(int i=0;i<load.length;i++) load[i] = best.load[i];
  }

  /**
     move e[i].to between e[j]
  */

  public  boolean OPMStep() {
boolean improved = false;
int M=e.length;
for(int i=0;i<M;i++) {
    if(e[i].to>0 & e[i].to<G.size()) {
  int q = G.nodes[e[i].to].getDemand();
  int s = e[i].getRoute()-1;
  double dold = e[i].length() + e[(i+1)].length();
  double dnew = G.getDistance(e[i].from,e[i+1].to);
  //for(int j=(i+2)%M;j!=i;j=(j+1)%M) {
  for(int m=0;m<G.NBCount[e[i].to];m++) {
      int j = pos[G.NBList[e[i].to][m]];
      if(j==i||j==i+1) continue;
      double d0=dold+e[j].length();
      double d1=dnew+G.getDistance(e[i].to,e[j].from)+G.
 getDistance(e[i].to,e[j].to);
      double saving = d0-d1;
      if(saving>0 & Math.abs(saving)>G.EPS) {
    if( e[i].inTheSameRoute(e[j]) ) {
        OPMUpdate(i,j);
        length-=saving;
        // update the route length
        dist[s]-=saving;
        improved = true;
```

```
    }
    else {
        int r=e[j].getRoute()-1;
        if(load[r]+q <= G.getCapacity()) {
      double dlen = d1-dnew-e[j].length();
      //check the distance constraint
      if(dist[r]+dlen<=G.getDisConstraint()) {
          load[r]+=q;
          load[s]-=q;
          dist[r]+=dlen;
          dist[s]-=dold-dnew;
          length-=saving;
          OPMUpdate(i,j);
          improved = true;
      }
        }
    }
      }
      if(improved) {
    updatePosition();
    break;
      }
  }
    }
    if(improved) break;
}
return improved;
  }

  public  boolean OPM_RRT(double record,double deviation) {
//  System.out.println("before OPM_RRT");
checkDistance("before_OPM_RRT");
boolean moved = false;
int M=e.length;
for(int i=0;i<M;i++) {
    if(e[i].to>0 & e[i].to<G.size()) {
  //    if(e[i].to>=G.size()) { System.out.println("faint!");
  System.exit(1); }
  boolean done_for_this_node = false;
  int q = G.nodes[e[i].to].getDemand();
  int s = e[i].getRoute()-1;
  double dold = e[i].length() + e[(i+1)%M].length();
  double dnew = G.getDistance(e[i].from,e[(i+1)%M].to);
  double best_saving=-1e10; int best_j=-1;
  //for(int j=(i+2)%M;j!=i;j=(j+1)%M) {
```

```
 for(int m=0;m<G.NBCount[e[i].to];m++) {
      int j = pos[G.NBList[e[i].to][m]];
      if(j==i||j==i+1) continue;
      double d0=dold+e[j].length();
      double d1=dnew+G.getDistance(e[i].to,e[j].from)+G.
getDistance(e[i].to,e[j].to);
      double dlen = G.getDistance(e[i].to,e[j].from)+G.
getDistance(e[i].to,e[j].to)-e[j].length();
      double saving = d0-d1;
      if(saving>G.EPS) {
    if(e[i].inTheSameRoute(e[j])) {
      //System.out.println("OPM RRT for "+e[i]+","+e[j]);
        OPMUpdate(i,j);
      // update route length
        dist[s]-=saving;
        length-=saving;
      //checkDistance();
        updatePosition();
        moved=true;
        done_for_this_node = true;
        break;
    }
    else {
        int r=e[j].getRoute()-1;
        if(load[r]+q <= G.getCapacity()) {
      if(dist[r]+dlen<=G.getDisConstraint()) {
          load[r]+=q;
          load[s]-=q;
          dist[r]+=dlen;
          dist[s]-=dold-dnew;
          length-=saving;
          //System.out.println("OPM_RRT for "+e[i]+","+e[j]);
          OPMUpdate(i,j);
          //checkDistance();
          updatePosition();
          moved = true;
          done_for_this_node=true;
          break;
      }
        }
        }
      }
      else {
    if(Math.abs(saving)>G.EPS & saving > best_saving) {
        if(e[i].inTheSameRoute(e[j]) & dist[s]-saving<=G.
```

```
getDisConstraint()) {
      best_saving = saving;
      best_j = j;
      }
        else {
      int r = e[j].getRoute()-1;
      if( load[r]+q<=G.getCapacity() & (dist[r]+dlen<=G.
getDisConstraint()) ) {
          best_saving = saving;
          best_j = j;
      }
        }
    }
      }

 }
 if(done_for_this_node) continue;
 else {
      if(best_j!=-1) {
    if(length-best_saving<record+deviation & Math.abs(
best_saving)>G.EPS) {
        if( e[i].inTheSameRoute(e[best_j]) ) {
      //System.out.println("OPM_RRT for "+e[i]+","+e[best_j])
;
      OPMUpdate(i,best_j);
      updatePosition();
      dist[s]-=best_saving;
      //checkDistance();
      if(dist[s]>G.getDisConstraint())
          System.out.println("[OPM_RRT]DISCONSTRAINT_ERROR");
      length-=best_saving;
      moved = true;
        }
        else {
      int r=e[best_j].getRoute()-1;
      if(load[r]+q <= G.getCapacity()) {
          double dl = G.getDistance(e[i].to,e[best_j].from)+G
.getDistance(e[i].to,e[best_j].to)-e[best_j].length();
          load[r]+=q;
          load[s]-=q;
          dist[r]+=dl;
          dist[s]-=dold-dnew;
          if(dist[r]>G.getDisConstraint())
        System.out.println("[OPM_RRT]DISCONSTRAINT_ERROR");
          //   System.out.println("OPM_RRT for "+e[i]+","+e[
```

```
best_j]);
            length-=best_saving;
            OPMUpdate(i,best_j);
            //  checkDistance();
            updatePosition();
            moved = true;
        }
          }
      }
        }
    }
      }
}
//  System.out.println("finish OPM_RRT");
checkDistance("after OPM_RRT");
//System.out.println("leave OPM_RRT");
return moved;
  }

  public  boolean OPM_RRT(Record R) {
//  System.out.println("before OPM_RRT");
//checkDistance();
boolean moved = false;
int M=e.length;
for(int i=0;i<M;i++) {
    if(e[i].to>0 & e[i].to<G.size()) {
  boolean done_for_this_node = false;
  int q = G.nodes[e[i].to].getDemand();
  int s = e[i].getRoute()-1;
  double dold = e[i].length() + e[(i+1)%M].length();
  double dnew = G.getDistance(e[i].from,e[(i+1)%M].to);
  double best_saving=-1e10; int best_j=-1;
  for(int j=(i+2)%M;j!=i;j=(j+1)%M) {
      double d0=dold+e[j].length();
      double d1=dnew+G.getDistance(e[i].to,e[j].from)+G.
 getDistance(e[i].to,e[j].to);
      double dlen = G.getDistance(e[i].to,e[j].from)+G.
 getDistance(e[i].to,e[j].to)-e[j].length();
      double saving = d0-d1;
      if(saving>G.EPS) {
    if(e[i].inTheSameRoute(e[j])) {
      //System.out.println("OPM RRT for "+e[i]+","+e[j]);
        OPMUpdate(i,j);
      // update route length
        dist[s]-=saving;
```

```
        length-=saving;
      //checkDistance();
        updatePosition();
        moved=true;
        done_for_this_node = true;
        break;
    }
    else {
        int r=e[j].getRoute()-1;
        if(load[r]+q <= G.getCapacity()) {
      if(dist[r]+dlen<=G.getDisConstraint()) {
          load[r]+=q;
          load[s]-=q;
          dist[r]+=dlen;
          dist[s]-=dold-dnew;
          length-=saving;
          //System.out.println("OPM_RRT for "+e[i]+","+e[j]);
          OPMUpdate(i,j);
          //checkDistance();
          updatePosition();
          moved = true;
          done_for_this_node=true;
          break;
      }
        }
    }
      }
      else {
    if(Math.abs(saving)>G.EPS & saving > best_saving) {
        if(e[i].inTheSameRoute(e[j]) & dist[s]-saving<=G.
getDisConstraint()) {
      best_saving = saving;
      best_j = j;
      }
        else {
      int r = e[j].getRoute()-1;
      if( load[r]+q<=G.getCapacity() & (dist[r]+dlen<=G.
getDisConstraint()) ) {
          best_saving = saving;
          best_j = j;
      }
        }
    }
      }
```

```
 }
 if(done_for_this_node) continue;
 else {
     if(best_j!=-1) {
   if(R.accept(length-best_saving) & Math.abs(best_saving)>G
.EPS) {
       if( e[i].inTheSameRoute(e[best_j]) ) {
     //System.out.println("OPM_RRT for "+e[i]+","+e[best_j])
;
     OPMUpdate(i,best_j);
     updatePosition();
     dist[s]-=best_saving;
     //checkDistance();
     if(dist[s]>G.getDisConstraint())
         System.out.println("[OPM_RRT]DISCONSTRAINT_ERROR");
     length-=best_saving;
     moved = true;
       }
       else {
     int r=e[best_j].getRoute()-1;
     if(load[r]+q <= G.getCapacity()) {
         double dl = G.getDistance(e[i].to,e[best_j].from)+G
.getDistance(e[i].to,e[best_j].to)-e[best_j].length();
         load[r]+=q;
         load[s]-=q;
         dist[r]+=dl;
         dist[s]-=dold-dnew;
         if(dist[r]>G.getDisConstraint())
       System.out.println("[OPM_RRT]DISCONSTRAINT_ERROR");
         //   System.out.println("OPM_RRT for "+e[i]+","+e[
best_j]);
         length-=best_saving;
         OPMUpdate(i,best_j);
         //   checkDistance();
         updatePosition();
         moved = true;
     }
       }
   }
     }
 }
   }
   if(length<R.length) {
R.length=length; R.deviation=0.01*length;
   }
```

```
}
return moved;
  }
  /**
     insert e[i].to in e[j]
  */
  private void OPMUpdate(int i,int j) {
Edge ei = new Edge(e[i].from,e[i+1].to,G.getDistance(e[i].
 from,e[i+1].to));
Edge ej = new Edge(e[j].from,e[i].to,G.getDistance(e[j].from,
 e[i].to));
Edge ek = new Edge(e[i].to,e[j].to,G.getDistance(e[i].to,e[j
 ].to));
ei.setRoute(e[i].getRoute());
ej.setRoute(e[j].getRoute());
ek.setRoute(e[j].getRoute());
if(i<j) {
    for(int t=i+2;t<j;t++)
  e[t-1]=e[t];
    e[j-1] = ej;
    e[j] = ek;
    e[i] = ei;
}
else {
    for(int t=i;t>j+1;t--) e[t]=e[t-1];
    e[j]=ej;
    e[j+1]=ek;
    e[i+1]=ei;
}
  }
  /**
    One Point Move heuristic
  */
  public double OPM(){
while(OPMStep());
return length;
  }

  /**
    Two points exchange heuristic
  */
  public boolean TPMStep() {
boolean improved = false;
int M = e.length;
for(int i=0;i<M;i++) {
```

```
   if(e[i].to<G.N & e[i].to>0) {
 int q = G.nodes[e[i].to].getDemand();
 int r = e[i].getRoute();
 double do1 = e[i].length() + e[(i+1)%M].length();
 for(int m=0;m<G.NBCount[e[i].to];m++) {
     int j = pos[G.NBList[e[i].to][m]];
     if(j==i) continue;
     //for(int j=(i+1)%M;j!=i;j=(j+1)%M){
     if( (e[j].getRoute()!=r) & (e[j].to<G.size() & e[j].to
>0)){
   int w = e[j].getRoute();
   double do2 =  e[j].length()+e[(j+1)%M].length();
   double dold = do1+do2;
   int s = G.nodes[e[j].to].getDemand();
   if(load[r-1]-q+s<=G.getCapacity()
      & load[e[j].getRoute()-1]-s+q<=G.getCapacity()) {
       double dn1 = G.getDistance(e[i].from,e[j].to)+G.
getDistance(e[(i+1)%M].to,e[j].to);
       double dn2 = G.getDistance(e[j].from,e[i].to)+G.
getDistance(e[(j+1)%M].to,e[i].to);
       double dnew = dn1+dn2;
       double saving = dold-dnew;
       if(saving>G.EPS) {
     if(dist[r-1]-do1+dn1<=G.getDisConstraint() & dist[w-1]-
do2+dn2<=G.getDisConstraint()) {
         load[r-1]+=s-q;
         load[w-1]+=q-s;
         dist[r-1]-=do1-dn1;
         dist[w-1]-=do2-dn2;

         Edge ei=new Edge(e[i].from,e[j].to,G.getDistance(e[
i].from,e[j].to));
         Edge ei_1 = new Edge(e[j].to,e[(i+1)%M].to,G.
getDistance(e[(i+1)%M].to,e[j].to));
         Edge ej = new Edge(e[j].from,e[i].to, G.getDistance
(e[j].from,e[i].to));
         Edge ej_1 = new Edge(e[i].to,e[(j+1)%M].to,G.
getDistance(e[(j+1)%M].to,e[i].to));
         ei.setRoute(r); ei_1.setRoute(r);
         ej.setRoute(w);ej_1.setRoute(w);
         e[i]=ei; e[(i+1)%M]=ei_1;
         e[j]=ej; e[(j+1)%M]=ej_1;
         length-=dold-dnew;
         updatePosition();
         improved = true;
```

```
      }
        }
    }
      }
      if(improved) break;
  }
    }
    if(improved) break;
}
return improved;
  }

  public double TPM() {
while(TPMStep());
return length;
  }

  public boolean TPM_RRT(double record, double deviation) {
checkDistance("before TPM_RRT");
boolean moved = false;
int M = e.length;
for(int i=0;i<M;i++) {
    if(e[i].to<G.N & e[i].to>0) {
  int q = G.nodes[e[i].to].getDemand();
  int r = e[i].getRoute();
  boolean done_for_this_node=false;
  double best_saving = -1e10; int best_j = -1;
  double do1 = e[i].length() + e[(i+1)%M].length();
  boolean swap=false;
  for(int m=0;m<G.NBCount[e[i].to];m++) {
      int j = pos[G.NBList[e[i].to][m]];
      if(j==i) continue;
      //for(int j=(i+1)%M; j!=i; j=(j+1)%M){
      if( (e[j].getRoute()!=r) & (e[j].to<G.size() & e[j].to
>0)){
    int w = e[j].getRoute();
    double do2 = e[j].length()+e[(j+1)%M].length();
    double dold = do1+do2;
    int s = G.nodes[e[j].to].getDemand();
    if(load[r-1]-q+s<=G.getCapacity()
       & load[e[j].getRoute()-1]-s+q<=G.getCapacity()) {
      double dn1 = G.getDistance(e[i].from,e[j].to)+G.
getDistance(e[(i+1)%M].to,e[j].to);
      double dn2 = G.getDistance(e[j].from,e[i].to)+G.
getDistance(e[(j+1)%M].to,e[i].to);
```

```
        double dnew = dn1+dn2;
        double saving = dold-dnew;
        if(saving>G.EPS) {
      if(dist[r-1]-do1+dn1<=G.getDisConstraint() & dist[w-1]-
do2+dn2<=G.getDisConstraint()) {
          best_saving = saving; best_j = j;
          done_for_this_node = true;
          swap=true;
          break;
      }
        }
        else{
      if(saving>best_saving & Math.abs(saving)>G.EPS) {
          if(dist[r-1]-do1+dn1<=G.getDisConstraint()& dist[w
-1]-do2+dn2<=G.getDisConstraint()) {
        best_saving = saving; best_j = j;
        swap=true;
          }
      }
        }
    }
      }
 }
 if(swap) {
      int j = best_j; int s = G.nodes[e[j].to].getDemand();
      if(done_for_this_node | length-best_saving<record+
deviation) {
    double do2 = e[j].length() + e[(j+1)%M].length();
    double dn1 = G.getDistance(e[i].from,e[j].to)+G.
getDistance(e[(i+1)%M].to,e[j].to);
    double dn2 = G.getDistance(e[j].from,e[i].to)+G.
getDistance(e[(j+1)%M].to,e[i].to);
    dist[r-1]-=do1-dn1;
    dist[e[j].getRoute()-1]-=do2-dn2;

    load[r-1]+=s-q;
    load[e[j].getRoute()-1]+=q-s;

    Edge ei=new Edge(e[i].from,e[j].to,G.getDistance(e[i].
from,e[j].to));
    Edge ei_1 = new Edge(e[j].to,e[(i+1)%M].to,G.getDistance(
e[(i+1)%M].to,e[j].to));
    Edge ej = new Edge(e[j].from,e[i].to, G.getDistance(e[j].
from,e[i].to));
    Edge ej_1 = new Edge(e[i].to,e[(j+1)%M].to,G.getDistance(
```

```
 e[(j+1)%M].to,e[i].to));
    ei.setRoute(r); ei_1.setRoute(r);
    ej.setRoute(e[j].getRoute());ej_1.setRoute(e[j].getRoute
 ());
    e[i]=ei; e[(i+1)%M]=ei_1;
    e[j]=ej; e[(j+1)%M]=ej_1;
    length-=best_saving;
    updatePosition();
    moved = true;
      }
  }
    }
}
checkDistance("after_TPM_RRT");
return moved;
  }

  public boolean TPM_RRT(Record R) {
boolean moved = false;
int M = e.length;
for(int i=0;i<M;i++) {
    if(e[i].to<G.N & e[i].to>0) {
  int q = G.nodes[e[i].to].getDemand();
  int r = e[i].getRoute();
  boolean done_for_this_node=false;
  double best_saving = -1e10; int best_j = -1;
  double do1 = e[i].length() + e[(i+1)%M].length();
  boolean swap=false;
  for(int j=(i+1)%M;j!=i;j=(j+1)%M){
      if( (e[j].getRoute()!=r) & (e[j].to<G.size() & e[j].to
 >0)){
    int w = e[j].getRoute();
    double do2 = e[j].length()+e[(j+1)%M].length();
    double dold = do1+do2;
    int s = G.nodes[e[j].to].getDemand();
    if(load[r-1]-q+s<=G.getCapacity()
       & load[e[j].getRoute()-1]-s+q<=G.getCapacity()) {
        double dn1 = G.getDistance(e[i].from,e[j].to)+G.
 getDistance(e[(i+1)%M].to,e[j].to);
        double dn2 = G.getDistance(e[j].from,e[i].to)+G.
 getDistance(e[(j+1)%M].to,e[i].to);
        double dnew = dn1+dn2;
        double saving = dold-dnew;
        if(saving>G.EPS) {
      if(dist[r-1]-do1+dn1<=G.getDisConstraint()
```

```
        & dist[w-1]-do2+dn2<=G.getDisConstraint()) {
           best_saving = saving; best_j = j;
           done_for_this_node = true;
           swap=true;
           break;
      }
        }
        else{
      if(saving>best_saving & Math.abs(saving)>G.EPS) {
           if(dist[r-1]-do1+dn1<=G.getDisConstraint()& dist[w
-1]-do2+dn2<=G.getDisConstraint()) {
        best_saving = saving; best_j = j;
        swap=true;
           }
      }
        }
    }
      }
 }
 if(swap) {
      int j = best_j; int s = G.nodes[e[j].to].getDemand();
      if(done_for_this_node | R.accept(length-best_saving) ){
    double do2 = e[j].length() + e[(j+1)%M].length();
    double dn1 = G.getDistance(e[i].from,e[j].to)+G.
getDistance(e[(i+1)%M].to,e[j].to);
    double dn2 = G.getDistance(e[j].from,e[i].to)+G.
getDistance(e[(j+1)%M].to,e[i].to);
    dist[r-1]-=do1-dn1;
    dist[e[j].getRoute()-1]-=do2-dn2;

    load[r-1]+=s-q;
    load[e[j].getRoute()-1]+=q-s;

    Edge ei=new Edge(e[i].from,e[j].to,G.getDistance(e[i].
from,e[j].to));
    Edge ei_1 = new Edge(e[j].to,e[(i+1)%M].to,G.getDistance(
e[(i+1)%M].to,e[j].to));
    Edge ej = new Edge(e[j].from,e[i].to, G.getDistance(e[j].
from,e[i].to));
    Edge ej_1 = new Edge(e[i].to,e[(j+1)%M].to,G.getDistance(
e[(j+1)%M].to,e[i].to));
    ei.setRoute(r); ei_1.setRoute(r);
    ej.setRoute(e[j].getRoute());ej_1.setRoute(e[j].getRoute
());
    e[i]=ei; e[(i+1)%M]=ei_1;
```

```
    e[j]=ej; e[(j+1)%M]=ej_1;
    length-=best_saving;
    updatePosition();
    moved = true;
      }
  }
    }
    if(length<R.length) {
  R.length=length; R.deviation = 0.01*length;
    }
}
return moved;
  }

  public void purturb() {
final int M = e.length;
class Ratio implements Comparable {
    int from,to,route;
    double r;
    Ratio(int i,int j,int ro, double ra) { from=i;to=j;route=
 ro; r=ra; }
    public int compareTo(Object o) {
  if(r<((Ratio)o).r) return -1;
  else if(r>((Ratio)o).r) return 1;
  else return 0;
    }
}
Ratio[] ratio = new Ratio[G.N-1];
for(int i=0;i<M;i++) {
    int id = e[i].to;
    if(id >0 & id<G.N) {
  double r=G.nodes[id].getDemand()/(e[i].length()+e[(i+1)%M].
 length()-G.getDistance(e[i].from,e[(i+1)%M].to));
  ratio[id-1]=new Ratio(e[i].from,id,e[i].getRoute(),r);
    }
}
Arrays.sort(ratio);
int k = Math.min((int)G.N/10,20);
for(int i=0;i<k;i++)
    insertNode(ratio[i].from,ratio[i].to,ratio[i].route);
  }

  private void insertNode(int tabu,int id, int route) {
int best_ins = -1; double best_cost = 1e8;
int q = G.nodes[id].getDemand();
```

```
double saving = e[pos[id]-1].length()+e[pos[id]].length()-G.
 getDistance(e[pos[id]-1].from,e[pos[id]].to);
for(int i=0;i<e.length;i++) {
    if(e[i].from!=tabu & e[i].from!=id & e[i].to!=id) {
  double inc = G.getDistance(e[i].from,id)+G.getDistance(id,e
 [i].to) - e[i].length();
  double cost= inc - saving;
  if(cost<best_cost) {
      if(e[i].getRoute()==route) {
    if(dist[route-1]+cost<=G.getDisConstraint()) {
        best_cost = cost;
        best_ins = i;
    }
      }
      else {
    if(load[e[i].getRoute()-1]+q<=G.getCapacity()) {
        if(dist[e[i].getRoute()-1]+inc<=G.getDisConstraint())
  {
      best_cost = cost;
      best_ins = i;
      }
    }
      }
  }
    }
}
if(best_ins!=-1) {
    int i = best_ins;
    int j = pos[id]-1;
    length+=best_cost;
    dist[route-1]-=saving;
    dist[e[i].getRoute()-1] += G.getDistance(e[i].from,id)+G.
 getDistance(id,e[i].to) - e[i].length();
    load[e[i].getRoute()-1]+=q;
    load[route-1]-=q;
    OPMUpdate(j,i);
    updatePosition();
}
  }


  public void checkCapacity() {
for(int i=1;i<MAX_ROUTES;i++) {
    if(load[i-1]>0) {
  System.out.print("load="+load[i-1]);
```

```
  int q=0;
  for(int k=0;k<e.length;k++) {
      if(e[k].getRoute()==i)
    if(e[k].from<G.size()) q+=G.nodes[e[k].from].getDemand();
  }
  System.out.println("\tq="+q);
  if(q!=load[i-1]) {
      System.out.println("route's load is not correctly
 maintained");
  }
  if( q>G.getCapacity() )
      System.out.println("Capacity constraint is voalated");
    }
}
  }
  public void checkDistance() {
for(int i=1;i<MAX_ROUTES;i++) {
    double dist = 0.0; //int cap=0;
    for(int j=0;j<e.length;j++) {
  if(e[j].getRoute()==i) {
      dist+=e[j].length();
      //cap+=e[j].getDemand();
  }
    }
    if(dist>G.getDisConstraint() | Math.abs(dist-this.dist[i
 -1])>1e-6 ) {
  System.out.println("distance constraint is violated on
 route " + i + " with dist "+dist);
  System.out.println("recorded distance for route " + i +" is
 "+ this.dist[i-1]);
  System.exit(1);
    }
}
  }

  public void checkDistance(String where) {
for(int i=1;i<MAX_ROUTES;i++) {
    double dist = 0.0; //int cap=0;
    for(int j=0;j<e.length;j++) {
  if(e[j].getRoute()==i) {
      dist+=e[j].length();
      //cap+=e[j].getDemand();
  }
    }
    if(dist>G.getDisConstraint() | Math.abs(dist-this.dist[i
```

```
 -1])>1e-6 ) {
  System.out.println("distance constraint is violated on
 route " + i + " with dist "+dist);
  System.out.println("recorded distance for route " + i +" is
 "+ this.dist[i-1]);
  dump_edge(0,e.length-1);
  System.exit(1);
    }
    //       if(cap>G.getCapacity() | cap!=load[i-1]) {
    //  System.out.println("capacity constraint is vialated
 on route "+i+"with cap "+cap);
    //  System.out.println("record capacity is "+load[i-1]);
    //  System.exit(1);
    // }
}
  }
  class Record {
public double length;
public double deviation;
public Record(double l,double d) { length=l; deviation=d;}
public boolean accept(double o) { return o-length<deviation;
 }
  }

  public Path improve() {
int counter=1; int I = 30;
Path best_record = (Path)clone();
double deviation = 0.01 * best_record.getLength();
//int [] I={30,25,20,15,10,5};
while(true) {
    boolean improved = false;
    // Record r = new Record(best_record.getLength(),
 deviation);
    for(int i=0;i<I;i++) {
  if(!OPM_RRT(best_record.getLength(),deviation) &
    !TPM_RRT(best_record.getLength(),deviation) &
    !twOpt_RRT(best_record.getLength(),deviation)) {
      //if(!OPM_RRT(r) & !TPM_RRT(r) &!twOpt_RRT(r)) {
      System.out.println("no movement in I loop, quit I loop"
 );
      break;
  }
  else{
      if(length<best_record.getLength()) {
    //System.out.println("[RRT]find a improving move,record
```

```
 is "+p.getLength());
    improved=true;
    best_record = (Path)clone();
    deviation=0.01*best_record.getLength();
    //  r = new Record(best_record.getLength(),deviation);
      }
 }
    } // I loop
    //System.out.println("counter="+counter);
    //checkDistance("before clean");
    clean();
    //checkDistance("after clean");
    if(length>best_record.getLength()) {
 //System.out.println("no improvement");
 improved = false;
    }
    else {
 if(Math.abs(length-best_record.getLength())>G.EPS) {
      improved = true;
      best_record = (Path) clone();
      deviation = 0.01*best_record.getLength();
      counter=0;
 }
    }
    counter++;
    //System.out.println("best record is "+ best_record.
 getLength());
    if(counter>5)
 break;
}
return best_record;
  }

  public boolean ThreeOPTStep(int r) {
boolean improved = false;
//  System.out.println("for route "+r);
//checkDistance();
for(int i=0;i<e.length;i++) {
    if(e[i].getRoute()==r) { //in case there is a self-loop
 of depot
 int j=i+2;
 while(j<e.length) {
      if(e[j].getRoute()==r ){
    int k=j+2;
    while(k<e.length) {
```

```
        if(e[k].getRoute()==r) {
      double[] savings = new double[4] ;
      for(int l=0;l<4;l++)
          savings[l] = ThreeOPTSaving(i,j,k,l);
      double max_saving = -G.INFTY;
      int type=-1;
      for(int l=0;l<4;l++) {
          if(savings[l]>max_saving){
        max_saving=savings[l];
        type = l;
          }
      }
      if(max_saving>1e-3) {
          //System.out.println("find 3OPT improvment:"+i+","+
 j+","+k+"with saving "+max_saving+"type:"+type);
          //System.out.println(":"+e[i]+e[j]+e[k]);
          // System.out.println("before 3OPT");checkDistance
 ();
          do3OPT(i,j,k,type);
          //System.out.println("3opt done");
          // System.out.println("after 3OPT");
          updatePosition();
          dist[r-1]-=max_saving;
          length -= max_saving;
          improved = true;
          //checkDistance();
      }
        }
        if(improved) break;
        k++;
    }
      }
      if(improved) break;
      j++;
  }
    }
    if(improved) break;
}
//  System.out.println("finish route "+r);
//checkDistance();
return improved;
  }

  public double ThreeOPT() {
//  dump();
```

```
for(int i=0;i<MAX_ROUTES;i++) {
    if(load[i]>0) {
  while(ThreeOPTStep(i)) ;//dump();
    }
}
return length;
  }

  private double ThreeOPTSaving(int i,int j,int k,int type) {
double saving ;
double old = e[i].length()+e[j].length()+e[k].length();
switch(type) {
case 0:
    saving = G.getDistance(e[i].from,e[j].from)+G.getDistance
 (e[i].to,e[k].from)
  + G.getDistance(e[j].to,e[k].to) - old;
    break;
case 1:
    saving =  G.getDistance(e[i].from,e[j].to)+G.getDistance(
 e[i].to,e[k].from)
  + G.getDistance(e[j].from,e[k].to) - old;
    break;
case 2:
    saving =  G.getDistance(e[i].from,e[j].to)+G.getDistance(
 e[j].from,e[k].from)
  + G.getDistance(e[i].to,e[k].to) - old;
    break;
case 3:
    saving =  G.getDistance(e[i].from,e[k].from)+G.
 getDistance(e[j].to,e[i].to)
  + G.getDistance(e[j].from,e[k].to) - old;
    break;
default:
    saving = G.INFTY;
    break;
}
saving = -saving;
return saving;
  }

  private void do3OPT(int i,int j,int k,int type) {
//System.out.println("type="+type);
int r = e[i].getRoute(); int itr;
Edge ei,ej,ek;
Edge [] copy;
```

```
copy = (Edge [])this.e.clone();
for(int t=0;t<e.length;t++) copy[t]=new Edge(e[t]);
switch(type) {
case 0:
    ei = new Edge(e[i].from,e[j].from,G.getDistance(e[i].from
 ,e[j].from));
    ej = new Edge(e[i].to,e[k].from,G.getDistance(e[i].to,e[k
 ].from));
    ek = new Edge(e[j].to,e[k].to,G.getDistance(e[j].to,e[k].
 to));
    ei.setRoute(r); ej.setRoute(r); ek.setRoute(r);
    e[i] = ei; e[j]=ej; e[k]=ek;
    reverseChain(i,j); reverseChain(j,k);
    break;
case 1:
    //      Edge [] copy = new Edge[e.length];
    //copy = (Edge [])this.e.clone();
    ei = new Edge(e[i].from,e[j].to,G.getDistance(e[i].from,e
 [j].to));
    ej = new Edge(e[k].from,e[i].to,G.getDistance(e[k].from,e
 [i].to));
    ek = new Edge(e[j].from,e[k].to,G.getDistance(e[j].from,e
 [k].to));
    ei.setRoute(r); ej.setRoute(r); ek.setRoute(r);
    copy[i] = ei;
    for(int t=1;t<k-j;t++) copy[i+t] = e[j+t];
    itr = i+k-j;
    copy[itr] = ej;
    for(int t=1;t<j-i;t++) copy[itr+t] = e[i+t];
    copy[k] = ek;
    e = copy;
    break;
case 2:
    ei = new Edge(e[i].from,e[j].to,G.getDistance(e[i].from,e
 [j].to));
    ej = new Edge(e[k].from,e[j].from,G.getDistance(e[k].from
 ,e[j].from));
    ek = new Edge(e[i].to,e[k].to,G.getDistance(e[i].to,e[k].
 to));
    ei.setRoute(r); ej.setRoute(r); ek.setRoute(r);
    copy[i] = ei;
    for(int t=1;t<k-j;t++) copy[i+t] = e[j+t];
    itr = i+k-j;
    copy[itr] = ej;
    reverseChain(i,j);
```

```
    for(int t=1;t<j-i;t++) copy[itr+t] = e[i+t];
    copy[k] = ek;
    e = copy;
    break;
case 3:
    ei = new Edge(e[i].from,e[k].from,G.getDistance(e[i].from
 ,e[k].from));
    ej = new Edge(e[j].to,e[i].to,G.getDistance(e[j].to,e[i].
 to));
    ek = new Edge(e[j].from,e[k].to,G.getDistance(e[j].from,e
 [k].to));
    ei.setRoute(r); ej.setRoute(r); ek.setRoute(r);
    copy[i] = ei;
    reverseChain(j,k);
    for(int t=1;t<k-j;t++) copy[i+t] = e[j+t];
    itr = i+k-j;
    copy[itr] = ej;
    for(int t=1;t<j-i;t++) copy[itr+t] = e[i+t];
    copy[k] = ek;
    e = copy;
    break;
}
  }

  /**
     reverse the edges between e[i] and e[j]
  */
  private void reverseChain(int i,int j) {
int n = j-i+1;
for(int t=i+1;t<j;t++) e[t].swap();
for(int t=1;t<n/2;t++) {
    Edge temp=e[i+t];
    e[i+t]=e[j-t];
    e[j-t]=temp;
}
  }


  ////////////////////////////////////////////
  // Time-Dependent Vehicle Routeing Problem
  ////////////////////////////////////////////

  public double getTourLength() {
double len = 0;
for(int i=0; i<MAX_ROUTES; i++)
```

```
    if(load[i]>0)
  len+= getRouteLength(i);
return len;
  }

  private boolean insideTrafficJamRegion(Edge e) {
return G.insideTrafficJamRegion(e);
  }

  //here the index starts from 0
  private double getRouteLength(int r) {
double d1 = getClockwiseRouteLength(r);
double d2 = getAntiClockwiseRouteLength(r);
return d1<d2?d1:d2;
  }

  private double getClockwiseRouteLength(int r) {
double stime = G.getTimeStart();
double jamTime = G.getJamTime();
double v = G.getVelocity();
double f = G.getJamFactor();
int st = -1;
for(int i=0; i<e.length; i++)
    if(e[i].getRoute()==r+1) {
  st = i;
  break;
    }
double ctime = stime;
double res = 0D;
boolean startJam = false;
for(int i=st;i<e.length;i++) {
    if(e[i].getRoute()!=r+1) break;
    double d = e[i].length();
    if(startJam) {
  if(G.withinTrafficJamRegion(e[i])){
      if(G.inside(e[i].from)) {
    if(G.inside(e[i].to))
        res += f*d;
    else {
        res += G.getPartialLength(e[i].from,e[i].to,d);
    }
      }
      else {
    res += G.getPartialLength(e[i].to,e[i].from,d);
      }
```

```
  }
  else res+=d;
    }
    else {
  double t = d/v;
  if(ctime+t < jamTime) {
      ctime+=t;
      res+=d;
  }
  else {
      startJam=true;
      if(G.withinTrafficJamRegion(e[i]))
    res+=G.getPartialLength(e[i].from,e[i].to,d,ctime,jamTime
 ,v);
      else res+=d;
  }
    }
}
return res;
  }

  private double getAntiClockwiseRouteLength(int r) {
double stime = G.getTimeStart();
double jamTime = G.getJamTime();
double v = G.getVelocity();
double f = G.getJamFactor();
int st = -1;
for(int i=e.length-1;i>=0;i--)
    if(e[i].getRoute()==r+1){
  st = i;
  break;
    }
double ctime = stime;
boolean startJam = false;
double res1 = 0D;
for(int i=st;i>=0;i--) {
    if(e[i].getRoute()!=r+1) break;
    double d = e[i].length();

    if(startJam) {
  if(G.withinTrafficJamRegion(e[i])){
      /*
      // NOTE: here we are computing the
      // route length in the revers
      // direction, so we need to be careful
```

```
      // the orientation. i.e. e[i].to-->e[i].from
      */
      if(G.inside(e[i].to)) {
    if(G.inside(e[i].from))
        res1 += f*d;
    else {
        res1 += G.getPartialLength(e[i].to,e[i].from,d);
    }
      }
      else {
    res1 += G.getPartialLength(e[i].from,e[i].to,d);
      }
  }
  else res1+=d;
    }
    else {
  double t = d/v;
  if(ctime+t < jamTime) {
      ctime+=t;
      res1+=d;
  }
  else {
      startJam=true;
      if(G.withinTrafficJamRegion(e[i]))
    res1+=G.getPartialLength(e[i].to,e[i].from,d,ctime,
 jamTime,v);
      else res1+=d;
  }
    }
}
return res1;
  }
  public void reverseRoute(int r) {
for(int i=0;i<e.length;i++) {
    if(e[i].getRoute()==r+1) e[i].swap();
}
//TD_checkDistance("reverse Route "+ r);
  }

  public Path TD_improve() {
int counter=1; int I = 30;
Path best_record = (Path)clone();
double deviation = 0.01 * best_record.getLength();
//int [] I={30,25,20,15,10,5};
while(true) {
```

```
    boolean improved = false;
    // Record r = new Record(best_record.getLength(),
 deviation);
    TD_checkDistance("before doing anything");
    for(int i=0;i<I;i++) {
 if(!TD_OPM_RRT(best_record.getTourLength(),deviation) &
    !TD_TPM_RRT(best_record.getTourLength(),deviation) &
    !TD_twOpt_RRT(best_record.getTourLength(),deviation)) {
      //if(!OPM_RRT(r) & !TPM_RRT(r) &!twOpt_RRT(r)) {
      System.out.println("no movement in I loop, quit I loop"
);
      break;
 }
 else{
      if(length<best_record.getTourLength()) {
    //System.out.println("[RRT]find a improving move,record
 is "+p.getLength());
    improved=true;
    best_record = (Path)clone();
    deviation=0.01*best_record.getTourLength();
    //   r = new Record(best_record.getLength(),deviation);
      }
 }
    } // I loop
    //       checkDistance();
    TD_clean();
    //checkDistance();
    if(length>best_record.getTourLength()) {
 //System.out.println("no improvement");
 improved = false;
    }
    else {
 if(Math.abs(length-best_record.getTourLength())>G.EPS) {
      improved = true;
      best_record = (Path) clone();
      deviation = 0.01*best_record.getTourLength();
      counter=0;
 }
    }
    counter++;
    //System.out.println("best record is "+ best_record.
 getLength());
    if(counter>5)
 break;
}
```

```
return best_record;
  }

  public boolean TD_OPM_RRT(double record, double deviation)
 {
//dump();
TD_checkDistance("enter OPM_RRT");
TD_checkLength("enter OPM_RRT");
boolean moved = false;
final int M=e.length;
for(int i=0;i<M;i++) {
    if(e[i].to>0 & e[i].to<G.size()) {
  boolean done_for_this_node = false;
  int q = G.nodes[e[i].to].getDemand();
  // the route which i belongs to
  int s = e[i].getRoute()-1;
  double dold = e[i].length() + e[(i+1)%M].length();
  double dnew = G.getDistance(e[i].from,e[(i+1)%M].to);
  double best_saving=-1e10; int best_j=-1;
  for(int j=(i+2)%M;j!=i;j=(j+1)%M) {
      if(e[i].inTheSameRoute(e[j])) {
    OPMUpdate(i,j);
    double ds = getRouteLength(s);
    double td_saving = dist[s] - ds;
    if(td_saving>0) {
        // update route length
        dist[s] = ds;
        length-= td_saving;
        //checkDistance();
        updatePosition();
        moved=true;
        done_for_this_node = true;
        break;
    }
    else {
        if(td_saving>best_saving & length-td_saving < record+
 deviation ) {
      best_saving = td_saving;
      best_j = j;
        }
        if(i<j) OPMUpdate(j-1,i);
        else OPMUpdate(j,(i+1)%M);
    }
    TD_checkDistance("leaving OPM_RRT_Same_route");
      }
```

```
      /*
       different routes
      */
      else {

    int r=e[j].getRoute()-1;
    if(load[r]+ q <= G.getCapacity()) {
        OPMUpdate(i,j);
        /*
         ignore distance constrant here
        */
        double dr = getRouteLength(r);
        double ds = getRouteLength(s);
        double td_saving = dist[r]+dist[s] - (dr+ds);
        if(td_saving>G.EPS) {
      length -= td_saving;
      dist[r] = dr; dist[s]=ds;
      load[r]+=q;
      load[s]-=q;
      /*
       * Note: assume Euclean distance so triangle inequality
 satisfies here
       */
      //checkDistance();
      updatePosition();
      moved = true;
      done_for_this_node=true;
      break;
        }
        else{
      if(td_saving>best_saving & length-td_saving<record+
deviation) {
          best_saving = td_saving;
          best_j = j;
      }
      if(i<j) OPMUpdate(j-1,i);
      else OPMUpdate(j,(i+1)%M);
      TD_checkDistance("Leaving different routes");
        }
    }
      }
 }
 if(done_for_this_node) continue;
 else {
      TD_checkDistance("enter best_j="+best_j);
```

```
        if(best_j!=-1) {
      if( e[i].inTheSameRoute(e[best_j]) ) {
          OPMUpdate(i,best_j);
          double ds = getRouteLength(s);
          updatePosition();
        /*
         update route length
        */
          length -= (dist[s]-ds);
          dist[s] = ds;
          moved = true;
      }
      else {
          int r=e[best_j].getRoute()-1;
        if(load[r]+q <= G.getCapacity()) {
            OPMUpdate(i,best_j);
            load[r]+=q;
            load[s]-=q;
            double ds = getRouteLength(s);
            double dr = getRouteLength(r);
            double td_saving = dist[s]+dist[r]-(ds+dr);
            length -= td_saving;
            dist[r]=dr; dist[s]=ds;
            //  System.out.println("OPM_RRT for "+e[i]+","+e[
  best_j]);
            //  checkDistance();
            updatePosition();
            moved = true;
        }
        else {
            System.out.println("error in TD_OPM_RRT");
            System.exit(1);
        }
      }

        }
    }
    TD_checkDistance("finish for i="+i);
      }
}
//  System.out.println("finish OPM_RRT");
//checkDistance();
//System.out.println("leave OPM_RRT");
TD_checkDistance("leave TD_OPM_RRT");
TD_checkLength("leave OPM_RRT");
```

```
return moved;
  }

  public boolean TD_TPM_RRT(double record, double deviation)
 {

TD_checkDistance("enter_TPM_RRT");
TD_checkLength("enter_TPM_RRT");
boolean moved = false;
final int M = e.length;
for(int i=0;i<M;i++) {
    if(e[i].to<G.N & e[i].to>0) {
  int q = G.nodes[e[i].to].getDemand();
  int r = e[i].getRoute()-1;
  boolean done_for_this_node=false;
  double best_saving = -1e10; int best_j = -1;
  boolean swap=false;
  for(int j=(i+1)%M;j!=i;j=(j+1)%M){
      int w = e[j].getRoute()-1;
      if( w!=r & (e[j].to<G.size() & e[j].to>0)){
    int s = G.nodes[e[j].to].getDemand();
    if((load[r]-q+s<=G.getCapacity()) & (load[w]-s+q<=G.
 getCapacity())) {
        TD_TPMSwap(i,j);
        double dr = getRouteLength(r); double dw =
 getRouteLength(w);
        double td_saving = dist[r]+dist[w] - dr - dw;
        if(td_saving>G.EPS) {
      length-=td_saving;
      dist[r]=dr; dist[w]=dw;
      load[r] = load[r]-q+s; load[w] = load[w]-s+q;
      done_for_this_node = true;
      moved = true;
      swap=true;
      updatePosition();
      break;
        }
        else{
      TD_TPMSwap(i,j);
      if(td_saving>best_saving & Math.abs(td_saving)>G.EPS) {
          best_saving = td_saving; best_j = j;
          swap=true;
      }
        }
    }
```

```
        }
    }
    if(done_for_this_node) continue;
    if(swap) {
        int j = best_j; int s = G.nodes[e[j].to].getDemand();
        int w = e[j].getRoute()-1;
        if(length-best_saving<record+deviation) {
      TD_TPMSwap(i,j);
      double dr = getRouteLength(r);
      double dw = getRouteLength(w);
      length -= best_saving;
      dist[r] = dr; dist[w] = dw;
      load[r]+=s-q;
      load[w]+=q-s;
      updatePosition();
      moved = true;
        }
    }
      }
}
TD_checkDistance("TPM_RRT");
TD_checkLength("leave_TPM_RRT");
return moved;
//  return false;
    }

    public boolean TD_twOpt_RRT(double record, double deviation
 ) {

//TD_checkDistance("enter 2OPT RRT");
//TD_checkLength("enter 2OPT RRT");
boolean moved=false;
int size = G.size() + G.numRoutes-1;
for(int i=0;i<size;i++) {
    double saving;
    boolean swap=false;
    boolean done_for_this_node=false;
    Edge e1 = e[i];
    Edge e2;
    int C1 = e1.from;
    int C2 = e1.to;
    double best_saving = -1e10;
    int best_j = -1;
    int r = e[i].getRoute()-1;
    for(int dir=0;dir<2;dir++) {
```

```
int id2;
if(dir==0)
    id2 = C2<G.size()?C2:0;
else
    id2 = C1<G.size()?C1:0;
for(int m=0;m<G.MAX_NB;m++) {
    int C3 = pos[G.NBList[id2][m]];
    int j=0;
    if(dir==0) {
  if(C3!=i+1) {
      if(C3>0) j=C3-1;
      else j=e.length-1;
  }
  if(j==i+1 | j==i-1) continue;
    }

    else {
  if(C3==i) continue;
  else j=C3;
  if(j==i+1|j==i-1) continue;
    }
    e2 = e[j];
    if(e1.inTheSameRoute(e2)) {
  if(i<j) TD_swapEdge(i,j);
  else TD_swapEdge(j,i);
  double dr = getRouteLength(r);
  double td_saving = dist[r]-dr;
  if(td_saving>G.EPS) {
      moved =true;
      done_for_this_node = true;
      length -=td_saving;
      dist[r] = dr;
      updatePosition();
      //TD_checkDistance("2OPT 1");
      //TD_checkLength("2OPT 1");
  }
  else {
      if(i<j) TD_swapEdge(i,j);
      else TD_swapEdge(j,i);
      //TD_checkDistance("2OPT 2");
      //TD_checkLength("2OPT 2");
      if(td_saving>best_saving) {
    best_saving = td_saving;
    best_j = j;
    swap=true;
```

```
        }
    }
      }
      // in different routes
      else {
    if(do2OPTBetweenRoutes) {
        int s = e[j].getRoute()-1;
        if(TD_checkFeasibility(Math.min(i,j),Math.max(i,j)))
{
      TD_swapEdge(Math.min(i,j),Math.max(i,j));
      double dr = getRouteLength(r);
      double ds = getRouteLength(s);
      double td_saving = dist[r] + dist[s] - (dr+ds);
      if(td_saving>G.EPS) {
          done_for_this_node = true;
          moved = true;
          length -= td_saving;
          dist[r] = dr; dist[s] = ds;
          //TD_checkDistance("2OPT 3");
          //TD_checkLength("2OPT 3");
      }
      else {
          TD_swapEdge(Math.min(i,j),Math.max(i,j));
          //TD_checkDistance("2OPT 4");
          //TD_checkLength("2OPT 4");
          if(td_saving>best_saving & Math.abs(td_saving)>G.
EPS) {
        best_saving = td_saving;
        best_j=j;
       swap=true;
          }
      }
        }
    }
      }
      if(done_for_this_node) break;
 }
 if(done_for_this_node) break;
   }
   if(done_for_this_node) continue;
   if(swap) {
 //TD_checkDistance("before enter 2OPT RRT between Routes");
 //TD_checkLength("before enter 2OPT RRT between Routes");
 e2 = e[best_j];
 if(length-best_saving<record+deviation) {
```

```
      //  TD_checkDistance("before enter 2OPT RRT between
 Routes");
      //TD_checkLength("before enter 2OPT RRT between Routes
 ");
      double dr=0D ;
      if(e1.inTheSameRoute(e2)) {
    TD_swapEdge(Math.min(i,best_j),Math.max(i,best_j));
    updatePosition();
    dr = getRouteLength(r);
    length-=(dist[r]-dr);
    dist[r] = dr;
    moved=true;
    //TD_checkDistance("2OPT 5");
    //TD_checkLength("2opt 5");
      }
      else {
    int s = e2.getRoute()-1;
    TD_swapEdge(Math.min(i,best_j),Math.max(i,best_j));
    updatePosition();
    dr = getRouteLength(r);
    double ds = getRouteLength(s);
    length-=(dist[r]+dist[s]-dr-ds);
    dist[r] = dr; dist[s]=ds;
    moved=true;
    //TD_checkDistance("2OPT 6");
    //TD_checkLength("2opt 6");
    //System.out.println("done 2-opt RRT between route "+(r
 +1)+" and route "+(s+1));
      }

  }
    }
}
//TD_checkDistance("leaving 2OPT RRT");
//TD_checkLength("leave 2OPT RRT");
return moved;
//return false;
  }

  public void TD_clean() {
TD_OPM();
TD_twOpt();
TD_TPM();
//TD_ThreeOPT();
  }
```

```
  public void TD_cleanup() {
double d1,d2,d3;
do {
    d1 = TD_OPM();
    d2 = TD_twOpt();
    d3 = TD_TPM();
} while(d1!=d2 | d2!=d3);
  }

  public double TD_OPM() {
while(TD_OPMStep()) TD_checkDistance("TD_OPMStep");
return length;
  }

  public double TD_TPM() {
while(TD_TPMStep()) TD_checkDistance("TD_TPMStep");
return length;
  }

  public double TD_twOpt() {
while(TD_twOptStep()) TD_checkDistance("TD_twOptStep");
return length;
  }

  private double TD_ThreeOPT() {
while(TD_ThreeOPTStep()) ;
return length;
  }

  public void TD_checkDistance(String where){
double len=0D;
for(int i=0;i<MAX_ROUTES;i++) {
    if(load[i]>0) {
  double r=getRouteLength(i);
  len+=r;
  if(dist[i]!=r) {
      System.out.println("///"+where+"//");
      System.out.println("distance inconsistent on route "+(i
 +1));
      System.out.println("recorded = "+dist[i]+",computed ="+
 r);
      dump();
      System.exit(1);
  }
```

```
  int cap = getRouteCapacity(i);
  if(load[i]!=cap | load[i]>G.getCapacity() ) {
      System.out.println("///"+where+"///");
      System.out.println("capacity constraint is violated on
 route "+(i+1));
      System.out.println("recorded="+load[i]+",computed="+cap
 );
      dump();
      System.exit(1);
  }
    }
}
  }

  private int getRouteCapacity(int r) {
int cap=0;
for(int i=0;i<e.length;i++) {
    if(e[i].getRoute()==r+1) {
  if(e[i].from<G.size()) cap+=G.nodes[e[i].from].getDemand();
    }
}
return cap;
  }

  /*
   change distance for each route for TDVRP
  */
  public void TD_Path() {
double len = 0D;
for(int i=0;i<MAX_ROUTES;i++) {
    if(load[i]>0){
  dist[i] = getRouteLength(i);
  len+=dist[i];
    }
}
length = len;
  }

  private boolean TD_OPMStep() {
//  debug = true;
//TD_checkDistance("enter OPMStep");
//TD_checkLength("enter OPMStep");
boolean improved = false;
int M=e.length;
for(int i=0;i<M;i++) {
```

```
   if(e[i].to>0 & e[i].to<G.size()) {
 int q = G.nodes[e[i].to].getDemand();
 int s = e[i].getRoute()-1;
 double dold = e[i].length() + e[(i+1)].length();
 double dnew = G.getDistance(e[i].from,e[i+1].to);
 for(int j=(i+2)%M;j!=i;j=(j+1)%M) {
     double d0=dold+e[j].length();
     double d1=dnew+G.getDistance(e[i].to,e[j].from)+G.
getDistance(e[i].to,e[j].to);
     double saving = d0-d1;
     if(saving>0 & Math.abs(saving)>G.EPS) {
   if( e[i].inTheSameRoute(e[j]) ) {
       if(saving>0 & Math.abs(saving)>G.EPS) {
      OPMUpdate(i,j);
      double newdist = getRouteLength(s);
      if(newdist<dist[s]){
          length-= dist[s]-newdist;
          // update the route length
          dist[s] = newdist;
          improved = true;
          //TD_checkDistance("OPM 1");
      }
      else {
          if(i<j)
        OPMUpdate(j-1,i); //undo the move
          else OPMUpdate(j,(i+1)%M);
          //TD_checkDistance("OPM 2");
      }
        }

        else {
      OPMUpdate(i,j);
      double newdist = getRouteLength(s);
      if(newdist<dist[s]) {
          length -= dist[s]-newdist;
          dist[s] = newdist;
          improved = true;
          //TD_checkDistance("OPM 3");
      }
      else {
          if(i<j) OPMUpdate(j-1,i);
          else OPMUpdate(j,(i+1)%M);
          //TD_checkDistance("OPM 4");
      }
        }
```

```
        //TD_checkDistance("leave OPM within route");
    }
    //different routes
    else {
        //TD_checkDistance("enter OPM between route");
        int r=e[j].getRoute()-1;
        if(load[r]+q <= G.getCapacity()) {
      //double dlen = d1-dnew-e[j].length();
      //check the distance constraint
      if(saving>0 & Math.abs(saving)>G.EPS) {
          OPMUpdate(i,j);
          double ds = getRouteLength(s); double dr =
getRouteLength(r);
          if(ds+dr < dist[r]+dist[s]) {
        length += (ds+dr)-(dist[r]+dist[s]);
        load[r]+=q;
        load[s]-=q;
        dist[r] = dr;
        dist[s] = ds;
        improved = true;
          }
          else {
        if(i<j) OPMUpdate(j-1,i);
        else OPMUpdate(j,(i+1)%M);
          }
      }
      else {
          OPMUpdate(i,j);
          double ds = getRouteLength(s); double dr =
getRouteLength(r);
          saving = dist[r]+dist[s] - ds - dr;
          if(saving>0) {
        length -= saving;
        dist[s] = ds;
        dist[r] = dr;
        load[r] +=q;
        load[s] -=q;
        improved = true;
          }
          else {
        if(i<j) OPMUpdate(j-1,i);
        else OPMUpdate(j,(i+1)%M);
          }
      }
        }
```

```
    }
      }
      if(improved) {
    updatePosition();
    break;
      }
  }
    }
    if(improved) break;
}
TD_checkDistance("leave OPMStep");
TD_checkLength("Leave OPMStep");
return improved;
  }

  private void TD_checkLength(String where) {
double len = getTourLength();
if(Math.abs(length-len)>1e-4) {
    System.out.println("////////////////"+where+"
 ////////////////");
    System.out.println("recorded="+length+",computed="+len);
    System.exit(1);
}
  }

  private boolean TD_twOptStep() {
boolean improved = false;
for(int i=0;i<e.length;i++) {
    improved = TD_seek(i,0);
    TD_checkDistance("after TDSeek "+i+",0");
    if(!improved) improved=TD_seek(i,1);
    TD_checkDistance("after TDSeek "+i+",1");
    if(improved) break;
}
return improved;
  }
  /*
  just check capacity, adding max route length later
  */
  private boolean TD_checkFeasibility(int i,int j) {
int r1 = e[i].getRoute()-1;
int r2 = e[j].getRoute()-1;
int p1=0; int p2=0;
int k=i+1;
final int M = e.length;
```

```
while(k<M & e[k].getRoute()==r1+1){
    p1+=G.nodes[e[k].from].getDemand();
    k++;
}
k=j-1;
while(k>=0 & e[k].getRoute()==r2+1){
    if(e[k].to<G.size()) {
  p2+=G.nodes[e[k].to].getDemand();
  k--;
    }
    else {
  System.out.println(e[k]);
  System.exit(1);
    }
}
int n1 = load[r1]-p1+p2;
int n2 = load[r2]-p2+p1;
if(n1<=G.getCapacity() & n2<=G.getCapacity()){
    return true;
}
else return false;
  }

  // i<j
  public void TD_swapEdge(int i,int j) {
int r1 = e[i].getRoute(); int r2 = e[j].getRoute();
Edge e1 = new Edge(e[i].from,e[j].from,G.getDistance(e[i].
 from,e[j].from));
Edge e2 = new Edge(e[i].to,e[j].to, G.getDistance(e[i].to,e[j
 ].to));
e1.setRoute(r1); e2.setRoute(r2);
int p1=0; int p2=0;
if(r1!=r2) {
    int k=i+1;
    while(k<e.length & e[k].getRoute()==r1){
  e[k].setRoute(r2);
  p1+=G.nodes[e[k].from].getDemand();
  k++;
    }
    k=j-1;
    while(k>=0 & e[k].to<G.size()){
  e[k].setRoute(r1);
  p2+=G.nodes[e[k].to].getDemand();
  k--;
    }
```

```
    load[r1-1] = load[r1-1]-p1+p2;
    load[r2-1] = load[r2-1]-p2+p1;
}
e[i] = e1;e[j] = e2;
for(int k=i+1;k<j;k++) e[k].swap();
int n=j-i-1;
for(int t=1;t<=n/2;t++) {
    Edge temp=e[i+t];
    e[i+t]=e[j-t];
    e[j-t]=temp;
}
  }

  public boolean TD_seek(int i, int dir) {
boolean improved = false;
Edge e1 = e[i];
Edge e2 = null;
int C1 = e1.from;
int C2 = e1.to;
int id2;
if(dir==0)
    id2 = C2<G.size()?C2:0;
else
    id2 = C1<G.size()?C1:0;
for(int m=0;m<G.MAX_NB;m++) {
    int C3 = pos[G.NBList[id2][m]]; int C4=-1;
    int j=0;
    if(dir==0) {
  if(C3!=i+1) {
      if(C3>0) j=C3-1;
      else j=e.length-1;
      C4=e[j].from;
  }
  if(j==i+1 | j==i-1) continue;
    }
    else {
  if(C3==i) continue;
  else j=C3;
  if(j==i+1 | j==i-1) continue;
  C4=e[j].to;
    }
    e2 = e[j];
    double d12 = e1.length();
    double d23;
    if(dir==0)
```

```
d23 = G.getDistance(e1.to,e2.to);
  else
d23 = G.getDistance(e1.from,e2.from);
  {
double d14;
if(dir==0)
    d14 = G.getDistance(e1.from,e2.from);
else
    d14 = G.getDistance(e1.to,e2.to);
double d34 = e2.length();
double saving = d12+d34-d23-d14;
if(true) {
    int r = e[i].getRoute()-1;
    if(e1.inTheSameRoute(e2)) {
  if(saving>G.EPS) {
      if(i<j)
    TD_swapEdge(i,j);
      else
    TD_swapEdge(j,i);
      double newdist = getRouteLength(r);
      double td_saving = dist[r] - newdist;
      if(td_saving>G.EPS) {
    improved = true;
    length-=td_saving;
    /*
       update the length for that route
    */
    dist[r] = newdist;
      }
      else {
    if(i<j) TD_swapEdge(i,j);
    else TD_swapEdge(j,i);
    TD_checkDistance(new String("i="+i+",j="+j));
      }
  }
  else {
      if(i<j) TD_swapEdge(i,j);
      else TD_swapEdge(j,i);
      double newdist = getRouteLength(r);
      double td_saving = dist[r] - newdist;
      if(td_saving>G.EPS) {
    improved = true;
    length -= td_saving;
    dist[r] = newdist;
      }
```

```
        else {
      if(i<j) TD_swapEdge(i,j);
      else TD_swapEdge(j,i);
      TD_checkDistance(new String("i="+i+",j="+j));
        }
    }
      }
      else {
    if(do2OPTBetweenRoutes) {
        int s = e[j].getRoute()-1;
        if(i<j) {
      if(TD_checkFeasibility(i,j)){
          TD_swapEdge(i,j);
          double dr=getRouteLength(r);
          double ds = getRouteLength(s);
          double td_saving = dist[r]+dist[s] - (dr+ds);
          if(saving>G.EPS) {
        if(td_saving>G.EPS) {
            improved=true;
            length-=td_saving;
            dist[r] = dr; dist[s] = ds;
        }
        else {
            TD_swapEdge(i,j);
        }
          }
          else {
        if(td_saving>G.EPS) {
            improved = true;
            length -= td_saving;
            dist[r] = dr; dist[s] = ds;
        }
        else TD_swapEdge(i,j);

          }
      }
        }
        else{ //j<i
      if(TD_checkFeasibility(j,i)){
          TD_swapEdge(j,i);
          double dr=getRouteLength(r); double ds =
getRouteLength(s);
          double td_saving = dist[r]+dist[s] - (dr+ds);
          if(saving>G.EPS) {
        if(td_saving>G.EPS) {
```

```
                improved=true;
                length-=td_saving;
                dist[r] = dr; dist[s] = ds;
            }
            else {
                TD_swapEdge(j,i);
            }
              }
              else {
            if(td_saving>G.EPS) {
                improved = true;
                length -= td_saving;
                dist[r] = dr; dist[s] = ds;
            }
            else TD_swapEdge(j,i);

              }
          }
            }
        }
          }
      }
    }
    if(improved) {
  updatePosition();
  break;
    }
}

TD_checkLength("leave_2opt_seek");
return improved;
  }

  private boolean TD_ThreeOPTStep() {
return true;
  }

  private void TD_TPMSwap(int i,int j) {
int M = e.length;
int key1 = e[i].to; int key2= e[j].to;
e[i].set(e[i].from, key2 ,G.getDistance(e[i].from,key2));
e[(i+1)%M].set(key2,e[(i+1)%M].to,G.getDistance(key2,e[(i+1)%
 M].to));
e[j].set(e[j].from,key1,G.getDistance(e[j].from,key1));
e[(j+1)%M].set(key1,e[(j+1)%M].to,G.getDistance(key1,e[(j+1)%
```

```
 M].to));
  }

  public boolean TD_TPMStep() {
boolean improved = false;
final int M = e.length;
for(int i=0;i<M;i++) {
    if(e[i].to<G.N & e[i].to>0) {
  int q = G.nodes[e[i].to].getDemand();
  int r = e[i].getRoute()-1;
  double do1 = e[i].length() + e[(i+1)%M].length();
  for(int j=(i+1)%M;j!=i;j=(j+1)%M){
      int w = e[j].getRoute()-1;
      if( w!=r & (e[j].to<G.size() & e[j].to>0)){
    double do2 =  e[j].length()+e[(j+1)%M].length();
    double dold = do1+do2;
    int s = G.nodes[e[j].to].getDemand();
    if(load[r]-q+s<=G.getCapacity() & load[w]-s+q<=G.
 getCapacity()) {
        TD_TPMSwap(i,j);
        double dr = getRouteLength(r);
        double dw = getRouteLength(w);
        double td_saving = dist[r] + dist[w] - (dr+dw);
        if(td_saving>G.EPS) {
      length -= td_saving;
      dist[r] = dr; dist[w] = dw;
      load[r]-=q-s; load[w] -=s-q;
      improved = true;
      updatePosition();
        }
        else {
      TD_TPMSwap(i,j);
        }
    }
      }
      if(improved) break;
  }
    }
    if(improved) break;
}
return improved;
  }

}
```

A.7 VRPInstance.java

```
import java.util.*;
/**
   a VRPInstance is always associated with a VRPGraph
*/
public class VRPInstance {
    public VRPGraph G;
    public Path bestTour;
    boolean purturb;
    public VRPInstance(VRPGraph g) {
  G = g;
    }

    public void debug() {
  Path p = G.CWParallel(0.6);
  p.TD_Path();
  System.out.println("[clark wright] tour length="+p.getLength
   ());
  //p.dump();
  //p.dumpPath();
  //p.TD_OPM();
  double record = p.getLength(); double deviation = 0.01*record
   ;
  for(int i=0;i<10;i++) {
      //p.TD_OPM_RRT(record,deviation);
      //p.TD_TPM_RRT(record,deviation);
      p.TD_twOpt_RRT(record,deviation);
  }
  for(int i=0;i<2;i++) {
      p.TD_OPM();
      p.TD_twOpt();
      p.TD_TPM();
  }
  //p.TD_TPM();
  //p.TD_seek(0,1);
  //p.TD_checkDistance("");
  //  p.dump();
  //p.dumpPath();
  //  p.TD_swapEdge(0,8);
  //p.TD_checkDistance("");
  //p.dump();
  p.dumpPath();
  System.out.println(p.getLength()+"==?"+p.getTourLength());
    }
```

```
  public void displayLineEquation(Path p) {
G.setJamFactor(1.0);
double d0 = p.getTourLength();
G.setJamFactor(2.0);
double d1 = p.getTourLength();
double slope = d1-d0;
System.out.println("line equation="+d0+"+"+slope+"*(f-1)");
  }


  public static void main(String args[]) throws Exception {
long begin = System.currentTimeMillis();
VRPFileReader vr = new VRPFileReader(args[0]);
if(args.length>1) {
    System.out.println("cut prob="+Double.parseDouble(args
 [1]));
    VRPGraph.beta = Double.parseDouble(args[1]);
    //VRPGraph.setCutProb(Double.parseDouble(args[1]));
}
VRPGraph g = vr.readData();
//if(args.length>1) g.setJamFactor(Double.parseDouble(args
 [1]));
//   System.out.println("////////////////"+g.size()
 +"////////////////");
//System.out.println(g);

VRPInstance vrp = new VRPInstance(g);
vrp.purturb = Boolean.getBoolean("pt");


//vrp.lk();
//long end = System.currentTimeMillis();
//System.out.println("total time is "+ (end-begin)/1e3 +"
 secs");
//System.out.println("********** beta="+Path.beta
 +"**********");
begin = System.currentTimeMillis();
// uncomment out this if you want to solve VRP
Path best = vrp.RRT();
// uncomment out this if you want to solve TDVRP
// this is for Time-dependent VRP
//Path best = vrp.TD_RRT();
//System.out.println(best);
best.dumpPath();
```

```
long end = System.currentTimeMillis();
System.out.println("total time for RRT is "+ (end-begin)/6e4
 + " minutes");
//System.out.println("best tour ="+best.getLength());
  }

  public void lk() {
Path p = G.CWParallel(1.0);
System.out.println("clark and wright "+p.length);
//p = p.improve();
p.init_tour(G.psize());
long start = System.currentTimeMillis();
p.CVRP_Clean();
long end = System.currentTimeMillis();
//p.lk_dumpPath();
System.out.println("after 2opt tour length="+p.lk_tourLength
 ()+"("+p.length+")");
System.out.println("running time for CVRP_Clean is "+ (end-
 start)/1e3 +" secs");

Path p1 = G.CWParallel(1.0);
start =  System.currentTimeMillis();
p1.clean();
end = System.currentTimeMillis();
System.out.println("after 2opt tour length="+p1.getLength());
System.out.println("running time for CVRP_Clean is "+ (end-
 start)/1e3 +" secs");

  }

  public Path TD_RRT() {
double best_length=1e10;
Path best_tour = null;
//double [] lamda = {0.6,1.4,1.6};
for(double lamda=0.6;lamda<2.0;lamda+=0.2) {
    //if(lamda==0.6|lamda==1.4|lamda==1.6) continue;
    //for(int i=0;i<3;i++) {
    Path p = G.CWParallel(lamda);
    p.TD_Path();
    //p.TD_clean();
    //      System.out.println("[clark wright]"+p+","+lamda);
    Path best_record = p.TD_improve();
    best_record.TD_cleanup();
    System.out.println("[lamda="+lamda+"]"+best_record);
    if(best_record.getLength()< best_length) {
```

```
  best_tour = (Path) best_record.clone();
  best_length = best_record.getLength();
    }
    if(purturb) {
  best_record.purturb();
  Path imp = best_record.improve();
  System.out.println("after purturbation "+imp);
  if(imp.getLength()<best_length) {
      best_tour = (Path) imp.clone();
      best_length = imp.getLength();

  }
    }
}
return best_tour;
  }

  public Path RRT() {
double best_length=1e10;
Path best_tour = null;
double [] lamda = {0.6,0.8,1.0,1.2,1.4,1.6,1.8};
//for(double lamda=0.4;lamda<2.0;lamda+=0.2) {
    //if(lamda==0.6|lamda==1.4|lamda==1.6) continue;
    for(int i=0;i<lamda.length;i++) {
  Path p = G.CWParallel(lamda[i]);
  //System.out.println("there are "+p.perc()+" % edges rem:
 ain");
  //System.out.println("[clark wright]"+p+","+lamda[i]);
  Path best_record = p.improve();
  //best_record.cleanUp();
  System.out.println("[lamda="+lamda[i]+"]"+best_record);
  //System.out.println("there are "+best_record.perc()+" %
 edges remain");
  if(best_record.getLength()< best_length) {
      best_tour = (Path) best_record.clone();
      best_length = best_record.getLength();
  }
  if(purturb) {
      int count=0;
      while(count++<10) {
    best_record.purturb();
    Path imp = best_record.improve();
      //System.out.println("after purturbation "+imp);
    if(imp.getLength()<best_length) {
        best_tour = (Path) imp.clone();
```

```
            best_length = imp.getLength();
            count=0;
        }
        best_record = (Path) best_tour.clone();
          }
    }
        }
return best_tour;
    }
    /*
    public void improve(Path p) {
int counter=1; int I = 30;
Path best_record = (Path)clone();
double deviation = 0.01 * best_record.getLength();
while(true) {
        boolean improved = false;
        for(int i=0;i<I;i++) {
     if(!OPM_RRT(best_record.getLength(),deviation) & !TPM_RRT(
    best_record.getLength(),deviation) &!twOpt_RRT(best_record.
    getLength(),deviation)) {
            System.out.println("no movement in I loop, quit I loop
    ");
            break;
    }
     else{
            if(length<best_record.getLength()) {
        //System.out.println("[RRT]find a improving move,record
    is "+p.getLength());
        improved=true;
        best_record = (Path)clone();
        deviation=0.01*best_record.getLength();
          }
    }
        } // I loop
        clean();
        if(length>best_record.getLength()) {
    //System.out.println("no improvement");
    improved = false;
        }
        else {
    if(Math.abs(length-best_record.getLength())>G.EPS) {
            improved = true;
            best_record = (Path) clone();
            deviation = 0.01*best_record.getLength();
            counter=0;
```

```
    }
      }
      counter++;
      //System.out.println("best record is "+ best_record.
   getLength());
      if(counter>5)
    break;
  }
  return best_record;
    }
    */
    public Path RRT_Purturb() {
  double best_length = 1e10;
  Path best_tour = null;
  Path p = G.CWParallel(1.6);
  Path best_record = p.improve();
  best_record.clean();
  best_tour = (Path) best_record.clone();
  System.out.println("best solution "+best_tour);
  best_record.purturb();
  int counter=1;
  while(true) {
      Path imp = best_record.improve();
      if(imp.getLength()>best_tour.getLength()) break;
      else {
    best_tour = (Path) imp.clone();
    System.out.println("new solution " + best_tour);
          best_record =imp;
    best_record.purturb();
      }
  }
  return best_tour;
    }


}
```

Appendix B

Noisy Traveling Salesman Problem Code

B.1 NTSPGraph.java

```
/*
 *noisy tsp graph
 * generate points with mean uniformly distributed on circle
 * with radius r, and std=1
 */
import java.util.Random;
import java.io.*;
import java.awt.*;

public class NTSPGraph extends Graph implements Cloneable{
    static int seq=0;

    double mean[][];
    double dev ;
    double x[],y[];
    double std_sigma=0D;
    double prob_p=1.0;
    int fixCount = 0;

    // optimal tour for fixed nodes
    int fixTour[];

    static Random rand = new Random(1234567890);
    Random mean_rnd = new Random(1234567);
    boolean change[];

    public NTSPInstance own;

    public void setDeviation(double d)
```

```
  {
dev = d;
  }
  public double getDeviation()
  {
return dev;
  }

  public void setProb(double p)
  {
prob_p = p;
  }

  public double getProb()
  {
return prob_p;
  }


  public Object clone()
  {
NTSPGraph t = null;
t = (NTSPGraph) super.clone();
int n = size();
t.mean = new double[n][2];
t.x = new double[n];
t.y = new double[n];
for(int i=0;i<n;i++) {
    t.mean[i][0] = mean[i][0];
    t.mean[i][1] = mean[i][1];
}
System.arraycopy(x,0,t.x,0,n);
System.arraycopy(y,0,t.y,0,n);
return t;
  }


  /*
   * test instance 1
   * means are distributed around unit circle
   */

  /**
   * Creates a new <code>NTSPGraph</code> instance.
   * Data set 1, means are located evenly on circle with
```

```
 radius r.
   * @param r the radius of the circle
   * @param n problem size
   * @param deviation deviation of the normal distribution
   */
  public NTSPGraph(double r,int n,double deviation) {
super(n);
mean = new double[n][2];
x = new double[n];
y = new double[n];
final double alpha = 2*Math.PI/n;
for(int i=0;i<n;i++) {
    double theta = alpha*i;
    mean[i][0] = r*Math.cos(theta);
    mean[i][1] = r*Math.sin(theta);
}
dev = deviation;
double sigma = Math.sqrt(dev);
      for(int i=0;i<n;i++) {
    x[i] = mean[i][0] + sigma * rand.nextGaussian();
    y[i] = mean[i][1] + sigma * rand.nextGaussian();
}

own = new NTSPInstance(x,y);
for(int i=0;i<n;i++)
    for(int j=i+1;j<n;j++) {
  double d = Math.sqrt( (x[i]-x[j])*(x[i]-x[j]) + (y[i]-y[j])
 *(y[i]-y[j]) );
  connect(i,j,d);
  connect(j,i,d);
    }
change = new boolean[n];
for(int i=0;i<n;i++) change[i] = true;
System.out.println("test case 1, dev="+deviation);
  }


  /**
   * Creates a new <code>NTSPGraph</code> instance.
   *
   * @param r an <code> radius of the circle </code>
   * @param n an <code>int</code> # points on the circle
   * @param deviation a <code>double</code> deviation of
 Gaussian noise
   * @param p a <code>double</code> probablility of adding
```

```
 Gaussian noise
   */
  public NTSPGraph(double r,int n,double deviation,double p)
 {
super(n);
mean = new double[n][2];
x = new double[n];
y = new double[n];
change = new boolean[n];
dev = deviation;
prob_p = p;
final double alpha = 2*Math.PI/n;
for(int i=0;i<n;i++) {
    double theta = alpha*i;
    mean[i][0] = r*Math.cos(theta);
    mean[i][1] = r*Math.sin(theta);
}
double sigma = Math.sqrt(dev);
Random rnd = new Random();
for(int i=0;i<n;i++) {
    if(rnd.nextDouble() < p) {
  change[i] = true;
    }
    else {
  change[i] = false;
    }
}

      for(int i=0;i<n;i++) {
    if(change[i]){
  x[i] = mean[i][0] + sigma * rand.nextGaussian();
  y[i] = mean[i][1] + sigma * rand.nextGaussian();
    }
    else {
  x[i] = mean[i][0];
  y[i] = mean[i][1];
    }
}
own = new NTSPInstance(x,y);
for(int i=0;i<n;i++)
    for(int j=i+1;j<n;j++) {
  double d = Math.sqrt( (x[i]-x[j])*(x[i]-x[j]) + (y[i]-y[j])
 *(y[i]-y[j]) );
  connect(i,j,d);
  connect(j,i,d);
```

```
    }
System.out.println("test 1: deviation="+dev+",prob="+p);
  }

  /*
   * test instance 2 two hierachical instance
   * the means are located on circle with radius r
   * for each point, m points are clustered around it with N
 (0,sigma^2)
   */

  /**
   * Creates a new <code>NTSPGraph</code> instance.
   *
   * @param r a <code>double</code> value
   * @param n an <code>int</code> value
   * @param r1 a <code>double</code> value
   * @param m an <code>int</code> value
   * @param deviation a <code>double</code> value
   * @param p a <code>double</code> value
   */
  public NTSPGraph(double r,int n,double r1,int m,double
 deviation,double p)
  {
super(m*n);
System.out.println("test case 2,n="+n+",m="+m);
mean = new double[n*m][2];
change = new boolean[n*m];
final double alpha = 2*Math.PI/n;
double m1[][];
m1 = new double[n][2];
for(int i=0;i<n;i++) {
    m1[i][0] = r*Math.cos(i*alpha);
    m1[i][1] = r*Math.sin(i*alpha);
}


x = new double[n*m];
y = new double[n*m];
final double beta = 2*Math.PI/m;
    for(int i=0;i<n;i++) {
    for(int j=0;j<m;j++) {
  mean[i*m+j][0] = m1[i][0] + r1*Math.cos(j*beta);
  mean[i*m+j][1] = m1[i][1] + r1*Math.sin(j*beta);
    }
```

```
}
dev = deviation;
prob_p = p;
final double sigma = Math.sqrt(dev);
Random rnd = new Random();
for(int i=0;i<m*n;i++) {
    if(rnd.nextDouble()<p) change[i] = true;
    else change[i] = false;
}

for(int i=0;i<n;i++) {
    for(int j=0;j<m;j++) {
  if(change[i*m+j]) {
      x[i*m+j] = mean[i*m+j][0] + sigma*rand.nextGaussian();
      y[i*m+j] = mean[i*m+j][1] + sigma*rand.nextGaussian();
  }
  else {
      x[i*m+j] = mean[i*m+j][0];
      y[i*m+j] = mean[i*m+j][1];
  }
    }
}
for(int i=0;i<n*m;i++)
    for(int j=i+1;j<n*m;j++) {
  double d = Math.sqrt( (x[i]-x[j])*(x[i]-x[j]) + (y[i]-y[j])
 *(y[i]-y[j]) );
  connect(i,j,d);
  connect(j,i,d);
    }
own = new NTSPInstance(x,y);
  }


  private void getMeanPoints(int n,double std_sigma)
  {
mean = new double[n][2];

Random meanRnd = new Random(1234567);
for (int  i = 0; i <n ; i++) {
    mean[i][0] = std_sigma*meanRnd.nextGaussian();
    mean[i][1] = std_sigma*meanRnd.nextGaussian();
} // end of for (int  = 0;  < ; ++)
  }

  /*
```

```
   * data set 3
   */

  public NTSPGraph(double r,int n,double std_sigma1,double
 deviation,double p)
  {
super(n);
std_sigma=std_sigma1;
dev = deviation;
prob_p = p;
x = new double[n];
y = new double[n];
change = new boolean[n];
// make sure they have the same mean points
getMeanPoints(n,std_sigma1);

double sigma = Math.sqrt(dev);

System.out.println("#test case 3 with p="+prob_p);
Random rnd = new Random();
for(int i=0;i<n; i++) {
    if(rnd.nextDouble()<prob_p) {
  change[i] = true;
    }
    else change[i] = false;
}

fixCount=0;
for (int  i = 0; i < n;i ++) {
    if(change[i]) {
  x[i] = mean[i][0] + sigma*rand.nextGaussian();
  y[i] = mean[i][1] + sigma*rand.nextGaussian();
    }
    else {
      fixCount++;
  x[i] = mean[i][0];
  y[i] = mean[i][1];
    }
} // end of for (int  = 0;  < ; ++)
System.out.println("#there are "+fixCount+" nodes remain
 unchaged");

/*
 * added on 5/27/03
 * the main idea is to compute a good/optimal tour for those
```

```
 nodes
 * fixed so we can use it in the new instance
 */
fixTour = new int[fixCount];
double fix_x[] = new double[fixCount];
double fix_y[] = new double[fixCount];
int curPos = 0;
for(int i=0; i<n; i++) {
  if(!change[i]) {
    fix_x[curPos] = x[i];
    fix_y[curPos] = y[i];
    fixTour[curPos++]=i;
  }
}

System.out.println("#"+curPos + " nodes remain unchanged");
int [] opt = new int[fixCount];
int [] ind = new int[fixCount];
for(int i=0;i<fixCount;i++) ind[i] = i;
System.arraycopy(fixTour,0,opt,0,fixCount);
if(fixCount>0)
    ind = MyORUtil.getOPT(fix_x,fix_y);
for(int i=0;i<fixCount;i++)
    fixTour[i] = opt[ind[i]];
for (int  i = 0; i < n ; i++) {
    for (int  j = i + 1; j <n ; j++) {
  double d = Math.sqrt( (x[i]-x[j])*(x[i]-x[j]) + (y[i]-y[j])
 *(y[i]-y[j]) );
  connect(i,j,d);
  connect(j,i,d);
    } // end of for (int  = 0;  < ; ++)

} // end of for (int  = 0;  < ; ++)
own = new NTSPInstance(x,y,fixCount,change,fixTour);
  }

  /*
   * test instance 4, all the means are distributed uniformly
  in
   * [0,r]x[0,r]
   */
  public NTSPGraph(int n,double r,double deviation,double p)
  {
super(n);
dev = deviation;
```

```
prob_p = p;
x = new double[n];
y = new double[n];
change = new boolean[n];
// make sure they have the same mean points
//getMeanPoints(n, std_sigma1);
mean = new double[n][2];
for(int i=0;i<n;i++) {
    mean[i][0] = r*rand.nextDouble();
    mean[i][1] = r*rand.nextDouble();
}

double sigma = Math.sqrt(dev);

System.out.println("test case 4 with p="+prob_p);
Random rnd = new Random();
for(int i=0;i<n; i++) {
    if(rnd.nextDouble()<prob_p) {
  change[i] = true;
    }
    else change[i] = false;
}

fixCount=0;
for (int  i = 0; i < n;i ++) {
    if(change[i]) {
  x[i] = mean[i][0] + sigma*rand.nextGaussian();
  y[i] = mean[i][1] + sigma*rand.nextGaussian();
    }
    else {
      fixCount++;
  x[i] = mean[i][0];
  y[i] = mean[i][1];
    }
} // end of for (int  = 0; < ; ++)
System.out.println("there are "+fixCount+" nodes remain
 unchaged");

/*
 * added on 5/27/03
 * the main idea is to compute a good/optimal tour for those
 nodes
 * fixed so we can use it in the new instance
 */
fixTour = new int[fixCount];
```

```
double fix_x[] = new double[fixCount];
double fix_y[] = new double[fixCount];
int curPos = 0;
for(int i=0; i<n; i++) {
  if(!change[i]) {
    fix_x[curPos] = x[i];
    fix_y[curPos] = y[i];
    fixTour[curPos++]=i;
  }
}

System.out.println(curPos + " nodes remain unchanged");
// compute the optimal tour for thoese nodes
int [] opt = new int[fixCount];
int [] ind = new int[fixCount];
for(int i=0;i<fixCount;i++) ind[i] = i;
System.arraycopy(fixTour,0,opt,0,fixCount);
if(fixCount>0)
    ind = MyORUtil.getOPT(fix_x,fix_y);
for(int i=0;i<fixCount;i++)
    fixTour[i] = opt[ind[i]];
for (int  i = 0; i < n ; i++) {
    for (int  j = i + 1; j <n ; j++) {
  double d = Math.sqrt( (x[i]-x[j])*(x[i]-x[j]) + (y[i]-y[j])
 *(y[i]-y[j]) );
  connect(i,j,d);
  connect(j,i,d);
    } // end of for (int  = 0;  < ; ++)

} // end of for (int  = 0;  < ; ++)
own = new NTSPInstance(x,y,fixCount,change,fixTour);
  }

  /**
   * generate a new TSP instance.
   * The new instance is constructed by sampling from a
 normal distribution around each mean point.
   * @return a new <code>NTSPInstance</code>
   */
  public NTSPInstance getInstance() {
double [] nx;
double [] ny;
int n = size();
nx = new double[n];
ny = new double[n];
```

```
final double sigma = Math.sqrt(dev);

for(int i=0; i<n; i++) {
    if(!change[i]) {
  nx[i] = mean[i][0];
  ny[i] = mean[i][1];
    }
    else {
  nx[i] = mean[i][0] + sigma*rand.nextGaussian();
  ny[i] = mean[i][1] + sigma*rand.nextGaussian();
    }
}
NTSPInstance ins = new NTSPInstance(nx,ny,fixCount,change,
 fixTour);
return ins;
  }
  /*
   * nearest neighbor insertion heuristic
   */

  /**
   * Nearest Insertion heuristic for TSP
   *
   * @return the TSP tour
   */
  public int[] nnInsertion() {
int [] tour;
int n = size();
tour = new int[n];
boolean flag[] = new boolean[n];
for(int i=0;i<n;i++) flag[i] = false;
int j=0;
flag[j] = true;
for(int i=0;i<n-1;i++) {
    double d = 1e10;
    int l=0;
    for(int k=0;k<n;k++) {
  if(!flag[k] && distance(j,k)<d) {
      d = distance(j,k);
      l = k;
  }
    }
    tour[i] = j;
    tour[i+1] = l;
```

```
    flag[l] = true;
    j = l;
}
return tour;
  }


  /**
   * write the TSP instance into a file with the format n-m.
 ntsp
   * where n is # of nodes, and m is increased by one
 automatically.
   */
  public void dump() {
PrintWriter out=null;
try {
    FileWriter fos = new FileWriter(x.length+"-"+(seq++)+".
 ntsp",false);
    out = new PrintWriter(new BufferedWriter(fos));
} catch(FileNotFoundException e) {}
catch(IOException e) {}
out.println(x.length);
for(int i=1;i<x.length;i++)
    out.println(i+"\t"+x[i]+"\t"+y[i]);
out.println("0\t"+x[0]+"\t"+y[0]);
out.close();
  }

  private double theta(double d, double d1, double d2, double
  d3)
  {
      double d4 = d2 - d;
      double d5 = d3 - d1;
      double d6 = Math.abs(d4);
      double d7 = Math.abs(d5);
      double d8;
      if((d6 == 0.0D) & (d7 == 0.0D))
          d8 = 0.0D;
      else
          d8 = d5 / (d6 + d7);
      if(d4 < 0.0D)
          d8 = 2D - d8;
      else
    if(d5 < 0.0D)
  d8 = 4D + d8;
```

```
        return d8 * 90D;
    }


    /**
     * a convex hull nearest insertion heuristic for TSP
     * @param ai the tour array
     * @return  # of nodes on the convex hull
     */
    public int convex_hull(int [] ai) {
int N = size();
int i = 0;
        boolean aflag[] = new boolean[N];
        //int ai[] = new int[N];
        for(int i1 = 0; i1 < N; i1++)
      aflag[i1] = true;
        int l = 0;
        for(int j1 = 1; j1 < N; j1++)
            if(y[j1] < y[l])
                l = j1;

        ai[0] = l;
        i = 1;
        double d = 360D;
        int j = l;
        for(int k1 = 0; k1 < N; k1++)
            if(k1 != l)
            {
                double d2 = theta(x[l], y[l], x[k1], y[k1]);
                if(d2 < d)
                {
                    d = d2;
                    j = k1;
                }
            }

        i = 2;
        ai[1] = j;
        aflag[j] = false;
        double d4 = d;
        do
        {
            double d1 = 360D;
            int k = l;
            for(int l1 = 0; l1 < N; l1++)
```

```
                if(aflag[l1])
                {
                    double d3 = theta(x[ai[i - 1]], y[ai[i -
 1]], x[l1], y[l1]);
                    if((d3 > d4) & (d3 < d1))
                    {
                        k = l1;
                        d1 = d3;
                    }
                }

            d4 = d1;
            if(k == l)
                break;
            ai[i] = k;
            i++;
            aflag[k] = false;
        } while(true);
        aflag[l] = false;
        return i;
    }

    /*
     * nearest insertion with convex hull heuristic
     */
    public void niInsertion(int aedge[])
    {
int n = size();
        boolean aflag[] = new boolean[n];
        int i = convex_hull(aedge);
for(int j = 0; j < i; j++)
        {
            aflag[aedge[j]] = true;
        }

        int k3 = -1;
        int l3 = -1;
        for(; i < n; i++)
        {
            if(true)
            {
                double d = 1e10;
                for(int k = 0; k < n; k++)
                    if(!aflag[k])
                    {
```

```
                        for(int i1 = 0; i1 < i; i1++)
                        {
                            int i2 = aedge[i1];
                            int l2 = aedge[(i1+1)%i];
                            double d4 = (dist[i2][k] + dist[k][
 l2]) - dist[i2][l2];
                            if(d4 < d)
                            {
                                d = d4;
                                k3 = k;
                                l3 = i1;
                            }
                        }

                    }

            }
            aflag[k3] = true;
      for(int p=i;p>l3+1;p--)
    aedge[p] = aedge[p-1];
      aedge[l3+1] = k3;
        }
    }
    /*
     * nearest insertion with some points fixed
     */
    public void fixed_niInsertion(int aedge[])
    {
int n = size();
        boolean aflag[] = new boolean[n];
        int i = fixCount;
for(int j = 0; j < i; j++)
        {
            aflag[aedge[j]] = true;
        }

        int k3 = -1;
        int l3 = -1;
        for(; i < n; i++)
        {
            if(true)
            {
                double d = 1e10;
                for(int k = 0; k < n; k++)
                    if(!aflag[k])
```

```
                    {
                        for(int i1 = 0; i1 < i; i1++)
                        {
                            int i2 = aedge[i1];
                            int l2 = aedge[(i1+1)%i];
                            double d4 = (dist[i2][k] + dist[k][
    l2]) - dist[i2][l2];
                            if(d4 < d)
                            {
                                d = d4;
                                k3 = k;
                                l3 = i1;
                            }
                        }

                    }

                }
                aflag[k3] = true;
      for(int p=i;p>l3+1;p--)
     aedge[p] = aedge[p-1];
       aedge[l3+1] = k3;
         }
     }

     /*
      * We use the mean points to generate the average
     trajectory
      * Added : 02/04/2005
      */
     public Trajectory getMeanTrajectory() {
   int n = size();
   double [] xcord = new double[n];
   double [] ycord = new double[n];
   for(int i=0; i<n ; i++) {
       xcord[i] = mean[i][0];
       ycord[i] = mean[i][1];
   }
   return new Trajectory(new NTSPInstance(xcord,ycord));
     }
}
```

B.2 QuadTree.java

```
//package Tree;
```

```
import java.awt.*;
import javax.swing.*;
import java.util.Vector;


/**
 * A <code>QTNode</code> in a QuadTree contains it's
   coordinates as well as it's dependents if any.
 *
 * @author <a href="mailto:lify@math.umd.edu">Feiyue Li</a>
 * @version 1.0
 */

class QTNode implements Cloneable
{
    static public boolean debug = false;

    final int point_radius = 2;
    static int id = 1;
    boolean single = false;
    /**
     * The dimension of the data. Usually it is 2.
     *
     */
    int Dimension;

    static int maxSize = 1;
    /**
     * <code>center</code> is the center of the quadrant that
    this node resides
     * Each quadrant is a rectangle in the hyper space with
    geometry 2*<code>size</code>
     *
     */
    double[] center;
    double[] size;

    /**
     *  <code>data</code> is the coordiates of the point in the
    quadrant if any.
     *
     */
    //double[]  data;
    Vector data;
```

```
/**
 * <code>sons[]</code> are all the dependent nodes of
 * this node.
 *
 */
QTNode sons[];


/**
 * A Quadtree node is <code>BLACK</code> if it is a
terminal node
 * If you insert some data into a black node, that node
becomes grey
 * and you need to create all the dependent nodes of it and
 insert
 * the data into the appropriate node.
 */
static final int BLACK = 1;

/**
 * A TreeNode is <code>WHITE</code> if it has no data.
 * If you insert some data into a white node, that node
becomes
 * black.
 */
static final int WHITE = 0;

/**
 * A TreeNode is <code>GREY</code> if it is not a terminal
node
 * When you insert some data into a grey node, you need to
insert it
 * into some appropriate depedent nodes of it.
 */
static final int GREY = 2;


/**
 * <code>state</code> is either black, white or grey.
 *
 */
int state;

int count = 0;
```

```
  static final int NE = 0;
  static final int NW = 1;
  static final int SW = 3;
  static final int SE = 2;

  public QTNode(int dim,double[] center_coord,double[]
 size_coord)
  {
// should check the consistency of the two data late
Dimension = dim;
center = new double[dim];
size = new double[dim];
System.arraycopy(center_coord,0,center,0,dim);
System.arraycopy(size_coord,0,size,0,dim);
data = new Vector(4);
if(data.size()>0) {
    System.out.println("weird!");
    System.exit(1);
}
state = WHITE;
single = false;
  }

  public QTNode(int dim,double [] center_coord,double[]
 size_coord, HPoint p)
  {
this(dim,center_coord,size_coord);
//data = new double[dim];
//System.arraycopy(data_point,0,data,0,dim);
data.add(p);
count = 1;
state = BLACK;
if(p.isFixed()) single=true;
if(data.size()>1) {
    System.out.println("bad!");
    System.exit(1);
}
  }

  public static synchronized  void setMaxSize(int n)
  {
maxSize = n;
System.out.println("#maxSize = "+maxSize);
  }
```

```
  public Object clone()
  {
QTNode newNode = null;
try {
    newNode = (QTNode)super.clone();
} catch(CloneNotSupportedException e) {}
center = new double[Dimension];
size = new double[Dimension];
System.arraycopy(newNode.center,0,center,0,Dimension);
System.arraycopy(newNode.size,0,size,0,Dimension);
if(data!=null)
    data = (Vector)newNode.data.clone();
switch(state)
    {
    case WHITE:
  newNode.sons = null;
  if(sons!=null) System.out.println("sons not null in white
 state");
  break;
    case BLACK:
  newNode.sons = null;
  if(sons!=null) System.out.println("sons not null in black
 state");
  break;
    case GREY:
  int n = 1<<Dimension;
  newNode.sons = new QTNode[n];
  for(int i=0;i<n;i++)
      {
    if(sons[i]!=null)
        newNode.sons[i] = (QTNode)sons[i].clone();
      }
  break;
    }
return newNode;
  }

  public boolean isEmpty()
  {
return state==WHITE;
  }


  public void insert_single(HPoint p) throws
 KeyDuplicateException
```

```
  {
switch(state) {
case WHITE:
    data.add(p);
    state = BLACK;
    single = true;
    count = 1;
    if(data.size()!=count) {
  System.out.println("wrong in white insert");
  System.exit(1);
    }
    break;
case BLACK:
    if(data.contains(p))
  throw new KeyDuplicateException();
    else {
  HPoint [] col = new HPoint[data.size()];
  data.toArray(col);
  int n = 1<<Dimension;
  sons = new QTNode[n];
  int pos;
  for(int i=0; i<col.length;i++) {
      pos = compare(center,col[i].coord);
      if(sons[pos]==null)
    sons[pos] = new QTNode(Dimension,getCenter(pos),scale(
 size,2));
      if(single)
    sons[pos].insert_single(col[i]);
      else sons[pos].insert(col[i]);
  }
  pos = compare(center,p.coord);
  if(sons[pos]==null)
      sons[pos] = new QTNode(Dimension,getCenter(pos),scale(
 size,2));
  sons[pos].insert_single(p);
  data = null;
  state = GREY;
  count++;
    }
    break;
case GREY:

    int pos = compare(center,p.coord);
    if(sons[pos]==null)
  sons[pos] = new QTNode(Dimension,getCenter(pos),scale(size
```

```
 ,2));
    sons[pos].insert_single(p);
    count++;
    break;
}

  }

  // this function and above are two recursive functions
 where we only
  // allow non-changing nodes to be in one quadrant.
  public void insert(HPoint p) throws KeyDuplicateException
  {
//  System.out.println("inserting\t"+p);
switch(state)
    {
    case WHITE:
  //System.out.println("inserting white");
  data.add(p);
  state = BLACK;
  count = 1;
  if(data.size()!=count) {
      System.out.println("wrong in white insert");
      System.exit(1);
  }
  break;

    case BLACK:
  if(data.contains(p))
      throw new KeyDuplicateException();
  else
      {
    if(single) {
        HPoint [] col = new HPoint[data.size()];
        data.toArray(col);
        int n = 1<<Dimension;
        sons = new QTNode[n];
        int pos;
        for(int i=0; i<col.length;i++) {
      pos = compare(center,col[i].coord);
      if(sons[pos]==null)
          sons[pos] = new QTNode(Dimension,getCenter(pos),
 scale(size,2));
      sons[pos].insert_single(col[i]);
        }
```

```
        pos = compare(center,p.coord);
        if(sons[pos]==null)  sons[pos] = new QTNode(Dimension
,getCenter(pos),scale(size,2));
        sons[pos].insert(p);
        state = GREY;
        data = null;
        count++;
    }
    else {
        data.add(p);
        count++;
        if(count!=data.size()){
      System.out.println("data inconsistent!");
      System.out.println("count="+count+",vector size="+data.
size());
      System.out.println(data);
      System.exit(1);
        }
        if(data.size()>maxSize)
      {
          HPoint [] col = new HPoint[data.size()];
          data.toArray(col);
          int n = 1<<Dimension;
          sons = new QTNode[n];
          for(int i=0; i<col.length;i++) {
        int pos = compare(center,col[i].coord);
        if(sons[pos]==null)
            sons[pos] = new QTNode(Dimension,getCenter(pos),
scale(size,2));
        sons[pos].insert(col[i]);
          }
          data = null;
          state = GREY;
      }
    }
      }
 break;

    case GREY:
 //System.out.println("insert grey");
 int where = compare(center,p.coord);
 if(sons[where]==null)
      {
    sons[where] = new QTNode(Dimension,getCenter(where),scale
(size,2),p);
```

```
        }
    else
        {
      sons[where].insert(p);
        }
    count++;
    break;
      default:
    System.err.println("error in insert");
      }
    }

    // this function is called when we allow some changing
   nodes to be in the same
    // quadrant as the non-changing nodes as long as the #
   doesn't exceed max.
    public void insert_duplicate(HPoint p) throws
   KeyDuplicateException
    {
//   System.out.println("inserting\t"+p);
switch(state)
    {
      case WHITE:
    //System.out.println("inserting white");
    data.add(p);
    state = BLACK;
    if(p.isFixed()) single = true;
    count = 1;
    if(data.size()!=count) {
        System.out.println("wrong in white insert");
        System.exit(1);
    }
    break;

      case BLACK:
    if(data.contains(p))
        throw new  KeyDuplicateException();
    else
        {
      data.add(p);
      count++;
      if(count!=data.size()){
          System.out.println("data inconsistent!");
          System.out.println("count="+count+",vector size="+
   data.size());
```

```
        System.out.println(data);
        System.exit(1);
    }
    if( data.size()>maxSize || (single && p.isFixed()) )
        {
      HPoint [] col = new HPoint[data.size()];
      data.toArray(col);
      int n = 1<<Dimension;
      sons = new QTNode[n];
      for(int i=0; i<col.length;i++) {
          int pos = compare(center,col[i].coord);
          if(sons[pos]==null)
        sons[pos] = new QTNode(Dimension,getCenter(pos),scale
(size,2));
          sons[pos].insert_duplicate(col[i]);
      }
      data = null;
      state = GREY;
        }
    else if(p.isFixed()) single=true;
      }
 break;

   case GREY:
 //System.out.println("insert grey");
 int where = compare(center,p.coord);
 if(sons[where]==null)
     {
   sons[where] = new QTNode(Dimension,getCenter(where),scale
(size,2),p);
     }
 else
     sons[where].insert_duplicate(p);
 count++;
 break;
   default:
 System.err.println("error in insert");
   }
 }


 // this function is called when all the points are changing
 from
 // one instance to another.
 public void insert_allchange(HPoint p) throws
```

```
 KeyDuplicateException
  {
//   System.out.println("inserting\t"+p);
switch(state)
    {
    case WHITE:
  data.add(p);
  state = BLACK;
  count = 1;
  if(data.size()!=count) {
      System.out.println("wrong in white insert");
      System.exit(1);
  }
  break;

    case BLACK:
  if(data.contains(p))
      throw new KeyDuplicateException();
  else
      {
    data.add(p);
    count++;
    if(count!=data.size()){
        System.out.println("data inconsistent!");
        System.out.println("count="+count+",vector size="+
 data.size());
        System.out.println(data);
        System.exit(1);
    }
    if(data.size()>maxSize)
        {
      HPoint [] col = new HPoint[data.size()];
      data.toArray(col);
      int n = 1<<Dimension;
      sons = new QTNode[n];
      for(int i=0; i<col.length;i++) {
        int pos = compare(center,col[i].coord);
        if(sons[pos]==null)
            sons[pos] = new QTNode(Dimension,getCenter(pos),
 scale(size,2));
        sons[pos].insert_allchange(col[i]);
      }
      data = null;
      state = GREY;
        }
```

```
        }
    break;

      case GREY:
    int where = compare(center,p.coord);
    if(sons[where]==null)
        {
      sons[where] = new QTNode(Dimension,getCenter(where),scale
   (size,2),p);
        }
    else
        {
      sons[where].insert_allchange(p);
        }
    count++;
    break;
      default:
    System.err.println("error in insert");
      }
    }

    public void remove(HPoint p)
    {
    }

    private boolean check_consistency()
    {
  switch(state) {
  case WHITE:
      return count==data.size();
  case BLACK:
      return count==data.size();
  case GREY:
      int n = 1<<Dimension;
      boolean ret = true ;
      for(int i=0;i<n;i++) {
    if(sons[i]!=null)
        ret = ret & sons[i].check_consistency();
      }
      return ret;
  default:return true;

  }
    }
```

```
  /**
   * <code>compare</code> compares two coordinates and return
  an int
   * to tell the order of the coordinates
   *
   * @param point1 the first coordinate
   * @param point2 the second coordinate
   * @return an <code>int</code> value to tell the location
   */
  private int compare(double [] point1, double [] point2)
  {
int i,d;
int bitresult;

d = 0;
for (i = 0; i < Dimension; i++)
    {
  bitresult = (point1[i] > point2[i]) ? 1 : 0;
  d = d | (bitresult << i);
    }
return d;
  }


  /**
   * Describe <code>getCenter</code> method here.
   *
   * @param c a <code>double[]</code> value
   * @param s a <code>double[]</code> value
   * @param pos an <code>int</code> value
   * @return a <code>double[]</code> value
   */
  public double[] getCenter(int pos)
  {
double ret [] = new double[Dimension];
for(int i=0; i<Dimension;i++)
    {
  int delta = (pos & (1<<i))>0?-1:1;
  ret[i] = center[i] + delta*size[i]/2;
    }
return ret;
  }

  private double[] scale(double[] p,double s)
  {
```

```
double [] ret = new double[Dimension];
for(int i=0;i<Dimension;i++)
    ret[i]=p[i]/s;
return ret;
  }

  public void traverse(Visitor v)
  {
switch(state)
    {
    case WHITE:
  break;
    case BLACK:
  v.visit(this);
  break;
    case GREY:
  for(int i=0;i<sons.length;i++){
      if(sons[i]!=null)
    sons[i].traverse(v);
  }

    }
  }

  public HPoint average()
  {
double avg[] = new double[Dimension];
for(int i=0;i<Dimension;i++) avg[i]=0.0;
int n = data.size();
int dup=0;
for(int i=0;i<n;i++) {
    HPoint p = (HPoint) data.get(i);
    //if(p.isFixed()) dup++;
    for(int j=0;j<Dimension;j++){
  if(p.isFixed()) {
      // avg[j]+=maxSize*p.coord[j];
      avg[j] += p.coord[j];
  }
  else
      avg[j]+=p.coord[j];
    }
}
//n+=dup*(maxSize-1);
for(int i=0;i<Dimension;i++)
    avg[i]/=n;
```

```
return new HPoint(avg);
  }



  public String toString()
  {
StringBuffer sb = new StringBuffer();
switch(state)
    {
    case WHITE:
  sb.append("white");
  if(sons!=null) System.out.println("@@@@weird@@@@");
  sb.append(",sons="+sons);
  sb.append(",vector="+data);
  break;
    case BLACK:
  sb.append("black");
  sb.append("center(");
  for(int i=0; i<Dimension; i++) sb.append(center[i]+",");
  sb.append(")");
  sb.append("data[");
  for(int i=0;i<data.size();i++) sb.append(data.get(i)+",");
  sb.append("]\n");
  break;
    case GREY:
  sb.append("grey("+count+")");
  for(int i=0;i<sons.length;i++) {
      if(sons[i]!=null)
    sb.append(sons[i].toString());
  }
  break;
    }
return sb.toString();
  }

  public void dump()
  {
System.out.println(this);
  }



  public void drawNode(Graphics g)
  {
//System.out.println("drawing..");
```

```
Graphics2D g2 = (Graphics2D) g;
switch(state)
    {
    case WHITE:
  return;
    case BLACK:
  if(data.size()!=count) {
      System.out.println("error!");
      System.exit(1);
  }

  int r = point_radius;
  g2.setPaint(Color.black);
  for(int i=0;i<data.size();i++) {
      HPoint p =(HPoint) data.get(i);
      //System.out.println("drawing "+p);
      if(p.isFixed()) {
    //System.out.println("fixed");
    g2.setPaint(Color.blue);
    g2.fillOval((int)p.coord[0]-r,(int)p.coord[1]-r,2*r,2*r);
    g2.setPaint(Color.black);
      }
      else {
    g2.setPaint(Color.black);
    g2.drawOval((int)p.coord[0]- r,(int)p.coord[1]-r,2*r,2*r)
;
      }

  }
  g2.setPaint(Color.red);
  HPoint avg = average();
  g2.fillOval((int)avg.coord[0]-r,(int)avg.coord[1]-r,2*r,2*r
);
  g2.setPaint(Color.black);

  //System.out.println("draw("+data[0]+","+data[1]+")");
  break;
    case GREY:
  int cx = (int)center[0];
  int cy = (int)center[1];
  final float dash1[] = {15.0f,5.0f };
  g2.setStroke(new BasicStroke(1.0f,BasicStroke.CAP_BUTT,
 BasicStroke.JOIN_MITER,10.0f,dash1,0.0f));
  g2.drawLine(cx-(int)size[0],cy,cx+(int)size[0],cy);
  g2.drawLine(cx,cy-(int)size[1],cx,cy+(int)size[1]);
```

```
    //g.drawOval((int)data[0],(int)data[1],5,5);
    int n = 1<<Dimension;
    for(int i=0;i<n;i++)
        if(sons[i]!=null)
      sons[i].drawNode(g);

      }
    }

}

class MyQTVisitor implements Visitor
{
    Vector v ;

    public MyQTVisitor(int size)
    {
  v = new Vector(size);
    }

    public void visit(Object o)
    {
  v.add(((QTNode) o).average());
    }

    public HPoint [] getResult()
    {
  HPoint [] avg = new HPoint[v.size()];
  v.toArray(avg);
  return avg;
    }

}

public class QuadTree implements Cloneable
{

    /**
     * <code>root</code> is the root of the quadtree
     *
     */
    QTNode root;
    Vector elm;

    /**
```

```
     * <code>total</code> is the total number of data points in
    the quadtree.
     *
     */

    int maxPerQuad = 1;
    HPoint c; // center of the world
    HPoint s; // half size of the world
    boolean changed = false;

    /**added for probabilistic quadtree **/
    boolean [] change;
    int fixCount;

    int Dimension;

    public QuadTree(int dim,HPoint center,HPoint size)
    {
Dimension = dim;
c = new HPoint(center);
s = new HPoint(size);
root = new QTNode(dim,center.coord,size.coord);
elm = new Vector(20);
//System.out.println("damned it is called,exiting...");
//System.exit(1);
    }

    public QuadTree(int dim,NTSPInstance t)
    {
Dimension = dim;
HPoint [] geom = t.getGeometry();
HPoint orig = geom[0];
HPoint size = geom[1];
double [] center = { orig.coord[0]+size.coord[0]/2, orig.
 coord[1]+size.coord[1]/2 };
double [] half_size = { size.coord[0]/2, size.coord[1]/2 };
c = new HPoint(center);
s = new HPoint(half_size);
change = new boolean[t.size()];
root = new QTNode(dim,center,half_size);
elm = new Vector(t.size());
change = new boolean[t.size()];
System.arraycopy(t.change,0,change,0,t.size());
for(int i=0;i<t.size();i++) {
    double [] coord = { t.x[i],t.y[i] };
```

```
      try {
    if(NTSParam.isAllChange())
        insert_allchange(new HPoint(coord,i));
    else insert(new HPoint(coord,i));
      }
      catch(KeyDuplicateException e) {
    System.out.println("oops!duplicated_node_found!");
      }
}
/** added for probabilistic quadtree **/
fixCount = t.fixCount;
  }

  /**
   * public interface to create a quadtree
   */
  public QuadTree(int dim,NTSPGraph g)
  {
this(dim,g.own);
  }

  public void hookup(NTSPGraph g)
  {
change = new boolean[g.size()];
System.arraycopy(g.change,0,change,0,g.size());
fixCount = g.fixCount;
  }

  public int size()
  {
if(elm.size()!=root.count){
    System.out.println("wrong!!!!");
    System.exit(1);
}
return elm.size();
  }

  public void setMaxSize(int max)
  {
QTNode.setMaxSize(max);
changed = true;
  }

  public Object clone()
  {
```

```
QuadTree newTree = null;
try {
    newTree = (QuadTree)super.clone();
}
catch(CloneNotSupportedException e) {
}
newTree.s = (HPoint)s.clone();
newTree.c = (HPoint)c.clone();
newTree.root = (QTNode)root.clone();
newTree.elm = (Vector)elm.clone();
newTree.change = (boolean [])change.clone();
return newTree;
  }

  public String toString()
  {
StringBuffer sb = new StringBuffer();
sb.append("size("+s+")");
sb.append("center("+c+")");
sb.append("data["+elm+"]");
sb.append("\ndetails:");
sb.append(root.toString());
return sb.toString();
  }


  public void insert(HPoint p) throws KeyDuplicateException
  {
if(!elm.contains(p)) {
    if(!change[p.getID()])
  root.insert_single(p);
    else
  root.insert(p);
    elm.add(p);
}
  }

  public void insert_duplicate(HPoint p) throws
 KeyDuplicateException
  {
root.insert_duplicate(p);
elm.add(p);
  }

  public void insert_allchange(HPoint p)  throws
```

```
 KeyDuplicateException
  {
root.insert_allchange(p);
elm.add(p);
  }

  public void insert(NTSPInstance t) throws
 KeyDuplicateException
  {
for(int i=0;i<t.size();i++) {
    double [] p = {t.x[i],t.y[i] };
    insert(new HPoint(p));
}
System.out.println("should not be called .....");
System.exit(1);
  }



  public void remove(HPoint p)
  {
root.remove(p);
  }

  public Trajectory agregate(int max)
  {
if(root.isEmpty()) return null;
QuadTree copy = (QuadTree)this.clone();
copy.setMaxSize(max);
if(NTSParam.isAllChange())
    copy.reinsert_allchange();
else copy.reinsert();
MyQTVisitor visitor = new MyQTVisitor(copy.size());
copy.traverse(visitor);
HPoint [] avg = visitor.getResult();
NTSPInstance ins = new NTSPInstance(avg);
Trajectory avg_trajectory = new Trajectory(ins);
if(avg_trajectory.size()>1) {
    avg_trajectory.niInsertion();
    avg_trajectory.twOpt();
    //avg_trajectory.setSeqence(MyORUtil.getOPT(
 avg_trajectory.getX(),avg_trajectory.getY()));
    return avg_trajectory;
}
else return null;
```

```
  }

  public void traverse(Visitor v)
  {
root.traverse(v);
  }

  private void reinsert()
  {
root = null;
root = new QTNode(Dimension,c.coord,s.coord);
for(int i=0;i<elm.size();i++) {
    try {
  HPoint p = (HPoint)elm.get(i);
  if(!change[p.getID()])
      root.insert_single(p);
  else root.insert(p);
    }
    catch(KeyDuplicateException e) {}
}
  }

  private void reinsert_duplicate()
  {
root = null;
root = new QTNode(Dimension,c.coord,s.coord);
for(int i=0;i<elm.size();i++) {
    try {
  HPoint p = (HPoint)elm.get(i);
  root.insert_duplicate(p);
    }
    catch(KeyDuplicateException e) {}
}
  }

  private void reinsert_allchange()
  {
root = null;
root = new QTNode(Dimension,c.coord,s.coord);
for(int i=0;i<elm.size();i++) {
    try {
  HPoint p = (HPoint)elm.get(i);
  root.insert_allchange(p);
    }
    catch(KeyDuplicateException e) {}
```

```
  }
    }

    public void draw(Graphics g)
    {
  if(changed)
      reinsert_duplicate();
  root.drawNode(g);
    }

    public void dump()
    {
  System.out.println(this);
    }

    public static void main(String args[]) throws Exception
    {
  double[] c = { 4,4 };
  double[] s = { 8,8 };
  HPoint center = new HPoint(c);
  HPoint size = new HPoint(s);
  QuadTree one = new QuadTree(2,center,size);
  //QuadTree two = new QuadTree(2,center,size);
  QuadTree two = (QuadTree)one.clone();
  System.out.println("tree 2"+ two);
  one.insert(center);
  one.insert(size);
  System.out.println("tree 1"+one);
  System.out.println("tree 2"+two);
    }

}
```

B.3 HPoint.java

```
// Hyper-Point class supporting KDTree class

//package edu.brandeis.cs.levy.CG;

class HPoint implements Cloneable{

    public double [] coord;
    int id=-1;
    boolean fixed ;
```

```
  public HPoint(int n) {
coord = new double [n];
  }

  public HPoint(double [] x) {

coord = new double[x.length];
for (int i=0; i<x.length; ++i) coord[i] = x[i];
  }

  public HPoint(double []x, int id)
  {
this(x);
this.id=id;
  }

  public HPoint(double []x,int id,boolean change)
  {
this(x,id);
this.fixed = !change;
  }

  public boolean isFixed()
  {
return fixed;
  }

  public HPoint(HPoint that)
  {
coord = new double[that.coord.length];
coord = (double []) that.coord.clone();
id = that.id;
  }

  public Object clone() {
HPoint ret = null;
try {
    ret = (HPoint)super.clone();
}catch(CloneNotSupportedException e){
}
ret.coord = (double []) coord.clone();
return ret;
  }

  public boolean equals(Object p) {
```

```
if(p instanceof HPoint) {

// seems faster than java.util.Arrays.equals(), which is not
// currently supported by Matlab anyway

    for (int i=0; i<coord.length; ++i)
  if (coord[i] != ((HPoint) p).coord[i])
      return false;
    return true;
}
else return false;
  }

  public boolean equals(HPoint p)
  {
for (int i=0; i<coord.length; ++i)
    if (coord[i] != ((HPoint) p).coord[i])
  return false;
return true;
  }

  public int getID()
  {
return id;
  }

  public static double sqrdist(HPoint x, HPoint y) {

double dist = 0;

for (int i=0; i<x.coord.length; ++i) {
    double diff = (x.coord[i] - y.coord[i]);
    dist += (diff*diff);
}

return dist;

  }

  public static double eucdist(HPoint x, HPoint y) {

return Math.sqrt(sqrdist(x, y));
  }

  public String toString() {
```

```
  String s = "(";

  for (int i=0; i<coord.length; ++i) {
      s += coord[i];
      if (i < coord.length-1) s += " ";
  }
  s += ")";
  return s;

    }


}
```

B.4 MarkoveChain.java

```
import java.util.Random;
public class MarkovChain {
    NTSPGraph g;
    public State current_state;
    static MyRandom rand = new MyRandom();
    public double T=0.1D;
    public State bestState;

    public MarkovChain(NTSPGraph g) {
  this.g = (NTSPGraph)g.clone();
    }

    public void init(State s) {
  current_state = s;
  bestState = s;
    }

    public void setTemperature(double t) {
  T = t;
    }

    public void transit() {
  State oldState = (State) ((Permutation)current_state).clone()
   ;
  double oldObj = oldState.getObjValue();
  current_state.transit();
  double newObj = current_state.getObjValue();
```

```
double threshold = Math.exp((oldObj-newObj)/T);
      if(rand.nextDouble()>threshold) {
    current_state = oldState;
}
  }

  public void transit(int n) {
for(int i=0;i<n;i++)
    transit();
  }

  public String toString() {
return bestState.toString();
  }

  public State getCurrentState() {
return current_state;
  }

  public void burnIn(int step) {
transit(step);
  }

  public void burnIn() {
transit(5000);
  }

  public void testBurnIn(int step) {
for(int i=0;i<step;i++) {
    System.out.print(current_state.getObjValue()+";");
    transit();
}
  }

  public Trajectory average(int size) {
burnIn();
State t[] = new State[size];
t[0] = (State) ((Permutation)current_state).clone();
for(int i=1;i<size;i++) {
    for(int j=0;j<100;j++)
  transit();
    t[i] = (State) ((Permutation) current_state).clone();
    ((Permutation) t[0]).normalize((Permutation)t[i]);
}
double [] avgx = new double[g.size()];
```

```
  double [] avgy = new double[g.size()];
  for(int i=0;i<g.size();i++) {
      avgx[i] = 0D; avgy[i] = 0D;
  }
  for(int i=0;i<g.size();i++) {
      for(int k=0;k<size;k++) {
    avgx[i] += g.x[((Permutation) t[k]).seq[i]];
    avgy[i] += g.y[((Permutation) t[k]).seq[i]];
      }
  }
  for(int i=0;i<g.size();i++) {
      avgx[i]/=size;
      avgy[i]/=size;
  }
  NTSPInstance instance = new NTSPInstance(avgx,avgy);
  Trajectory tjr = new Trajectory(instance);
  for(int i=0;i<g.size();i++) tjr.seq[i] = i;
  tjr.setInstance(instance);
  return tjr;
    }

}
```

B.5 Trajectory.java

```
/*
 * written on Mar.8,2003
 * a trajectory is a permutation which maps the set {1,2,..,n}
 * to points on R^2
 * May 10, add another heuristic
*/
import java.util.Random;
import java.util.Arrays;
import java.awt.*;

public class Trajectory implements Cloneable{
    NTSPInstance instance;
    int [] seq;
    double obj;

    public Trajectory(NTSPInstance i) {
  instance = (NTSPInstance)i.clone();
  seq = new int[i.size()];
  for(int j=0;j<i.size();j++)
      seq[j] = j;
```

```
obj = getObjValue();
  }

  public Trajectory(Trajectory t) {
instance = t.instance;
obj = t.obj;
seq = new int[t.seq.length];
System.arraycopy(t.seq,0,seq,0,t.seq.length);
  }

  public void setInstance(NTSPInstance i) {
instance = i;
  }

  public void init(int [] seed) {
if(seq==null ||seq.length!=seed.length) seq = new int[seed.
 length];
System.arraycopy(seed,0,seq,0,seed.length);
obj = getObjValue();
  }

  public double [] getX()
  {
return instance.x;
  }

  public double[] getY()
  {
return instance.y;
  }

  public void setSeqence(int [] s) {
seq = new int[s.length];
System.arraycopy(s,0,seq,0,s.length);
obj = getObjValue();
  }

  public int size() { return instance.size(); }

  public Object clone() {
Trajectory t =null;
try {
    t = (Trajectory)super.clone();

} catch(CloneNotSupportedException e) {}
```

```
t.instance = (NTSPInstance)instance.clone();
System.arraycopy(seq,0,t.seq,0,seq.length);
return t;
  }

  public String toString() {
StringBuffer sb = new StringBuffer();
sb.append(instance.toString());
sb.append("tour=[");
for(int i=0;i<seq.length-1;i++)
    sb.append(seq[i]+";");
sb.append(seq[seq.length-1]+"];");
return sb.toString();
  }

  public Trajectory interplate(NTSPInstance t) {
Trajectory newTraj = new Trajectory(t);
class NTSPInt implements Comparable {
    int ind;
    double intp;
    NTSPInt(int i, double t) {
  ind =i;
  intp = t;
    }
    public int compareTo(Object o) {
  if(intp<((NTSPInt)o).intp) return -1;
  else if(intp==((NTSPInt)o).intp) return 0;
  else return 1;
    }

}
NTSPInt [] interp = new NTSPInt[t.size()];
for(int i=0;i<t.size();i++) {
    interp[i] = new NTSPInt(i,getIntVal(t.x[i],t.y[i]));
}
Arrays.sort(interp);
for(int i=0;i<t.size();i++) {
    newTraj.seq[i] = interp[i].ind;
}
newTraj.obj = newTraj.getObjValue();
return newTraj;
  }

  public Trajectory MyInterplate(NTSPInstance t)
  {
```

```
Trajectory newTraj = new Trajectory(t);
int n = size();
int [] bigTour = new int[2*n];
System.arraycopy(seq,0,bigTour,0,n);
int doneSoFar = n;
for(int k=0;k<n;k++) {
    double cheapCost=1e10;
    int insInd = -1;
    for(int i=0;i<doneSoFar;i++) {
  int i1 = bigTour[i];
  int i2 = bigTour[(i+1)%doneSoFar];
  int i3 = k+n;
  double x1,y1,x2,y2,x3,y3;
  if(i1<n) {
      x1 = this.instance.x[i1];
      y1 = this.instance.y[i1];
  }
  else {
      x1 = t.x[i1-n];
      y1 = t.y[i1-n];
  }
  if(i2<n) {
      x2 = this.instance.x[i2];
      y2 = this.instance.y[i2];
  }
  else {
      x2 = t.x[i2-n];
      y2 = t.y[i2-n];
  }
  x3 = t.x[k];
  y3 = t.y[k];
  double insCost = Math.sqrt((x1-x3)*(x1-x3)+(y1-y3)*(y1-y3))
      + Math.sqrt((x2-x3)*(x2-x3)+(y2-y3)*(y2-y3))
      - Math.sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
  if(insCost<cheapCost) {
      cheapCost = insCost;
      insInd = i+1;
  }
    }
    for(int j=doneSoFar;j>insInd;j--)
  bigTour[j] = bigTour[j-1];
    bigTour[insInd] = k+n;
    doneSoFar++;
}
int k=0;
```

```
for(int i=0;i<n;i++) {
    while(bigTour[k]<n) k++;
    newTraj.seq[i] = bigTour[k]-n;
    k++;
}
newTraj.obj = newTraj.getObjValue();
return newTraj;
  }



  private double getIntVal(double x,double y) {
int n = instance.size();
double minDist=1e10;
double interpVal = 0.0;
double x1,x2,y1,y2,len1,len2,len;
for(int i=0;i<n;i++) {
    x1 = instance.x[seq[i]];
    y1 = instance.y[seq[i]];
    x2 = instance.x[seq[(i+1)%n]];
    y2 = instance.y[seq[(i+1)%n]];
    len1 = (x-x1)*(x-x1) + (y-y1)*(y-y1);
    len2 = (x-x2)*(x-x2) + (y-y2)*(y-y2);
    len  = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2);
    if(len1>minDist && len2>minDist) continue;
    if(len1<=len2+len && len2<=len1+len) {
  double lamda = (len2-len1)/(2*len) + 0.5;
  double min = lamda*len1+(1-lamda)*len2-lamda*(1-lamda)*len;
  if(minDist>min) {
      interpVal = 1-lamda + i;
      minDist = min;
  }
    }
    else {
  if(len1<len2) {
      if(minDist > len1) {
    minDist = len1;
    interpVal = i;
      }
  }
  else {
      if(minDist > len2) {
    minDist = len2;
    interpVal = i+1;
      }
```

```
  }
    }
}
return interpVal;
  }

  private double theta(double d, double d1, double d2, double
  d3)
  {
      double d4 = d2 - d;
      double d5 = d3 - d1;
      double d6 = Math.abs(d4);
      double d7 = Math.abs(d5);
      double d8;
      if((d6 == 0.0D) & (d7 == 0.0D))
          d8 = 0.0D;
      else
          d8 = d5 / (d6 + d7);
      if(d4 < 0.0D)
          d8 = 2D - d8;
      else
      if(d5 < 0.0D)
          d8 = 4D + d8;
      return d8 * 90D;
  }

  public int convex_hull(int [] ai) {
int N = size();
int i = 0;
      boolean aflag[] = new boolean[N];
      for(int i1 = 0; i1 < N; i1++)
    aflag[i1] = true;
      int l = 0;
      for(int j1 = 1; j1 < N; j1++)
          if(instance.y[j1] < instance.y[l])
              l = j1;

      ai[0] = l;
      i = 1;
      double d = 360D;
      int j = l;
      for(int k1 = 0; k1 < N; k1++)
          if(k1 != l)
          {
              double d2 = theta(instance.x[l], instance.y[l],
```

```
  instance.x[k1], instance.y[k1]);
                if(d2 < d)
                {
                    d = d2;
                    j = k1;
                }
            }

        i = 2;
        ai[1] = j;
        aflag[j] = false;
        double d4 = d;
        do
        {
            double d1 = 360D;
            int k = l;
            for(int l1 = 0; l1 < N; l1++)
                if(aflag[l1])
                {
                    double d3 = theta(instance.x[ai[i - 1]],
 instance.y[ai[i - 1]], instance.x[l1], instance.y[l1]);
                    if((d3 > d4) & (d3 < d1))
                    {
                        k = l1;
                        d1 = d3;
                    }
                }

            d4 = d1;
            if(k == l)
                break;
            ai[i] = k;
            i++;
            aflag[k] = false;
        } while(true);
        aflag[l] = false;
        return i;
    }

    /*
     * nearest insertion with convex hull heuristic
     */
    public void niInsertion()
    {
int n = size();
```

```
      boolean aflag[] = new boolean[n];
      int i = convex_hull(seq);
for(int j = 0; j < i; j++)
      {
          aflag[seq[j]] = true;
      }

      int k3 = -1;
      int l3 = -1;
      for(; i < n; i++)
      {
          if(true)
          {
              double d = 1e10;
              for(int k = 0; k < n; k++)
                  if(!aflag[k])
                  {
                      for(int i1 = 0; i1 < i; i1++)
                      {
                          int i2 = seq[i1];
                          int l2 = seq[(i1+1)%i];
        double d4 = instance.dist(i2,k) + instance.dist(k,l2)
  - instance.dist(i2,l2);
                          if(d4 < d)
                          {
                              d = d4;
                              k3 = k;
                              l3 = i1;
                          }
                      }

                  }

          }
          aflag[k3] = true;
    for(int p=i;p>l3+1;p--)
  seq[p] = seq[p-1];
    seq[l3+1] = k3;
      }
obj = getObjValue();
  }

  /*
   * fixed nearest insertion with convex hull heuristic
   */
```

```
  public void fixed_niInsertion()
  {
int n = size();
      boolean aflag[] = new boolean[n];
int [] newseq = new int[n];
for(int j = 0; j < n; j++)
      {
          aflag[seq[j]] = instance.change[seq[j]];
      }
System.arraycopy(instance.partialTour,0,seq,0,instance.
 fixCount);
      int k3 = -1;
      int l3 = -1;
int i = instance.fixCount;
for(; i < n; i++)
      {
    if(true)
          {
              double d = 1e10;
              for(int k = 0; k <n ; k++)
                  if(aflag[k])
                  {
                      for(int i1 = 0; i1 < i; i1++)
                      {
                          int i2 = seq[i1];
                          int l2 = seq[(i1+1)%i];
        double d4 = instance.dist(i2,k) + instance.dist(k,l2)
  - instance.dist(i2,l2);
                          if(d4 < d)
                          {
                              d = d4;
                              k3 = k;
                              l3 = i1;
                          }
                      }

                  }

          }
          aflag[k3] = false;
    for(int p=i;p>l3+1;p--)
  seq[p] = seq[p-1];
    seq[l3+1] = k3;
      }
obj = getObjValue();
```

```
  }

  public double getObjValue() {
double val = 0.0;
int n = size();
for(int i=0;i<n;i++)
    val += instance.dist(seq[i],seq[(i+1)%n]);
return val;
  }

  public void twOpt() {
while(twOptStep()) ;
check_obj();
  }

  private void check_obj()
  {
if(Math.abs(getObjValue()-obj)>1e-6) {
    System.out.println("obj value not consistent!");
    System.exit(1);
}
  }

  private boolean  twOptStep() {
int n =size();
boolean ret = false;
for(int i=0;i<n;i++) {
    for(int j=i+1;j<n;j++) {
  int previ = i>0?i-1:n-1;
  int nextj = j==n-1?0:j+1;
  if(previ!=j && nextj!=i) {
      double saving = instance.dist(seq[previ],seq[i])+
 instance.dist(seq[j],seq[nextj])
    - instance.dist(seq[previ],seq[j]) - instance.dist(seq[i
 ],seq[nextj]);
      if(saving>1e-8) {
    flip(i,j);
    obj -=saving;
    return true;
      }
  }
    }
}
return false;
  }
```

```
  private void flip(int i,int j) {
int k = j-i+1;
int tmp;
for(int t=0;t<k/2;t++) {
    tmp = seq[i+t];
    seq[i+t] = seq[j-t];
    seq[j-t] = tmp;
}
  }

  class exSearchObj {
int s;
int t;
int[] m;
double obj;
exSearchObj(int s1, int t1,int []m1,double obj1) {
    s = s1;t = t1; obj = obj1;
    m = new int[m1.length];
    System.arraycopy(m1,0,m,0,m1.length);
}

public String toString() {
    StringBuffer sb = new StringBuffer();
    sb.append("["+s+",");
    for(int i=0;i<m.length;i++) sb.append(m[i]+",");
    sb.append(t+"]");
    sb.append(obj);
    return sb.toString();
}
  }

  public void clean(int w) {
int n = size();
boolean improve = true;
while(improve) {
    improve = false;
    int [] v = new int[w+1];
    for(int i=0;i<n;i++) {
  for(int k=1;k<w+2;k++) v[k-1] = seq[(i+k)%n];
  double val=0.0;
  for(int k=0;k<w;k++) {
      val+=instance.dist[v[k]][v[k+1]];
  }
  val+=instance.dist[seq[i]][v[0]]+instance.dist[seq[(i+w+2)%
```

```
 n]][v[w]];
  double oldObj = obj;
  exSearchObj r = exsearch(seq[i],seq[(i+w+2)%n],v,w+1);
  for(int k=1;k<w+2;k++) seq[(i+k)%n] = r.m[k-1];
  obj = obj - val + r.obj;
  if(oldObj-obj>1e-8)
      improve = true;
  if(Math.abs(obj-getObjValue())>1e-8) {
      System.out.println("obj="+obj);
      System.out.println("getobj="+getObjValue());
      System.out.println("error in clean, i="+i);
      System.exit(1);
  }
    }
}
  }
  public void debug() {
int n = size();
int [] v = new int[6];
v[0] = 3; v[1] = 5 ; v[2] = 6; v[3]=2; v[4] = 1; v[5]=4;
exSearchObj r = exsearch(seq[0],seq[7],v,6);
System.out.println(r.obj);
for(int i=0;i<6;i++) System.out.print(r.m[i]);
  }

  public exSearchObj exsearch(int s,int t, int []v, int n) {
if(n==2) {
    double d1= instance.dist[s][v[1]]+ instance.dist[v[0]][t
 ]+instance.dist[v[0]][v[1]];
    double d2 = instance.dist[s][v[0]] + instance.dist[v[1]][
 t]+instance.dist[v[0]][v[1]];
    if(d1>d2) return new exSearchObj(s,t,v,d2);
    else {
  int w[] = new int[2];
  w[0] = v[1]; w[1] = v[0];
  return new exSearchObj(s,t,w,d1);
    }
}
else {
    if(n!=v.length) {
  System.out.println("error in length");
  System.exit(1);
    }
    double ret[] = new double[n];
    exSearchObj [] back = new exSearchObj[n];
```

```
    double min=1e10;
    int min_ind=-1;
    for(int i=0;i<n;i++) {
  int [] left = new int[n-1];
  for(int k=0,ind=0;k<n;k++) {
      if(k==i) continue;
      left[ind++] = v[k];
  }
  back[i] = exsearch(v[i],t,left,n-1);
  ret[i] = instance.dist[s][v[i]]+back[i].obj;
  if(ret[i]<min) {
      min=ret[i];
      min_ind = i;
  }
    }
    int []vec = new int[n];
    vec[0] = v[min_ind];
    for(int i=1;i<n;i++) vec[i] = back[min_ind].m[i-1];
    return new exSearchObj(s,t,vec,min);
}
  }


  /**
   * Describe <code>nnInsertion</code> method here.
   * Nearest Insertion heuristic for TSP
   *
   */
  public void nnInsertion() {
int n = size();
boolean flag[] = new boolean[n];
for(int i=0;i<n;i++) flag[i] = false;
int j=0;
flag[j] = true;
for(int i=0;i<n-1;i++) {
    double d = 1e10;
    int l=0;
    for(int k=0;k<n;k++) {
  if(!flag[k] && instance.dist(j,k)<d) {
      d = instance.dist(j,k);
      l = k;
  }
    }
    seq[i] = j;
    seq[i+1] = l;
```

```
      flag[l] = true;
      j = l;
  }
    }

    public void draw(Graphics g) {
  Graphics2D g2 = (Graphics2D) g;
  g2.setPaint(Color.blue);
  g2.setStroke(new BasicStroke());
  int n = size();
  for(int i=0;i<n;i++) {
      g2.drawLine((int)instance.x[seq[i%n]],(int)instance.y[seq
   [i%n]],
            (int)instance.x[seq[(i+1)%n]],(int)instance.y[seq[(i
   +1)%n]]);
  }
  g2.setPaint(Color.black);
    }

    public void dump()
    {
  for(int i=0;i<seq.length;i++)
      System.out.print(seq[i]+",");
  System.out.println();
    }



}
```

B.6 Permutation.java

```
/*
 * written on Mar.8,2003
 * a trajectory is a permutation which maps the set {1,2,..,n}
 * to points on R^2
 */
import java.util.Random;
public class Permutation implements Cloneable,State{
    int [] seq;
    NTSPGraph g;
    int n;
    double obj;
    static Random rand = new Random();
```

```
  public Permutation(NTSPGraph g) {
this.g = g;
seq = new int[g.size()];
n = g.size();
  }

  public Permutation(Permutation t) {
n = t.n;
seq = new int[n];
System.arraycopy(t.seq,0,seq,0,n);
g = t.g;
obj = t.obj;
  }

  public void init(int [] seed) {
if(n!=seed.length) {
    System.out.println("in Permutation::init(),different size
 ");
    System.exit(1);
}
System.arraycopy(seed,0,seq,0,n);
obj = evaluate();
  }

  /*
   * do a random 2opt move
   */
  public void transit() {
int i=-1,j=-1;
do {
    i = rand.nextInt(n);
    j = rand.nextInt(n);

    int tmp;
    if(i>j) {
  tmp = i;
  i = j;
  j = tmp;
    }
} while((j-i)%n==1 || (j-i)%n==n-1);
flip(i,j);
  }

  private void flip(int i,int j) {
```

```
int k = j-i+1;
int tmp;
int previ = i>0?i-1:n-1;
int nextj = j==n-1?0:j+1;
if(previ!=j && i!=nextj) {
    for(int t=0;t<k/2;t++) {
  tmp = seq[i+t];
  seq[i+t] = seq[j-t];
  seq[j-t] = tmp;
    }
    obj+=(g.distance(seq[previ],seq[i])+g.distance(seq[j],seq
 [nextj])
    -g.distance(seq[previ],seq[j])-g.distance(seq[i],seq[
 nextj]));
}
  }

  public Object clone() {
Permutation t =null;
try {
    t = (Permutation)super.clone();
} catch(CloneNotSupportedException e) {}
t.seq = new int[n];
System.arraycopy(seq,0,t.seq,0,n);
return t;
  }

  public double getObjValue() {
return obj;
  }

  public double evaluate() {
double OBJ = 0D;
for(int i=0;i<n-1;i++)
    OBJ += g.distance(seq[i],seq[i+1]);
OBJ += g.distance(seq[n-1],seq[0]);
return OBJ;
  }

  public String toString() {
StringBuffer sb = new StringBuffer();
sb.append("tour=[");
for(int i=0;i<n-1;i++) sb.append(seq[i]+",");
sb.append(seq[n-1]+","+seq[0]+"];");
sb.append("\n"+obj+"=="+evaluate());
```

```
return sb.toString();
  }

  /*
   * normalize the trajectory t with this trajectory
   */
  final public double normalize(Permutation t) {
if(n!=t.n) {
    System.out.println("can't normalize for different
 permutation!");
    System.exit(1);
}
double min_norm = 1e10;
int j=-1;
for(int i=0;i<n;i++) {
    double norm =0D;
    for(int k=0;k<n;k++) {
  /*
   * use L_1 norm here
   */
  int index = t.seq[(k+i)%n];
  norm += Math.max(Math.abs(g.x[seq[k]]-g.x[index]),Math.abs(
 g.y[seq[k]] - g.y[index]));
    }
    if(min_norm>norm) {
  min_norm = norm;
  j = i;
    }
}
boolean reverse = false;
for(int i=0;i<n;i++) {
    double norm=0D;
    for(int k=0;k<n;k++) {
  int index = t.seq[(2*n-k-i)%n];
  norm += Math.max(Math.abs(g.x[seq[k]]-g.x[index]), Math.abs
 (g.y[seq[k]] - g.y[index]));
    }
    if(min_norm > norm) {
  min_norm = norm;
  j = i;
  reverse = true;
    }
}
int [] copy = new int[n];
System.arraycopy(t.seq,0,copy,0,n);
```

```
  if(!reverse){
      if(j>0) {
    for(int k=0;k<n;k++) {
        t.seq[k] = copy[(k+j)%n];
    }
      }
  }
  else {
      for(int k=0;k<n;k++)
    t.seq[k] = copy[(-k-j+2*n)%n];
  }
  return min_norm;
    }

    public void testNormalize() {
  Permutation t = (Permutation) this.clone();
  normalize(t);
  System.out.println(t);
    }
}
```

B.7 MyRandom.java

```
import java.util.Random;
public class MyRandom {
    static Random rand = new Random();

    public MyRandom() {
  if(rand==null)
      rand = new Random();
    }

    public double nextGaussian() { return rand.nextGaussian();
   }
    public int nextInt() { return rand.nextInt(); }
    public int nextInt(int range) { return rand.nextInt(range);
    }
    public float nextFloat() { return rand.nextFloat(); }
    public double nextDouble() { return rand.nextDouble(); } //
    return U(0,1)
    public double nextDouble(double a,double b) // Uniform(a,b)
    {
  return a+rand.nextDouble()*(b-a);
    }
    public int nextInt(double a,double b)
```

```
    {
  return (int)nextDouble(a,b);
    }

    /*
     * generate a random permutation from {0,1,...,n-1}
     */
    public static int[] getRandPerm(int n) {
  int rperm [] = new int[n];
  for(int i=0;i<n;i++) rperm[i]=i;
  for(int i=n;i>1;i--) {
      int k = rand.nextInt(i);
      int tmp = rperm[i-1];
      rperm[i-1] = rperm[k];
      rperm[k] = tmp;
  }
  return rperm;
    }
}
```

B.8 NTSPInstance.java

```
/**
 * Noisy Traveling Salesman Problem
 *
 * @author Feiyue Li(lify@math.umd.edu)
 */
public class NTSPInstance implements Cloneable{
    public double x[];
    public double y[];
    public int n;
    public int fixCount;
    double [][] dist;
    boolean [] change;
    int [] partialTour;


    public NTSPInstance(double [] x, double []y) {
  n = x.length;
  this.x = new double[n];
  this.y = new double[n];
  System.arraycopy(x,0,this.x,0,n);
  System.arraycopy(y,0,this.y,0,n);
  dist = new double[n][n];
```

```
for(int i=0;i<n;i++)
    for(int j=i+1;j<n;j++) {
  double d = Math.sqrt( (x[i]-x[j])*(x[i]-x[j])+(y[i]-y[j])*(
 y[i]-y[j]) );
  dist[i][j] = d;
  dist[j][i] = d;
    }
fixCount=0;
change = new boolean[n];
for(int i=0;i<n;i++) change[i] = true;
  }

  public NTSPInstance(double []x, double []y,int fix,boolean
 flag[],int goodTour[])
  {
this(x,y);
fixCount=fix;
change = new boolean[n];
for(int i=0;i<n;i++) {
    change[i] = flag[i];
}
partialTour = new int[fix];
int pos=0;
for(int i=0;i<fix;i++) {
    partialTour[i] = goodTour[i];
}

  }

  // this is only works for 2D problem
  public NTSPInstance(HPoint [] p) {
n = p.length;
x = new double[n];
y = new double[n];
for(int i=0;i<n;i++) {
    x[i] = p[i].coord[0];
    y[i] = p[i].coord[1];
}
dist = new double[n][n];
for(int i=0;i<n;i++)
    for(int j=i+1;j<n;j++) {
  double d = Math.sqrt( (x[i]-x[j])*(x[i]-x[j])+(y[i]-y[j])*(
 y[i]-y[j]) );
  dist[i][j] = d;
  dist[j][i] = d;
```

```
    }
  }

  public int size() { return n;}

  public double dist(int i,int j) {
return dist[i][j];
  }

  public String toString() {
StringBuffer sb = new StringBuffer();
sb.append("node=[");
for(int i=1;i<n;i++)
    sb.append(x[i]+"\t"+y[i]+";\n");
sb.append(x[0]+"\t"+y[0]+"];\n");
return sb.toString();
  }

  public Object clone() {
NTSPInstance t = null;
try {
    t = (NTSPInstance) super.clone();
} catch(CloneNotSupportedException e) {}
System.arraycopy(x,0,t.x,0,n);
System.arraycopy(y,0,t.y,0,n);
t.dist = new double[n][n];
for(int i=0;i<n;i++)
    for(int j=i+1;j<n;j++) {
  t.dist[i][j] = dist[i][j];
  t.dist[j][i] = dist[j][i];
    }
return t;
  }

  public HPoint[] getGeometry()
  {
double xmin=1e10,ymin=1e10,xmax=-1e10,ymax=-1e10;
for(int i=0;i<n;i++) {
    if(x[i]<xmin) xmin = x[i];
    if(x[i]>xmax) xmax = x[i];
    if(y[i]<ymin) ymin = y[i];
    if(y[i]>ymax) ymax = y[i];
}
double width = xmax - xmin;
double height = ymax - ymin;
```

```
 HPoint [] ret = new HPoint[2];
 double [] origin = { xmin,ymin };
 double [] size = { (xmax-xmin),(ymax-ymin) };
 ret[0] = new HPoint(origin);
 ret[1] = new HPoint(size);
 return ret;
   }

}
```

Appendix C

Open Vehicle Routing Problem Code

C.1 AdjMatrix.java

```
package ovrp;

/**
 * adjacent matrix for directed graph
 * @author: Feiyue Li (lify@math.umd.edu)
 * @version: 0.0.1
 */
public class AdjMatrix {
    private Graph g;
    // inner class for the adjacent list
    private adjList[] adj_mat;
    private int dim ;
    class adjList {
  boolean connected[];
  int edgeCount;
  public adjList(int size) {
      connected = new boolean[size];
      for(int i=0;i<size;i++) connected[i] = false;
      edgeCount = 0;
  }

  public boolean adjacentTo(int i) {
      return connected[i];
  }

  public void connect(int i) {
      connected[i] = true;
      edgeCount++;
  }
```

```
public void disconnect(int i) {
    connected[i] = false;
    edgeCount--;
}
  }

  public AdjMatrix(Graph g) {
this.g = g;
adj_mat = new adjList[g.size()];
for(int i=0;i<g.size();i++)
    adj_mat[i] = new adjList(g.size());
  }

  public AdjMatrix(int size) {
adj_mat = new adjList[size];
for(int i=0; i<size;i++)
    adj_mat[i] = new adjList(size);
  }

  public AdjMatrix(boolean [][] path) {
dim = path.length;
adj_mat = new adjList[dim];
for(int i=0;i<dim;i++) {
    adj_mat[i] = new adjList(dim);
    for(int j=0;j<dim;j++) {
  if(path[i][j])
      adj_mat[i].connect(j);
    }
}
  }

  public double value(){
double val = 0D;
for(int i=0; i< adj_mat.length;i++)
    for(int j=0;j<adj_mat.length;j++) {
  if( adj_mat[i].adjacentTo(j))
      val += g.distance(i,j);
    }
return val/2;
  }
  public Graph getGraph() { return g; }
  public boolean adjacentTo(int i,int j){ return adj_mat[i].
 adjacentTo(j); }
  public void connect(int i, int j) { adj_mat[i].connect(j);
```

```
    }
     public void disconnect(int i,int j) { adj_mat[i].disconnect
    (j); }
     public int oDegree(int i) { return adj_mat[i].edgeCount; }
    // the outdegree of node i
     public boolean accessible(int s,int t) {
   //return true if we can reach t staring from s
   boolean accessable = false;
   for(int i=0; i<g.size(); i++) {
       if(adj_mat[s].adjacentTo(i)){
     if(i==t){
         accessable = true;
     }
         else accessable = accessible(i,t);
     if(accessable) break;
       }
   }
   return accessable;
     }

     //warshall algorithm
     public void transclosure(boolean path[][])
     {
   int n = dim;
   //boolean p[][] = new boolean[n][n];
    for(int i = 0; i <n ; i++)
          for(int j = 0; j < n; j++)
              path[i][j] = adj_mat[i].adjacentTo(j);

     for(int i = 0;i <n; i++)
       for(int j = 0;j < n; j++)
          if(path[i][j]) {
        for(int k = 0; k < n; k++)
          if(path[j][k])
        path[i][k] = true;
          }
     }
}
```

C.2 OVRPSolution.java

```
package ovrp;
import java.util.*;
import org.apache.log4j.Logger;
import org.apache.log4j.BasicConfigurator;
```

```
import org.coinor.opents.*;
//4.14/05
// to use neighbor list in 1-0 move, 1-1 move, 2-opt move
// the route id should be a field of the ListNode class, not
// the VRPNode b/c in clone we just made a shadow copy of the
   VRPNode
// so the change in route id in VRPNode will affect the best
   solution

public class OVRPSolution extends SolutionAdapter implements
   Solution, Cloneable {
    private boolean DEFINE_REGRET = true;//false;//true;
    private boolean useNBListInRTR = false;
    private boolean LIMIT_PERTURB = true;

    private boolean debug = false;//true;
    private Tour[] tours;
    private VRPGraph g;
    protected double distance;

    public int numTours;
    static Logger logger;
    private int scores[][];
    static {
    logger = Logger.getLogger(OVRPTest.class);
    }

    AdjMatrix m;
    RTRTabuList tabuList;

    public OVRPSolution(VRPGraph g) {
  this.g = g;
  tabuList = new RTRTabuList();
  scores = new int[g.size()][g.size()];
    }


    public VRPGraph getGraph() { return g; }

    public void initialize(Algorithm alg) {
  System.out.println("using "+alg +" algorithm to get an
   initial solution!");
  AdjMatrix adj = new AdjMatrix(g);
  alg.solve(adj);
  distance = 0D;
```

```
init_solution(adj);
  }

  public void init_solution(AdjMatrix adj) {
// number of outgoing links of the depot
numTours = adj.oDegree(0);
logger.info("total "+numTours +" vehicles used");
tours = new Tour[numTours];
int count=0;
for(int i=1; i<g.size();i++) {
    if(adj.adjacentTo(0,i)) {
  tours[count] = new Tour(g.nodes[0],g);
  kickoff(tours[count],i,adj);
  count++;
    }
}
for(int i=0;i<numTours;i++)
   distance += tours[i].getDistance();
  }

  private void kickoff(Tour t, int node, AdjMatrix adj) {
t.add(g.nodes[node]);
for(int i=1;i<g.size();i++) {
    if(adj.adjacentTo(node,i)) {
  kickoff(t,i,adj);
  break;
    }
}
  }

  public double evaluate() {
distance = 0D;
for(int i=0;i<tours.length;i++)
    distance += tours[i].getDistance();
return distance;
  }

  public double getDistance() {
return evaluate();
  }

  public String toString() {
String ret = new String();
for(int i=0; i<tours.length;i++)
   ret += tours[i].toString();
```

```
ret +="\n Total distance traveled= "+ getTravelDistance()+" #
 vehicles used="+tours.length+"("+numTours+")\n";
return ret;
  }

  public double getTravelDistance() {
return (evaluate()-(g.size()-1)*g.getServiceTime());
  }

  public double getOriginalDistance() {
double ret = 0D;
for(int i=0;i<numTours;i++)
    ret += tours[i].getOriginalDistance();
return ret*g.maxDist;
  }

  public Object clone() {
OVRPSolution copy =null;
    copy = (OVRPSolution) super.clone();
copy.tours = (Tour[])tours.clone();
for(int i=0;i<tours.length;i++) {
    copy.tours[i] = (Tour) tours[i].clone();
}
return copy;
  }

  // full version of 2-opt
  public double twOpt() {
boolean keepgoing;
boolean flag = true;
while(flag) {
    //keepgoing = false;
    flag =false;
    for(int i=0;i<tours.length;i++) {
  for(int j=0;j<tours.length;j++) {
    if(j==i) tours[i].twOpt();
         else if(!tours[j].isEmpty()) {
        keepgoing = tours[i].twOpt(tours[j]);
        if(keepgoing) flag = true;
         }
      }
    }
}
//*/
return evaluate();
```

```
  }

  public double twoOneMove() {
//logger.info("enter 2-1 move");
boolean keepgoing = false;
boolean flag = true;
while(flag) {
    flag = false;
    for(int i=0; i<tours.length;i++)
  for(int j=0; j<tours.length;j++) {
      if(j==i) continue;
      if(!tours[j].isEmpty())
    keepgoing = tours[i].twoOneMove(tours[j]);
      if(keepgoing) flag = true;
  }
}
//logger.info("done 2-1 move");
return evaluate();

  }

  public double threeOneMove(){
//logger.info("enter 2-1 move");
boolean keepgoing = false;
boolean flag = true;
while(flag) {
    flag = false;
    for(int i=0; i<tours.length;i++)
  for(int j=0; j<tours.length;j++) {
      if(j==i) continue;
      if(!tours[j].isEmpty())
    keepgoing = tours[i].threeOneMove(tours[j]);
      if(keepgoing) flag = true;
  }
}
//logger.info("done 2-1 move");
return evaluate();

  }
  public double threeTwoMove(){
//logger.info("enter 2-1 move");
boolean keepgoing = false;
boolean flag = true;
while(flag) {
    flag = false;
```

```
    for(int i=0; i<tours.length;i++)
  for(int j=0; j<tours.length;j++) {
      if(j==i) continue;
      if(!tours[j].isEmpty())
    keepgoing = tours[i].threeTwoMove(tours[j]);
      if(keepgoing) flag = true;
  }
}
//logger.info("done 2-1 move");
return evaluate();

  }
  public double onePointMove() {
boolean keepgoing = true;
boolean flag = true;
//while(keepgoing) {
while(flag) {
    flag = false;
    keepgoing = false;
    boolean reduced = false;
    for(int i=0;i<tours.length;i++){
  if(!tours[i].isEmpty()) {
      for(int j=i;j<tours.length;j++) {
    if(j==i) tours[i].onePointMove();
      else {
    if(!tours[j].isEmpty()) keepgoing = tours[i].onePointMove
 (tours[j]);
    if(tours[i].isEmpty()) {
        delEmptyTour();
        //logger.info("reducing one vehicle");
        flag = true;
        reduced = true;
        break;
    }
      }
      if(keepgoing)
    flag = true;
    //break;
      }
      if(reduced) break;
  }
  //if(keepgoing) break;
    }
}
return evaluate();
```

```
  }

  public double orOptMove() {
boolean keepgoing = true;
while(keepgoing) {
    keepgoing = false;
    boolean reduced = false;
    for(int i=0;i<tours.length;i++){
  if(!tours[i].isEmpty()) {
      for(int j=i;j<tours.length;j++) {
    if(j==i) {
        tours[i].orOptMove();
    }
    else if(!tours[j].isEmpty()){
        keepgoing = tours[i].orOptMove(tours[j]);
        if(tours[i].isEmpty()) {
      reduced = true;
      delEmptyTour();
      break;
        }
    }
      }
  }
  if(reduced) break;
    }
}
return evaluate();
  }

  public double threeOptMove() {
boolean keepgoing = true;
while(keepgoing) {
    keepgoing = false;
    boolean reduced = false;
    for(int i=0;i<tours.length;i++){
  if(!tours[i].isEmpty()) {
      for(int j=i;j<tours.length;j++) {
    if(j==i) tours[i].threeOptMove();
    else if(!tours[j].isEmpty()){
        keepgoing = tours[i].threeOptMove(tours[j]);
        if(tours[i].isEmpty()) {
      delEmptyTour();
      reduced = true;
      break;
        }
```

```
    }
      }
  }
  if(reduced) break;
    }
}
return evaluate();
  }

  public double twoPointMove() {
boolean keepgoing = true;
boolean flag = true;
while(flag) {
    flag = false;
    keepgoing = false;
    boolean reduced = false;
    for(int i=0;i<tours.length;i++) {
  if(!tours[i].isEmpty()) {
      for(int j=0;j<tours.length;j++) {
    if(j==i) tours[i].twoPointMove();
    else if(!tours[j].isEmpty()) {
        keepgoing = tours[i].twoPointMove(tours[j]);
        if(tours[i].isEmpty()) {
      reduced = true;
      flag = true;
      delEmptyTour();
      break;
        }
    }
    if(keepgoing) flag=true;//break;
      }
  }
  if(reduced) break;
    }
}
return evaluate();
  }

  private boolean combineTours() {
boolean success = false;
final double serviceTime = g.getServiceTime();
final double disCon = g.getDisConstraint();
int cap = g.getCapacity();
for(int i=0;i<tours.length;i++) {
    if(!tours[i].isEmpty()) {
```

```
  for(int j=0;j<tours.length;j++){
      if(j==i) continue;
      if(!tours[j].isEmpty()) {
    if(tours[i].getLoad() + tours[j].getLoad()<= cap ) {
        // estimate the distance by adding the service time
        if(tours[j].getDistance()+ tours[i].numElements*
 serviceTime < disCon) {
      success=doCombineTour(tours[j],tours[i]);
      if(success) {
          delEmptyTour();
      }
        }
    }
      }
      if(success) break;
  }
    }
    if(success) break;
}
return success;
  }

  // put tour j into tour i
  // we test every two noees of tour i and try to insert j
 into it
  private boolean doCombineTour(Tour i, Tour j) {
boolean success = false;
ListNode first = j.getFirstNode();
ListNode last = j.getLastNode();
double chainDist = j.getChainDistance();
double newdist ;
Enumeration elm = i.elements();
ListNode current=null;
while(elm.hasMoreElements()) {
    current = (ListNode) elm.nextElement();
    ListNode prev = (ListNode) current.getPrevious();
    newdist = i.getDistance() + chainDist + g.distance(prev,
 first) + g.distance(last,current) - g.distance(prev,current)
 ;
    if(newdist <= g.getDisConstraint()) {
  i.load += j.getLoad();
  prev.setNext(first); first.setPrevious(prev);
  last.setNext(current); current.setPrevious(last);
  i.distance = newdist;
  i.numElements += j.numElements;
```

```
  i.checkDistance();
  j.numElements=0;
  j.distance = 0D;
  j.load = 0;
  success = true;
  break;
    }
} //end while
if(!success) { // try to put tour j in the end of  tour i
    newdist = i.getDistance() + chainDist + g.distance(
 current,first);
    if(newdist <= g.getDisConstraint()) {
  i.load += j.getLoad();
  current.setNext(first); first.setPrevious(current);
  i.distance = newdist;
  i.numElements += j.numElements;
  i.checkDistance();
  j.numElements=0;
  j.distance = 0D;
  j.load = 0;
  success = true;
    }
}
return success;
  }

  public void cleanUp() {
double d1,d2,d3,d4,d5,d6,d7,d8;
do {
    d1 = twOpt();
    d3 = twoPointMove();
    d5 = threeOptMove();
    d4 = orOptMove();
    d2 = onePointMove();
    reloadHead();
    d8 = threeTwoMove();
    d6 = twoOneMove();
    d7 = threeOneMove();
    delEmptyTour();
    Pause();
} while(Math.abs(d1-d7)>1e-5);
  }

  public void clean() {
double d1,d2,d3,d4,d5,d6,d7,d8;
```

```
if(numTours>g.getMinVehicles()) while(combineTours()) ;
    d4 = orOptMove();
    d1 = twOpt();
    d2 = onePointMove();
    d3 = twoPointMove();
    d5 = threeOptMove();
    reloadHead();
    d8 = threeTwoMove();
    d7 = threeOneMove();
    d6 = twoOneMove();
  }

  protected void delEmptyTour() {
numTours = 0;
List tourList = new ArrayList(tours.length);
for(int i=0; i<tours.length;i++){
    if(!tours[i].isEmpty()) {
  numTours++;
  tourList.add(tours[i]);
    }
}
tours = (Tour[]) tourList.toArray(new Tour[0]);
  }

  public void reloadHead() {
if(numTours!=tours.length) logger.error("numTours is not
 consistent!");
for(int i=0;i <tours.length; i++)
    tours[i].reloadHead();
  }

  public boolean twOptRTR(Record r) {
boolean flag[] = new boolean[g.size()];
int count = 0;
boolean moved = false;
while( count<g.size()-1 ) {
    for(int i=0;i<tours.length;i++) {
  if(tours[i].isEmpty()) continue;
  Enumeration elm = tours[i].elements();
  while(elm.hasMoreElements()) {
      ListNode current = (ListNode) elm.nextElement();
      if(!flag[current.getId()]){
    flag[current.getId()]=true;
    count++;
      }
```

```
      else continue;
      RTRInfo bestInfo = new RTRInfo(null,null,-1e8);
      for(int j=0;j<tours.length;j++) {
    if(tours[j].isEmpty()) continue;
    RTRInfo info ;
    if(j==i) info = tours[i].get2OptRTRInfo(current,r);
    else info = tours[i].get2OptRTRInfo(current,tours[j],r);
    if(info.getSaving() > bestInfo.getSaving()) {
        bestInfo = info;
    }
      }
      if(bestInfo.getNode()!=null) {
    tours[i].do2OptRTR(current,bestInfo);
    Pause();
    r.setCurrent(evaluate());
    moved = true;
    break;
      }
  }//while
    }//for
}//while
return moved;
  }

  public boolean onePointMoveRTR(Record r) {
boolean flag[] = new boolean[g.size()];
int count = 0;
boolean moved = false;
while( count<g.size()-1 ) {
    for(int i=0;i<tours.length;i++) {
  if(tours[i].isEmpty()) continue;
  Enumeration elm = tours[i].elements();
  while(elm.hasMoreElements()) {
      ListNode current = (ListNode) elm.nextElement();
      if(!flag[current.getId()]){
    flag[current.getId()]=true;
    count++;
      }
      else continue;
      RTRInfo bestInfo = new RTRInfo(null,null,-1e8);
      for(int j=0;j<tours.length;j++) {
    if(tours[j].isEmpty()) continue;
    RTRInfo info ;
    if(j==i) info = tours[i].getOnePointMoveRTRInfo(current,r
 );
```

```
    else info = tours[i].getOnePointMoveRTRInfo(current,tours
 [j],r);
    if(info.getSaving() > bestInfo.getSaving()) {
        bestInfo = info;
    }
      }
      if(bestInfo.getNode()!=null) {
    tours[i].doOnePointMoveRTR(current,bestInfo);
    Pause();
    r.setCurrent(evaluate());
    moved = true;
    break;
      }
  }//while
    }//for
}//while
return moved;
  }

  public boolean twoPointMoveRTR(Record r) {
boolean flag[] = new boolean[g.size()];
int count = 0;
boolean moved = false;
while( count<g.size()-1 ) {
    for(int i=0;i<tours.length;i++) {
  if(tours[i].isEmpty()) continue;
  Enumeration elm = tours[i].elements();
  while(elm.hasMoreElements()) {
      ListNode current = (ListNode) elm.nextElement();
      if(!flag[current.getId()]){
    flag[current.getId()]=true;
    count++;
      }
      else continue;
      RTRInfo bestInfo = new RTRInfo(null,null,-1e8);
      for(int j=0;j<tours.length;j++) {
    if(tours[j].isEmpty()) continue;
    RTRInfo info ;
    info = tours[i].getTwoPointMoveRTRInfo(current,tours[j],r
 );
    if(info.getSaving() > bestInfo.getSaving()) {
        bestInfo = info;
    }
      }
      if(bestInfo.getNode()!=null) {
```

```
    tours[i].doTwoPointMoveRTR(current,bestInfo);
    Pause();
    r.setCurrent(evaluate());
    moved = true;
    break;
      }
  }//while
    }//for
}//while
return moved;
  }

  private void checkDistance(String where) {
for(int i=0;i<tours.length;i++)
    tours[i].checkDistance(where+"tour "+i);
  }

  public OVRPSolution RTR(double perc) {
return RTR();
  }

  public OVRPSolution RTR() {
double perc = 0.01;
Random rand = new Random();
int counter = 0;
      int I = 1;
      clean();
      OVRPSolution bestSol = null;
          bestSol = (OVRPSolution) clone();
      double bestLength = bestSol.getDistance();
int minVehicle = bestSol.numTours;
      Record r = new Record(bestLength,perc);
      r.setCurrent(bestLength);
      while (true) {
          boolean improved = false;
          for (int i = 0; i < I; i++) {
  //logger.info("before RTR there are "+ numTours +" vehicles
 ");
  boolean opm = onePointMoveRTR(r);
  boolean tom = twOptRTR(r);
  //boolean tom = true;
  boolean tpm = twoPointMoveRTR(r);
  //logger.info("after RTR there are "+ numTours +" vehicles
 ");
              if (!opm && !tom && ! tpm ) {
```

```
  //if(!tom) {
                    logger.info("no movement in loop I. Quit
 loop I");
                    break;
                } else {
      //logger.info("i="+i);
                    if (numTours < minVehicle || (numTours==
 minVehicle && getDistance() < bestSol.getDistance())) {
                        improved = true;
                            bestSol = (OVRPSolution) clone();
                        r.setCurrent(getDistance());
    //r.setDeviation(perc*getTravelDistance());
    minVehicle = numTours;
                    }
                }
            }
    //logger.info("check distance after rtr");
    checkDistance("after rtr");
            clean();
    //logger.info("[in RTR()] after clean "+ numTours +"
 vehicles used minVehicle="+minVehicle);
            double currentLength =getDistance(); //getDistance
 ();
            if (numTours < minVehicle || (numTours==minVehicle
 && currentLength < bestSol.getDistance() -1e-5)) {
                    bestSol = (OVRPSolution) clone();
      r.setRecord(currentLength);
      minVehicle = numTours;
      counter = 0;
            }
    r.setCurrent(currentLength);
    counter++;
    if(counter>5) break;
}
bestSol.cleanUp();
return bestSol;
  }

  public void updateScores() {
for(int i=0; i<numTours; i++) {
    Enumeration elm = tours[i].elements();
    ListNode cur = tours[i].depot;
    while(elm.hasMoreElements()) {
  ListNode next = (ListNode) elm.nextElement();
  scores[cur.getId()][next.getId()]++;
```

```
  cur = next;
    }
}
  }

  public void printGoodEdges(int n) {
logger.info("total number of solutions =" +n);
for(int i=0;i<g.size();i++)
    for(int j=0;j<g.size();j++) {
  if(i==j) continue;
  if(scores[i][j]>n/2) logger.info(i+"-->"+j+"("+((double)
 scores[i][j])/n+")");
    }
  }

  public void purturb() {
class Ratio implements Comparable {
    double r;
    Tour t;
    ListNode id;
    Ratio(ListNode id,double r, Tour t) { this.id = id ; this
 .r = r; this.t = t; }
    public int compareTo(Object o) {
  if( r<((Ratio) o).r) return -1;
  else if(r > ((Ratio)o).r) return 1;
  else return 0;
    }
}
Ratio ratio [] = new Ratio[g.size()-1];
for(int i=0;i<tours.length;i++) {
    Enumeration elm = tours[i].elements();
    while(elm.hasMoreElements()) {
  ListNode cur = (ListNode) elm.nextElement();
  ListNode prev = (ListNode) cur.getPrevious();
  ListNode next = (ListNode) cur.getNext();
  double saving = g.distance((VRPNode)prev.data,(VRPNode)cur.
 data);
  if(next!=null) saving += g.distance((VRPNode)cur.data,(
 VRPNode)next.data)
        -g.distance((VRPNode)prev.data,(VRPNode)next.data);
  double r = ((VRPNode)cur.getData()).getDemand()/saving;
  ratio[cur.getId()-1] = new Ratio(cur,r,tours[i]);
    }
}
Arrays.sort(ratio); //sort the ratio in the ascending order
```

```
int numPurturb = Math.max((int) g.size()/10,5);
for(int i=0;i<numPurturb;i++)
    insertNode(ratio[i].id,ratio[i].t);
  }

  public void randPerturb1() {
int numPerturb = Math.min((int)g.size()/10,15);
MyRandom rand = new MyRandom();
int randPerm[] = rand.getRandPerm(g.size());
int perturbed = 0;
for(int i=0; i< g.size(); i++) {
    int nodeToPerturb = randPerm[i];
    if(nodeToPerturb==0) continue;
    for(int j=0;j<numTours;j++){
  ListNode cur = tours[j].locate(nodeToPerturb);
  if(cur!=null && insertNode(cur,tours[j])) {
      perturbed++;
      break;
  }
    }
    if(perturbed>numPerturb) break;
}
  }

  public void randPerturb() {
int numPerturb = (int) g.size()/numTours;
int rnode [] = new int[numPerturb];
MyRandom rand = new MyRandom();
int randPerm[] = rand.getRandPerm(g.size());
int perturbed = 0;
for(int i=0; i<g.size(); i++){
    int nodeToPerturb = randPerm[i];
    if(nodeToPerturb == 0) continue;
    else {
  rnode[perturbed++] = nodeToPerturb;
  deleteNode(nodeToPerturb);
  if(perturbed==numPerturb) break;
    }
}
clean();
for(int i=0; i<numPerturb;i++)
    insertNode(rnode[i]);
  }
  // try to delete one route and insert all the nodes into
 other routes
```

```
  // in order to minimize the number of vehicles used
  public void deleteOneRoute() {
Tour minTour = null;
int minLoad = 1000000;
for(int i=0;i<numTours;i++)
    if(tours[i].getLoad()<minLoad) {
  minLoad = tours[i].getLoad();
  minTour = tours[i];
    }
Enumeration elm = minTour.elements();
int nodeToDelete[] = new int[minTour.numElements];
int i=0;
while(elm.hasMoreElements()) {
    ListNode next = (ListNode)elm.nextElement();
    nodeToDelete[i++]=next.getId();
}
for(i=0;i<nodeToDelete.length;i++)
    deleteNode(nodeToDelete[i]);
delEmptyTour();
//clean();
for(i=0;i<nodeToDelete.length;i++)
    insertNode(nodeToDelete[i]);
clean();

  }
  public void ruinRecreate() {
final int p = Math.min((int)g.size()/8,g.MAX_NB-5);
MyRandom rand = new MyRandom();
 // select a node randomly and delete p closest neighbor from
  the tour
 // reinsert them in the least cost fashion
int neighbor[] = new int[p+1];
int nid = -1;
if(LIMIT_PERTURB) {
    nid = rand.nextInt(g.nodeToPerturb.length);
    nid = g.nodeToPerturb[nid];
}
else nid = rand.nextInt(1,g.size());
int count = 0;
//logger.info("total "+ (p+1) +" deleting nodes");
for(int i=0;i<g.MAX_NB;i++) {
    int nb = g.NBList[nid][i];
    if(nb>0){
  neighbor[count++] = nb;
  //logger.info("adding "+nb +" into deleting list");
```

```
    }
    if(count==p) break;
    //logger.info("node "+ nid+"'s nearest "+p+" neighbor "+
 neighbor[i]);
}
neighbor[p]=nid;
for(int i=0;i<=p;i++){
    if(neighbor[i]>0)
  deleteNode(neighbor[i]);
    else {
  logger.error("strange !");
  System.exit(1);
    }
}
// try to delete a route with max distance
clean();
if(DEFINE_REGRET) regretInsertNodes(neighbor);
else bestInsertNodes(neighbor);
  }

  public void ruinRecreate(int nid) {
//logger.info("run R&R");
final int p = Math.min((int)g.size()/8,g.MAX_NB/2);
//final int p = Math.min(15,g.MAX_NB-1);
/*
 * select a node randomly and delete p closest neighbor from
 the tour
 * reinsert them in the least cost fashion
 */
int neighbor[] = new int[p+1];
MyRandom rand = new MyRandom();
int count = 0;
//logger.info("total "+ (p+1) +" deleting nodes");
for(int i=0;i<g.MAX_NB;i++) {
    int nb = g.NBList[nid][i];
    if(nb>0){
  neighbor[count++] = nb;
  //logger.info("adding "+nb +" into deleting list");
    }
    if(count==p) break;
    //logger.info("node "+ nid+"'s nearest "+p+" neighbor "+
 neighbor[i]);
}
neighbor[p]=nid;
//for(int i=0;i<=p;i++)
```

```
 //    logger.info(neighbor[i]+",");
for(int i=0;i<=p;i++){
    if(neighbor[i]>0)
  deleteNode(neighbor[i]);
    else {
  logger.error("strange!");
  System.exit(1);
    }
}
clean();
int []randPerm = rand.getRandPerm(p+1);
for(int i=0;i<=p;i++)
    insertNode(neighbor[randPerm[i]]);
//logger.info("after R&R the solution is:");
//logger.info(this);

  }
  // probabilistic ruin and recreate
  public void probRuinRecreate(double p) {
MyRandom rand = new MyRandom();
int [] nodeToDelete = new int[g.size()];
int count=0;
for(int i=1;i<g.size();i++)
    if(rand.nextDouble()<p) nodeToDelete[count++] = i;


for(int i=0;i<count;i++)
    deleteNode(nodeToDelete[i]);
clean();
for(int i=0;i<count;i++)
    insertNode(nodeToDelete[i]);
  }

  private void deleteNode(int nid) {
VRPNode node = g.nodes[nid];
for(int i=0;i<tours.length;i++) {
    if(tours[i].contains(node)) {
  tours[i].deleteNode(node);
  tours[i].checkDistance();
  break;
    }
}
  }
```

```
  private void insertNode(int nid) {
//logger.info("inserting node "+nid);
VRPNode node = g.nodes[nid];
int d = node.getDemand();
Tour bestTour = null;
double bestCost = 1e10;
for(int i=0;i<tours.length;i++) {
    if(tours[i].getLoad() + d <= g.getCapacity()) {
  double cost = tours[i].getLeastCost(node);
  if(cost + tours[i].getDistance() <= g.getDisConstraint()
    && cost < bestCost) {
      bestTour = tours[i];
      bestCost = cost;
  }
    }
}
if(bestTour!=null){
    bestTour.leastCostInsertion(node);
    //logger.info(bestTour);
    bestTour.checkDistance();
}
else {
    Tour newTour = new Tour(g);
    newTour.add(g.nodes[nid]);
    ArrayList t = new ArrayList(numTours++);
    for(int i=0;i<tours.length;i++)
  t.add(tours[i]);
    t.add(newTour);
    tours = (Tour [])t.toArray(new Tour[0]);
}
evaluate();
  }

  private boolean insertNode(ListNode cur, Tour t) {
Tour bestTour = null;
double bestCost = 1e10;
VRPNode insNode = (VRPNode)cur.getData();
for(int i=0;i<tours.length;i++) {
    if(tours[i]==t||tours[i].isEmpty()) continue;
    if(insNode.getDemand() + tours[i].getLoad()>g.getCapacity
 ()
      ||tours[i].getDistance()+g.getServiceTime()>g.
 getDisConstraint()) continue;
    else {
  double insCost = tours[i].getLeastCost(insNode);
```

```
  if(insCost+tours[i].getDistance()<=g.getDisConstraint()) {
      if(insCost<bestCost) {
    bestCost = insCost;
    bestTour = tours[i];
      }
  }
    }
}
if(bestTour!=null) {
    ListNode prev = (ListNode)cur.getPrevious();
    ListNode next = (ListNode)cur.getNext();
    double saving = g.distance((VRPNode)prev.data,insNode);
    if(next!=null) saving += g.distance(insNode,(VRPNode)next
 .data)
          -g.distance((VRPNode)prev.data,(VRPNode)next.data);
    t.distance -= saving+g.getServiceTime();
    t.numElements--;
    t.load -= insNode.getDemand();
    prev.setNext(next);
    if(next!=null) next.setPrevious(prev);
    bestTour.leastCostInsertion(insNode);
    evaluate();
    return true;
}
return false;
  }
  // contract of abastract class Algorithm
  public void solve(AdjMatrix adj) {
m = adj;
Algorithm initAlg = AlgorithmFactory.getAlgorithm("Sweep");
g = (VRPGraph)adj.getGraph();
initAlg.solve(adj);
//pause();
logger.info("solution initialized");
init_solution(m);
RTR(0.01);
  }

  private void Pause() {
/*
update_adj_matrix();
//pause();
*/
  }
```

```
  private void update_adj_matrix() {
for(int i=0;i<g.size();i++) {
    for(int j=0;j<g.size();j++) {
  m.disconnect(i,j);
    }
}
for(int i=0;i<tours.length;i++) {
    Enumeration elm = tours[i].elements();
    ListNode cur = tours[i].getHead();
    while(elm.hasMoreElements()) {
  ListNode next = (ListNode) elm.nextElement();
  m.connect(cur.getId(),next.getId());
  cur = next;
    }
}
  }

  public Tour[] getTours() { return tours; }

  public int getNumTours() {
int ret = 0;
for(int i=0; i<tours.length;i++) {
    if(!tours[i].isEmpty()) ret++;
}
logger.assertLog(ret==numTours,"numTours is not consistent");
return numTours;
  }

  // recomputing the distance due to smooth
  public void reevalulate() {
distance = 0D;
for(int i=0;i<numTours;i++)
    distance += tours[i].evaluate();
  }

  public void optimize_solution(){
for(int i=0;i<numTours;i++) {
    tours[i].optimize_tour();
}
evaluate();
  }


  private void findTopTwoMin(double [] cost, double [] winner
 ) {
```

```
BinaryHeap heap = new BinaryHeap(cost.length);
try {
    for(double val : cost)
  heap.insert(new Double(val));
} catch(Overflow e) {}
heap.buildHeap();
winner[0] = ((Double) heap.deleteMin()).doubleValue();
winner[1] = ((Double) heap.deleteMin()).doubleValue();
  }

  private double findMin(double [] cost) {
double min = 1e10;
for(double val : cost) {
    if(val<min) min = val;
}
return min;
  }

  private void bestInsertNodes(int nid[]) {
final double disCon = g.getDisConstraint();
final int capacity = g.getCapacity();
int nRows = nid.length;
boolean [] done = new boolean[nRows];
int insertedSofar = 0;
while(insertedSofar < nRows) {
    int nCols = tours.length;
    double cost[][] = new double[nRows][nCols];
    double [] minCost = new double[nRows];
    double [] tmp = new double[nCols];
    for(int i=0; i< nRows; i++) {
  if(done[i]) continue;
  for(int j=0; j< nCols; j++) {
      cost[i][j] = tours[j].getLeastCost(g.nodes[nid[i]]);
      if(cost[i][j] + tours[j].getDistance() > disCon
        || g.nodes[nid[i]].getDemand() + tours[j].getLoad() >
  capacity)
    cost[i][j] = 1e5;
  }
    }
    // calculate the regret-2 value, which is f2-f1
    for(int i=0; i <nRows; i++) {
  if(done[i]) continue;
  for(int j = 0 ; j < nCols; j++) tmp[j] = cost[i][j];
  minCost[i] = findMin(tmp);
    }
```

```
    // find the maximum regret-2 value
    int winner = -1;
    double min = 1e6;
    for(int i=0; i<nRows; i++) {
  if(!done[i] && minCost[i]<min) {
      min = minCost[i];
      winner = i;
  }
    }
    //logger.info("max="+max+",winner="+winner);
    logger.assertLog(winner>=0,"no winner found in "
 bestInsertNodes");
    done[winner] = true;
    insertedSofar++;
    //logger.info("insert node "+nid[winner]);
    insertNode(nid[winner]);
}
  }

  // use regret-2 criteria to determine the order deleted
 nodes are re-inserted
  private void regretInsertNodes(int nid[]) {
final double disCon = g.getDisConstraint();
final int capacity = g.getCapacity();
int nRows = nid.length;
int insertedSofar = 0;
//flag for the ith node
boolean [] done = new boolean[nRows];

while(insertedSofar < nRows) {
    int nCols = tours.length;
    double cost[][] = new double[nRows][nCols];
    for(int i=0;i<nRows;i++) {
  if(done[i]) continue;
  for(int j=0; j<tours.length;j++){
      cost[i][j] = tours[j].getLeastCost(g.nodes[nid[i]]);
      if(cost[i][j] + tours[j].getDistance() > disCon || g.
 nodes[nid[i]].getDemand() + tours[j].getLoad() > capacity)
    cost[i][j] = 1e5;
  }
    }

    // calculate the regret-2 value, which is f2-f1
    double top2 [] = new double[2];
    double tmp [] = new double[nCols];
```

```
      double regret[] = new double[nRows];

      for(int i=0; i <nRows; i++) {
    if(done[i]) continue;
    for(int j = 0 ; j < nCols; j++) tmp[j] = cost[i][j];
    findTopTwoMin(tmp,top2);
    if(top2[0]>1e4 && top2[1]>1e4) {
        ;
    }
    regret[i] = top2[1] - top2[0];
    logger.assertLog(regret[i]>=0,"the regret-2 value is
   negative positive "+regret[i]);
      }

      // find the maximum regret-2 value
      int winner = -1;
      double max = -1D;
      for(int i=0; i<nRows; i++) {
    if(!done[i] && regret[i]>max) {
        max = regret[i];
        winner = i;
    }
      }
      logger.assertLog(winner>=0,"no winner found in regret-2")
   ;
      done[winner] = true;
      insertedSofar++;
      //logger.info("insert node "+nid[winner]);
      insertNode(nid[winner]);
  }

    }
}
```

C.3 Tour.java

```
/*
 * Created on 2003-10-15
 *
 * To change the template for this generated file go to Window
   - Preferences -
 * Java - Code Generation - Code and Comments
 */
package ovrp;
```

```
import java.util.Enumeration;
import java.util.NoSuchElementException;
import org.apache.log4j.Logger;
/**
 * @author lify
 *
 * To change the template for this generated type comment go to
    Window -
 * Preferences - Java - Code Generation - Code and Comments
 */

/**
 * A tour is a cicular double-linked list
 */
public class Tour implements Cloneable {
  static int TID = 0;
  boolean debug = false;
  protected ListNode depot;
  protected int numElements;
  protected double distance;
  protected int load;
  private VRPGraph g;
  private int tourID;

  static Logger logger = Logger.getLogger(Tour.class);
  private Tour() {
  }

  // d must be the depot node
  public Tour(Node d, VRPGraph g) {
    depot = new ListNode(d);
    depot.setPrevious(null);
    depot.setNext(null);
    this.g = g;
    numElements = 0;
    distance = 0D;
    load = 0;
    tourID = TID++;
  }

  public Tour(VRPGraph g) {
    //logger.error("this constructor Tour(g) should not be
   called");
    depot = new ListNode(g.getDepot());
```

```
    depot.setPrevious(null);
    depot.setNext(null);
    this.g = g;
    numElements = 0;
    distance = 0D;
    load = 0;
    tourID = TID++;
  }

  public int getTourID() {
    return tourID;
  }

  protected void reverse(Object ofrom, Object oto) {
    ListNode from = find(ofrom);
    ListNode to = find(oto);
    reverse(from, to);
  }

  // if the object is contained in the list
  private ListNode find(Object obj) {
    ListNode ret = null;
    Enumeration e = elements();
    while (e.hasMoreElements()) {
      ListNode node = (ListNode) e.nextElement();
      if (node.getData().equals(obj)) {
        ret = node;
        break;
      }
    }
    return ret;

  }

  public void deleteNode(Object n) {
    Enumeration elm = elements();
    while(elm.hasMoreElements()) {
      ListNode node = (ListNode) elm.nextElement();
      if(node.getData().equals(n)) {
        ListNode prev = (ListNode) node.getPrevious();
        ListNode next = (ListNode) node.getNext();
        prev.setNext(next);
        // update demand
        load -= node.getDemand();
        // update distance
```

```
      double saving = g.distance(prev,node);
      if(next!=null) {
        saving += g.distance(node,next) - g.distance(prev,
 next);
        next.setPrevious(prev);
      }
      distance -= saving;
      distance -= g.getServiceTime();
      // update number of nodes
      numElements--;

      break;
    }
  }
  checkDistance();
}
// reverse the chain between from and to, inclusively
public void reverse(ListNode from, ListNode to) {
  logger.assertLog(from!=null,"in reverse from==null");
  logger.assertLog(to!=null,"in reverse to==null");
  logger.assertLog(isAhead(from,to),"in reverse from is not
 ahead of to");
  ListNode prev = (ListNode) from.getPrevious();
  ListNode next = (ListNode) to.getNext();
  // prev->to->...->from->next
  ListNode iterator = from;
  while (iterator != to) {
    logger.assertLog(iterator!=null,"iterator is null");
    ListNode tmp = (ListNode) iterator.getNext();
    iterator.swap();
    iterator = tmp;
  }
  to.swap();
  prev.setNext(to);
  to.setPrevious(prev);
  from.setNext(next);
  if(next!=null) next.setPrevious(from);
}

public void reverse(int i, int j) {
  ListNode from = (ListNode) elementAt(i);
  ListNode to = (ListNode) elementAt(j);
  if (i < j)
    reverse(from, to);
  else
```

```
    reverse(to, from);
}

public void twOpt() {
  while (twOptStep());
}

public boolean twOpt(ListNode n) {
  return twOptSingleStep(n);
}

public boolean twOpt(Tour t) {
  boolean improved = true;
  while(improved) {
    improved = false;
    Enumeration elm = elements();
    while(elm.hasMoreElements()) {
      ListNode from = (ListNode) elm.nextElement();
      improved = twOpt(from,t);
      if(improved) break;
    }
  }
  return improved;
}

public boolean onePointMove(Tour t) {
  boolean improved = true;
  while(improved) {
    improved = false;
    Enumeration elm = elements();
    while(elm.hasMoreElements()) {
      ListNode from = (ListNode) elm.nextElement();
      improved = onePointMove(from,t);
      if(improved) break;
    }
  }
  return improved;
}

public boolean twoPointMove(Tour t) {
  boolean improved = true;
  while(improved) {
    improved = false;
    Enumeration elm = elements();
    while(elm.hasMoreElements()) {
```

```
      ListNode from = (ListNode) elm.nextElement();
      improved = twoPointMove(from,t);
      if(improved) break;
    }
  }
  return improved;
}

/**
 * from must be a node in this tour t is another tour
 * prev -- from      prev -- to
 *    ====
 * tprev -- to       tprev -- from
 */
public boolean twOpt(ListNode from, Tour t) {
  double serviceTime = g.getServiceTime();
  boolean improved = false;
  if (from.equals(depot))
    logger.warn("in global-2opt from should not be a depot!")
;
  ListNode prev = (ListNode) from.getPrevious();
  double d1 = g.distance(prev,from);
  double[] info1 = getRemainingDistance(from);
  Enumeration e = t.elements();
  while (e.hasMoreElements()) {
    ListNode to = (ListNode) e.nextElement();
    ListNode tprev = (ListNode) to.getPrevious();
    double[] info2 = t.getRemainingDistance(to);

    double diffStime = ((int)info1[2]-(int)info2[2])*
 serviceTime;
    int newload1 = load - (int) info1[1] + (int) info2[1];
    int newload2 = t.load - (int) info2[1] + (int) info1[1];
    double d3 = g.distance(prev,to);
    double newDist1 = distance - d1 - info1[0]+info2[0] + d3-
 diffStime;
    double d2 = g.distance(tprev,to);
    double d4 = g.distance(tprev,from);
    double newDist2 = t.getDistance()-d2-info2[0]+info1[0]+d4
 + diffStime;
    if (newload1 <= g.getCapacity() && newload2 <= g.
 getCapacity()
        && newDist1<=g.getDisConstraint() && newDist2<= g.
 getDisConstraint()) {
      double saving = d1 - d3 + d2 - d4;
```

```
        if (saving > 1e-5) {
          improved = true;
          prev.setNext(to); to.setPrevious(prev);
          tprev.setNext(from); from.setPrevious(tprev);
          load = newload1;
          t.load = newload2;
          distance = newDist1;
          t.distance = newDist2;
          int num = (int) info2[2] - (int) info1[2];
          numElements += num;
          t.numElements -= num;
          checkDistance();
          t.checkDistance();
        }
          }
      if (improved)
        break;
    }
    return improved;
  }
  // return a 1x3 double array,
  // the first number is the remaining distance
  // the second number is the remaining demands
  // the third number is the remaining number of nodes(
   including node n)
  protected double[] getRemainingDistance(ListNode n) {
    double value[] = new double[3];
    ListNode next;
    while (n!=null) {
      value[2] += 1; // this store partial number of nodes
      value[1] += ((VRPNode) n.data).getDemand(); // this
      next = (ListNode) n.getNext();
      if(next!=null) {
        value[0] += g.distance((VRPNode) n.data, (VRPNode) next
   .data);
      }
      n = next;
    }
    return value;
  }

  private boolean twOptSingleStep(ListNode from) {
    boolean improved = false;
    Enumeration elm = elements();
    boolean ahead = true;
```

```
    while(elm.hasMoreElements()) {
      ListNode to = (ListNode) elm.nextElement();
      if(to.equals(from)) {
        ahead = false;
        continue;
      }
      if(ahead) improved = check_2opt_move(to,from);
      else improved = check_2opt_move(from,to);
      if(improved) break;
    }
    return improved;
  }


  //if the saving is positive, we reverse the chain from-->to
  private boolean check_2opt_move(ListNode from, ListNode to) {
    boolean improved = false;
    ListNode prev = (ListNode) from.getPrevious();
    ListNode next = (ListNode) to.getNext();
    // prev->from->...->to->next <----------> prev->to->....->
   from->next
    double saving ;
    if(next!=null)
      saving = g.distance((VRPNode) prev.data, (VRPNode) from.
   data)
        + g.distance((VRPNode) to.data, (VRPNode) next.data)
        - g.distance((VRPNode) prev.data, (VRPNode) to.data)
        - g.distance((VRPNode) from.data, (VRPNode) next.data);
    else
      saving = g.distance((VRPNode)prev.data,(VRPNode)from.data
   )
        - g.distance((VRPNode)prev.data,(VRPNode)to.data);
    if (saving > 1e-5) {
      improved = true;
      //logger.info("local-2opt " + from + " to " + to +" with
   saving " + saving);
      reverse(from, to);
      distance -= saving;
    }
    return improved;
  }

  private boolean twOptStep() {
    boolean improved = false;
    ListNode[] list = toArray();
```

```
  for (int i = 0; i < list.length; i++) {
    ListNode from = list[i];
    improved = twOptSingleStep(from);
    if (improved)
      break;
  }
  return improved;
}


public ListNode[] toArray() {
  Enumeration e = elements();
  ListNode[] ret = new ListNode[numElements];
  int index = 0;
  while (e.hasMoreElements())
    ret[index++] = (ListNode) e.nextElement();
  return ret;
}

public ListNode remove(Object node) {
  // TODO Auto-generated method stub
  Enumeration e = elements();
  ListNode ret = null;
  while (e.hasMoreElements()) {
    ListNode lstnode = (ListNode) e.nextElement();
    if (lstnode.getData().equals(node)) {
      ret = lstnode;
      break;
    }
  }
  if (ret != null) {
    ListNode prev = (ListNode) ret.getPrevious();
    ListNode next = (ListNode) ret.getNext();
    prev.setNext(next);
    if(next!=null) next.setPrevious(prev);
    numElements--;
    load -= ((Node) node).getDemand();
    evaluate();
  }
  return ret;
}

protected double evaluate() {
  distance = 0D;
  final  double serviceTime = g.getServiceTime();
```

```
  Enumeration elm = elements();
  int cur = 0;
  while(elm.hasMoreElements()) {
    ListNode next = (ListNode)elm.nextElement();
    distance += g.distance(cur,next.getId()) + serviceTime;
    cur = next.getId();
  }
  return distance;
}
/*
 * (non-Javadoc)
 * return ListNode
 * @see hvrp.List#elements()
 */
public Enumeration elements() {
  return (new Enumeration() {
    ListNode current = (ListNode) depot.getNext();
    public boolean hasMoreElements() {
      return current != null;
    }

    public Object nextElement() {
      if (!hasMoreElements())
    return null;

  ListNode result = current;
  current = (ListNode) current.getNext();
  return result;
    }
  });
}
/*
 * (non-Javadoc)
 *
 * @see hvrp.List#add(java.lang.Object)
 */
public void add(Object o) {
  // TODO Auto-generated method stub
  append(o);
}

public double leastCostInsertion(VRPNode o) {
  double cost;
  //o.setRoute(tourID);
  VRPNode cur = (VRPNode) depot.data;
```

```
  if(isEmpty()) {
    cost = g.distance(cur,o);
    addAfter(depot,o);
  }
  else {
    VRPNode next = (VRPNode) ((ListNode) depot.getNext()).
 data;
    if(next!=null)
      cost = g.distance(cur, o) + g.distance(o, next) - g.
 distance(cur, next);
    else
      cost = g.distance(cur,o);
    ListNode insPlace = depot;
    Enumeration e = elements();
    while (e.hasMoreElements()) {
      ListNode current = (ListNode) e.nextElement();
      cur = (VRPNode) current.data;
      ListNode nextNode = (ListNode) current.getNext();
      if(nextNode!=null) {
        next = (VRPNode) nextNode.data;
        double theCost = g.distance(cur, o)
          + g.distance(o, next)
          - g.distance(cur, next);
        if (theCost < cost) {
          cost = theCost;
          insPlace = current;
        }
      }
      else { // it's the last node in the tour
        double theCost = g.distance(cur,o);
        if(theCost < cost) {
          cost = theCost;
          insPlace = current;
        }
      }
    }
    addAfter(insPlace, o);
  }
  return cost;
}

public double getLeastCost(VRPNode o) {
  VRPNode cur = (VRPNode) depot.data;
  VRPNode next = null;
  if( depot.getNext()!=null) {
```

```
    next = (VRPNode) ((ListNode) depot.getNext()).data;
  }
  double cost;
  if(next!=null)
    cost = g.distance(cur, o) + g.distance(o, next) - g.
 distance(cur, next);
  else
    cost = g.distance(cur,o);
  ListNode insPlace = depot;
  Enumeration e = elements();
  while (e.hasMoreElements()) {
    ListNode current = (ListNode) e.nextElement();
    cur = (VRPNode) current.data;
    ListNode nextNode = (ListNode) current.getNext();
    if(nextNode!=null) {
      next = (VRPNode) nextNode.data;
      double theCost = g.distance(cur, o)
        + g.distance(o, next)
        - g.distance(cur, next);
      if (theCost < cost) {
        cost = theCost;
        insPlace = current;
      }
    }
    else {
      double theCost = g.distance(cur,o);
      if(theCost < cost)
        cost = theCost;
    }
  }
  return cost+g.getServiceTime();
}

public void addAfter(ListNode n, Object obj) {
  ListNode lstnode = new ListNode(obj);
  addAfter(n, lstnode);
}

public void addFirst(Object obj) {
  addAfter(depot, obj);
}

// add n2 after n1
protected void addAfter(ListNode n1, ListNode n2) {
  //n2.setRoute(n1.getRoute());
```

```
  ListNode next = (ListNode) n1.getNext();
  if(next!=null) {
    double cost = g.distance((VRPNode) n1.data, (VRPNode) n2.
 data)
      + g.distance((VRPNode) n2.data, (VRPNode) next.data)
      - g.distance((VRPNode) n1.data, (VRPNode) next.data);
    distance += cost;
    n1.setNext(n2);
    next.setPrevious(n2);
    n2.setPrevious(n1);
    n2.setNext(next);
  }
  else {
    // n1 is the last node in the tour
    distance += g.distance((VRPNode)n1.data,(VRPNode)n2.data)
 ;
    n1.setNext(n2);
    n2.setPrevious(n1);
    depot.setPrevious(n2);
  }
  load += ((VRPNode) n2.data).getDemand();
  // now add the service time
  distance += g.getServiceTime();
  numElements++;
  //logger.warn("you should update the tour length");
}
// add n2 before n1
protected void addBefore(ListNode n1, ListNode n2) {
  addAfter((ListNode) n1.getPrevious(), n2);
}
/*
 * (non-Javadoc)
 *
 * @see hvrp.List#append(java.lang.Object)
 */
public void append(Object o) {
  // TODO Auto-generated method stub
  VRPNode cust = (VRPNode) o;
  ListNode newnode = new ListNode(o);
  if(depot.getPrevious()!=null)
    addAfter((ListNode) depot.getPrevious(), newnode);
  else addAfter(depot,newnode);
}
/*
 * (non-Javadoc)
```

```
 *
 * @see hvrp.List#contains(java.lang.Object)
 */
public boolean contains(Object o) {
  // TODO Auto-generated method stub
  return find(o) != null;
}

/*
 * (non-Javadoc)
 *
 * @see hvrp.List#elementAt(int) depot is not included
 */
public Object elementAt(int index) throws
 NoSuchElementException {
  // TODO Auto-generated method stub
  if (index < 0 || index >= numElements)
    throw new NoSuchElementException();
  Enumeration e = elements();
  ListNode node = null;
  for (int i = 0; i < index; i++) {
    node = (ListNode) e.nextElement();
  }
  return node;

}

public ListNode getNodeByID(int id) {
  ListNode find=null;
  Enumeration elm = elements();
  while(elm.hasMoreElements()) {
    ListNode node = (ListNode) elm.nextElement();
    if(node.getId()==id){
      find = node;
      break;
    }
  }
  return find;
}

public boolean isEmpty() {
  if(numElements==0) {
    //logger.assertLog(depot.getNext()==null, "tour should be
  empty!");
    logger.assertLog(Math.abs(distance)<1e-6,"distance should
```

```
 be 0 in empty tour");
    logger.assertLog(load==0,"load should be 0 in empty tour"
 );
  }
  return numElements == 0;
}

public String toString() {
  if (numElements == 0)
    return "<empty>";

  String s = "Tour("+tourID+")[" + depot + "]" + "<-->";
  Enumeration e = elements();

  while (e.hasMoreElements()) {
    ListNode node = (ListNode) e.nextElement();
    s = s + "[" + node + "]";
    //logger.info(node);
    if (node != null)
      s = s + "<-->";

  }
  //s += "[" + depot + "]";
  double computedDist = checkDistance();
  s +=" total " + numElements + " nodes, total load="+load+",
 total distance="+distance+"\n"+
    "check distance="+computedDist+"\n";
  if(Math.abs(computedDist-distance)>1e-6) System.exit(1);
  return s;
}
/*
 * (non-Javadoc)
 *
 * @see hvrp.List#length()
 */
public int length() {
  // TODO Auto-generated method stub
  return numElements;
}

public int getLoad() {
  return load;
}

public int checkLoad() {
```

```
  int ld = 0;
  Enumeration e = elements();
  while (e.hasMoreElements()) {
    ld
      += ((VRPNode) (((ListNode) e.nextElement()).getData()))
 .getDemand();
  }
  if (ld != this.load || ld>g.getCapacity())
    logger.error("load is not consistent in Tour");
  return ld;
}

private void checkRouteID() {
  Enumeration elm = elements();
  while(elm.hasMoreElements()) {
    ListNode anode = (ListNode) elm.nextElement();
    if(anode.getRoute()!= getTourID()) {
      logger.error("route id not consistent");
    }
  }
}

public double checkDistance() {
  boolean failed = false;
  final double serviceTime = g.getServiceTime();
  checkLoad();
  //checkRouteID();
  if(numElements==0) return 0;
  int totalLoad = 0;
  double computedDistance =0D;
  if(depot.getNext()!=null) computedDistance +=
    g.distance(
        (VRPNode) depot.data,
        (VRPNode) ((ListNode) depot.getNext()).data);
  //if(depot.getNext()!=null) computedDistance += serviceTime
 ;
  Enumeration e = elements();
  while (e.hasMoreElements()) {
    ListNode current = (ListNode) e.nextElement();
    computedDistance += serviceTime;
    totalLoad += ((VRPNode)current.data).getDemand();
    ListNode next = (ListNode) current.getNext();
    if(next!=null)
      computedDistance += g.distance((VRPNode) current.
 getData(), (VRPNode) next.getData());
```

```
    }
    if (Math.abs(computedDistance - distance) > 1e-5) {
      logger.error(
          "distance is not consistent in tour, computed="
          + computedDistance
          + ",record="
          + distance);
      failed = true;
    }
    if(totalLoad != load)
      logger.error("load is not consistent in tour("+getTourID
   ()+"), computed="
          + totalLoad +",record="+load);
    if(distance>g.getDisConstraint()) {
      logger.error("distance constraint is violated!");
      logger.error("computed="+computedDistance+"recorded="+
   distance+",distance constraint="+g.getDisConstraint());
      failed = true;
      //System.exit(1);
    }
    if(failed) System.exit(1);
    return computedDistance;
    //return distance;
  }

  public double checkDistance(String location) {
    final double serviceTime = g.getServiceTime();
    checkLoad();
    if(numElements==0) return 0;
    int totalLoad = 0;
    int checkNum=0;
    double computedDistance =
      g.distance(
          (VRPNode) depot.data,
          (VRPNode) ((ListNode) depot.getNext()).data);
    Enumeration e = elements();
    while (e.hasMoreElements()) {
      checkNum++;
      ListNode current = (ListNode) e.nextElement();
      if(current.getId()==0) {
        logger.error("depot is in the tour");
        System.exit(1);
      }
      totalLoad += ((VRPNode)current.data).getDemand();
      ListNode next = (ListNode) current.getNext();
```

```
    if(next!=null)
      computedDistance += g.distance((VRPNode) current.
 getData(), (VRPNode) next.getData());
    computedDistance += serviceTime;
  }
  if(numElements!=checkNum) {
    logger.error("numElements not consistent!, record="+
 numElements+",checked="+checkNum);
    System.exit(1);
  }
  if (Math.abs(computedDistance - distance) > 1e-5) {
    logger.error("in "+ location+
        " distance is not consistent in tour, computed="
        + computedDistance
        + ",record="
        + distance);
    System.exit(1);
  }
  if(totalLoad != load)
    logger.error("load is not consistent in tour, computed="
        + totalLoad +",record="+load);
  if(distance>g.getDisConstraint()) {
    logger.warn("distance constraint is violated!");
    logger.error("computed distance="+distance+",distance
 constraint="+g.getDisConstraint());
  }
  return computedDistance;
}
public double getDistance() {
  return distance;
}

public void onePointMove() {
  while (OPMStep());
}

// one point move in the same tour
public boolean onePointMove(ListNode from) {
  return onePointMoveSingleStep(from);
}

// one point move in different tour
// from should be in this tour
// if the current tour has only one node, and the move is
 feasible
```

```
// we should do the move immediately even the saving is
 negative
// b/c it reduces the vehicle number by 1
// 4.14/05 we can try to insert before or after a node
public boolean onePointMove(ListNode from, Tour t)
{
  int fd = ((VRPNode) from.data).getDemand();
  if( t.load + fd > g.getCapacity() )
    return false;
  final double serviceTime = g.getServiceTime();
  boolean improved = false;
  ListNode prev = (ListNode) from.getPrevious();
  ListNode next = (ListNode) from.getNext();
  double dold = g.distance((VRPNode) prev.data, (VRPNode)
 from.data);
  double dnew = 0;
  if(next!=null) {
    dold += g.distance((VRPNode) from.data, (VRPNode) next.
 data);
    dnew = g.distance((VRPNode) prev.data, (VRPNode) next.
 data);
  }
  ListNode firstNode = (ListNode) t.getHead().getNext();
  double newDist1 = distance - dold + dnew - serviceTime;
  double delta_t = g.distance(t.depot,firstNode) -
           g.distance(t.depot,from) - g.distance(from,
 firstNode);
  double newDist2 = t.distance - delta_t + serviceTime;
  double saving = dold - dnew + delta_t;
  if(newDist1 <= g.getDisConstraint() && newDist2 <= g.
 getDisConstraint()) {
    if(saving > 1e-6) {
      prev.setNext(next);
      if(next!=null) next.setPrevious(prev);
      t.depot.setNext(from);
      from.setPrevious(t.depot);
      from.setNext(firstNode);
      firstNode.setPrevious(from);
      distance = newDist1;
      t.distance = newDist2;
      load -= fd;
      t.load += fd;
      numElements--;
      t.numElements++;
      checkDistance();
```

```
      t.checkDistance();
      return true;
    }
  }
  Enumeration e = t.elements();
  while (e.hasMoreElements()) {
    boolean singleTour = false;
    ListNode to = (ListNode) e.nextElement();
    ListNode tonext = (ListNode) to.getNext();
    ListNode toprev = (ListNode) to.getPrevious();

    // TODO add distance constraint or other side constraint
    if (true) {
      double d1 = tonext==null?0:g.distance((VRPNode) to.data
, (VRPNode) tonext.data);
      double distold = dold + d1;
      double d2 =
        g.distance((VRPNode) to.data, (VRPNode) from.data);
      d2 += tonext==null?0:g.distance((VRPNode) from.data, (
VRPNode) tonext.data);
      double distnew = dnew + d2 ;
      saving = distold - distnew;
      newDist1 = distance-dold + dnew-serviceTime;
      newDist2 = t.distance-d1+d2+serviceTime;
      if(newDist1<=g.getDisConstraint() && newDist2<=g.
getDisConstraint()){
        if(numElements==1)  //we are the only node here, do
the move immediately
          singleTour = false;//true;
        if (singleTour || saving > 1e-5) {
          improved = true;
          //logger.info("one point move between routes "+
from +"-->"+to+" get saving "+ saving);
          prev.setNext(next);
          if(next!=null) next.setPrevious(prev);
          to.setNext(from);
          from.setPrevious(to);
          if(tonext!=null) tonext.setPrevious(from);
          from.setNext(tonext);
          // set up correct load and distance for each
vehicle
          //vehicle.setLoad(vehicle.getLoad()-fd);
          load -= fd;
          //t.getVehicle().setLoad(t.getVehicle().getLoad()+
fd);
```

```
            t.load += fd;
            distance = newDist1;
            t.distance = newDist2;
            numElements--;
            t.numElements++;
            if(singleTour) {
              if(debug)
                logger.info("reduce one tour "+ tourID+ " in
 one-point-move with saving="+saving);
              logger.assertLog(load==0,"the load should be zero
 ");
              logger.assertLog(Math.abs(distance)<1e-5,"the
 distance should be zero!");
            }
            //if(saving<0) t.clean();
          }
        }
      }
      if(improved) break;
    }// end while
    return improved;
  }

  private boolean onePointMoveSingleStep(ListNode from) {
    boolean improved = false;
    ListNode prev = (ListNode) from.getPrevious();
    ListNode next = (ListNode) from.getNext();
    double  dold = g.distance((VRPNode) prev.data, (VRPNode)
 from.data);
    double dnew = 0D;
    double distold=0D, distnew=0D;
    double saving=0D;
    if(next!=null) {
      dold +=g.distance((VRPNode) from.data, (VRPNode) next.
 data);
      dnew = g.distance((VRPNode) prev.data, (VRPNode) next.
 data);
    }
    ListNode firstNode = (ListNode) depot.getNext();
    if(from!=firstNode) { // try to insert from between depot
 and firstNode
      distold = dold + g.distance(depot,firstNode);
      distnew = dnew + g.distance(depot,from) + g.distance(from
 ,firstNode);
      saving = distold - distnew;
```

```
    if(saving > 1e-6) {
      depot.setNext(from);
      from.setPrevious(depot);
      from.setNext(firstNode);
      firstNode.setPrevious(from);
      prev.setNext(next);
      if(next!=null) next.setPrevious(prev);
      distance -= saving;
      checkDistance();
      return true;
    }
  }
  Enumeration elm = elements();
  while(elm.hasMoreElements()) {
    ListNode to = (ListNode) elm.nextElement();
    if(to.equals(prev) || to.equals(from)) continue;
    ListNode tonext = (ListNode) to.getNext();
    // prev->from->...->to->tonext <----------> prev->to
->....->from->tonext
    //double distold,distnew;
    if(tonext!=null) {
      distold = dold+g.distance((VRPNode) to.data, (VRPNode)
tonext.data);
      distnew = dnew + g.distance((VRPNode) to.data, (VRPNode
) from.data)
        + g.distance((VRPNode) from.data, (VRPNode) tonext.
data);
    }
    else {
      distold = dold;
      distnew = dnew + g.distance((VRPNode)to.data,(VRPNode)
from.data);
    }
    saving = distold - distnew;
    if (saving > 1e-5) {
      improved = true;
      //logger.info("onePointMove[within tour]" + from +
"-->" + to +" with saving " + saving);
      prev.setNext(next);
      if(next!=null) next.setPrevious(prev);
      to.setNext(from);
      from.setPrevious(to);
      from.setNext(tonext);
      if(tonext!=null) tonext.setPrevious(from);
      distance -= saving;
```

```
        checkDistance();
        break;
      } else {
        to = tonext;
      }
      if (improved)
        break;
    }
    return improved;
  }

  private boolean OPMStep() {
    boolean improved = false;
    ListNode[] list = toArray();
    for (int i = 0; i < list.length; i++) {
      ListNode from = list[i];
      improved = onePointMoveSingleStep(from);
      if (improved)
        break;
    }
    return improved;

  }

  /*
   * (non-Javadoc)
   *
   * @see java.lang.Object#clone()
   */
  public Object clone(){
    // TODO Auto-generated method stub
    Tour copy = null;
    try {
      copy = (Tour) super.clone();
    } catch(CloneNotSupportedException e){}
    copy.depot = (ListNode) depot.clone();
    ListNode current = copy.depot;
    Enumeration e = elements();
    while (e.hasMoreElements()) {
      ListNode next = (ListNode) ((ListNode) e.nextElement()).
   clone();
      current.setNext(next);
      next.setPrevious(current);
      current = next;
    }
```

```
    current.setNext(null);
    return copy;
  }

  /**
   * @return Returns the depot.
   */
  public VRPNode getDepot() {
    return (VRPNode) depot.data;
  }

  public ListNode getHead() {
    return depot;
  }

  public void twoPointMove() {
    while (twoPointMoveStep());
  }

  private boolean twoPointMoveStep() {
    boolean improved = false;
    ListNode[] list = toArray();
    for (int i = 0; i < list.length; i++) {
      ListNode from = list[i];
      improved = twoPointMoveSingleStep(from);
      if (improved)
        break;
    }
    return improved;
  }

  private boolean twoPointMoveSingleStep(ListNode from) {
    boolean improved = false;
    ListNode fprev = (ListNode) from.getPrevious();
    ListNode fnext = (ListNode) from.getNext();
    double dold = g.distance((VRPNode) fprev.data, (VRPNode)
   from.data);
    if(fnext!=null)
      dold += g.distance((VRPNode) from.data, (VRPNode) fnext.
   data);
    Enumeration elm = elements();
    //ListNode to = (ListNode) fnext.getNext();
    while(elm.hasMoreElements()) {
      ListNode to = (ListNode) elm.nextElement();
      if(to.equals(fprev) || to.equals(from) || to.equals(fnext
```

```
)) continue;
    ListNode tprev = (ListNode) to.getPrevious();
    ListNode tnext = (ListNode) to.getNext();
    double distOld = dold + g.distance((VRPNode) tprev.data,
(VRPNode) to.data);
    if(tnext!=null) distOld+= g.distance((VRPNode) to.data, (
VRPNode) tnext.data);
    double dnew = g.distance((VRPNode) tprev.data, (VRPNode)
from.data)
      + g.distance((VRPNode) fprev.data, (VRPNode) to.data);
    if(fnext!=null)
      dnew += g.distance((VRPNode) to.data, (VRPNode) fnext.
data);
    if(tnext!=null)
      dnew += g.distance((VRPNode) from.data, (VRPNode) tnext
.data);
    double saving = distOld - dnew;
    if (saving > 1e-5) {
      //logger.info("twoPointMove[within tour]"+from+"-->" +
to + ",saving = " +saving);
      improved = true;
      fprev.setNext(to);
      to.setPrevious(fprev);
      to.setNext(fnext);
      if(fnext!=null) fnext.setPrevious(to);
      tprev.setNext(from);
      from.setPrevious(tprev);
      from.setNext(tnext);
      if(tnext!=null) tnext.setPrevious(from);
      distance -= saving;
      break;
    }
  }
  return improved;
  //return false;
}

public boolean twoPointMove(ListNode n) {
  return twoPointMoveSingleStep(n);
}

public boolean twoPointMove(ListNode from, Tour t) {
  boolean improved = false;
  if (from.equals(depot))
    logger.warn("in global-twoPointMove from should not be a
```

```
depot!");
 ListNode fprev = (ListNode) from.getPrevious();
 ListNode fnext = (ListNode) from.getNext();
 double dold = g.distance((VRPNode) fprev.data, (VRPNode)
from.data);
 if(fnext!=null) dold += g.distance((VRPNode) from.data, (
VRPNode) fnext.data);

 Enumeration e = t.elements();
 while (e.hasMoreElements()) {
   ListNode to = (ListNode) e.nextElement();
   ListNode tprev = (ListNode) to.getPrevious();
   ListNode tnext = (ListNode) to.getNext();

   int fd = ((VRPNode) from.data).getDemand();
   int td = ((VRPNode) to.data).getDemand();
   int diff = fd - td;
   int newload1 = load - diff;
   int newload2 = t.load + diff;
   // TODO add distance constraint or other side constraint
   if (newload1 <= g.getCapacity()
       && newload2 <= g.getCapacity()) {
     double d1 = g.distance((VRPNode) tprev.data, (VRPNode)
to.data);
     if(tnext!=null) d1 += g.distance((VRPNode) to.data, (
VRPNode) tnext.data);
     double distOld = dold + d1;
     double d2 = g.distance((VRPNode) tprev.data, (VRPNode)
from.data);
     if(tnext!=null) d2 += g.distance((VRPNode) from.data, (
VRPNode) tnext.data);
     double d3 = g.distance((VRPNode) fprev.data, (VRPNode)
to.data);
     if(fnext!=null) d3 += g.distance((VRPNode) to.data, (
VRPNode) fnext.data);
     double dnew = d2 + d3;
     double saving = distOld - dnew;
     double newDist1 = distance - dold + d3;
     double newDist2 = t.distance -d1 + d2;
     if(newDist1<=g.getDisConstraint() && newDist2<=g.
getDisConstraint()) {
       if (saving > 1e-5) {
         improved = true;
         //logger.info("two point move between routes "+
from +"-->"+to+",saving ="+ saving);
```

```
            fprev.setNext(to);
            to.setPrevious(fprev);
            to.setNext(fnext);
            if(fnext!=null) fnext.setPrevious(to);
            tprev.setNext(from);
            from.setPrevious(tprev);
            from.setNext(tnext);
            if(tnext!=null) tnext.setPrevious(from);
            // set up correct load and distance for each
 vehicle
            load = newload1;
            t.load = newload2;
            distance = newDist1;
            t.distance = newDist2;
            break;
          }
        }
          }
      if (improved)
        break;
      //else to=tnext;
    }
    return improved;
}
// Record-to-Record travel part
// check feasibility
public RTRInfo get2OptRTRInfo(ListNode from, Record r) {
    double bestSaving = -1e10;
    ListNode bestNode = null;
    Enumeration elm = elements();
    boolean direction = true;
    while (elm.hasMoreElements()) {
      ListNode to = (ListNode) elm.nextElement();
      double saving;
      if(to.equals(from)) {
        direction = false;
        continue;
      }
      if(direction) {
        //prev->to->....from->next ----------> prev->from...to->
 next
        ListNode prev = (ListNode) to.getPrevious();
        ListNode next = (ListNode) from.getNext();
        saving = g.distance((VRPNode) prev.data, (VRPNode) to.
 data)
```

```
        - g.distance((VRPNode) prev.data, (VRPNode) from.data
);
      if(next!=null)
        saving += g.distance((VRPNode) from.data, (VRPNode)
next.data)
          - g.distance((VRPNode) to.data, (VRPNode) next.data
);
    }
    else {
      ListNode prev = (ListNode) from.getPrevious();
      ListNode next = (ListNode) to.getNext();
      // prev->from->...->to->next <----------> prev->to
->....->from->next
      saving = g.distance((VRPNode) prev.data, (VRPNode) from
.data)
        - g.distance((VRPNode) prev.data, (VRPNode) to.data);
      if(next!=null)
        saving += g.distance((VRPNode) to.data, (VRPNode)
next.data)
          - g.distance((VRPNode) from.data, (VRPNode) next.
data);
    }
    if(distance-saving>g.getDisConstraint()) continue;
    else {
      if (r.accept(saving)) {
        if (saving > bestSaving) {
          bestSaving = saving;
          bestNode = to;
        }
      }
    }
  }
  return new RTRInfo(bestNode, this, bestSaving);
}

public RTRInfo get2OptRTRInfo(ListNode from, Tour t, Record r
) {
  double serviceTime = g.getServiceTime();
  if (from.equals(depot))
    logger.warn(
        "get2optinfo in global-2opt from should not be a
depot!");
  ListNode next = (ListNode) from.getNext();
  double d1 = next==null?0:g.distance((VRPNode) from.data, (
VRPNode) next.data);
```

```
  double[] info1 = getRemainingDistance(next);
  double best_saving = -1e10;
  ListNode bestNode = null;
  Enumeration e = t.elements();
  while (e.hasMoreElements()) {
    ListNode to = (ListNode) e.nextElement();
    ListNode tonext = (ListNode) to.getNext();
    double[] info2 = t.getRemainingDistance(tonext);
    double diffStime = ((int)info1[2]-(int)info2[2])*
 serviceTime;
    int newload1 = load - (int) info1[1] + (int) info2[1];
    int newload2 = t.load - (int) info2[1] + (int) info1[1];
    // TODO add distance constraint or other side constraint
    if (newload1 <= g.getCapacity() && newload2 <= g.
 getCapacity()) {
      double d2 = tonext==null?0:g.distance((VRPNode) to.data
 , (VRPNode) tonext.data);
      double d3 = tonext==null?0:g.distance((VRPNode) from.
 data, (VRPNode) tonext.data);
      double d4 = next==null?0:g.distance((VRPNode) to.data,
 (VRPNode) next.data);
      double newDist1 = distance-info1[0]-d1+d3+info2[0]-
 diffStime;
      double newDist2 = t.distance-info2[0]-d2+d4+info1[0]+
 diffStime;
      if(newDist1<=g.getDisConstraint() && newDist2<=g.
 getDisConstraint()) {
        double saving = (d1 - d3) + (d2 - d4);
        if (r.accept(saving)) {
          if (saving > best_saving) {
            best_saving = saving;
            bestNode = to;
          }
        }
      }
    }
  }
  return new RTRInfo(bestNode, t, best_saving);
}

protected boolean isAhead(ListNode from, ListNode to) {
  boolean ret =false;
  Enumeration elm = elements();
  while(elm.hasMoreElements()) {
    ListNode current = (ListNode) elm.nextElement();
```

```
    if(current.equals(from)) {
      ret = true;
      break;
    }
    else if(current.equals(to)) {
      ret = false;
      break;
    }
  }
  return ret;
}

public void do2OptRTR(ListNode from, RTRInfo info) {
  //logger.assertLog(from!=null,"in do2OptRTR from is null");
  ListNode to = info.getNode();
  //logger.assertLog(to!=null,"in do2OptRTR to is null");
  Tour t = info.getTour();
  if (this == t) {
    if(isAhead(from,to)) {
      ListNode prev = (ListNode) from.getPrevious();
      ListNode next = (ListNode) to.getNext();
      double saving = g.distance((VRPNode) prev.data, (
VRPNode) from.data)
        - g.distance((VRPNode) prev.data, (VRPNode) to.data);
      if(next!=null)
        saving += g.distance((VRPNode) to.data, (VRPNode)
next.data)
          - g.distance((VRPNode) from.data, (VRPNode) next.
data);
      if (Math.abs(saving - info.getSaving()) > 1e-10) {
        logger.error("2opt RTR saving not consistent");
        System.exit(1);
      }
      reverse(from, to);
      distance -= saving;
    }
    else { // to is ahead of from
      ListNode prev = (ListNode) to.getPrevious();
      ListNode next = (ListNode) from.getNext();
      double saving = g.distance((VRPNode) prev.data, (
VRPNode) to.data)
        - g.distance((VRPNode) prev.data, (VRPNode) from.data
);
      if(next!=null)
        saving += g.distance((VRPNode) from.data, (VRPNode)
```

```
next.data)
          - g.distance((VRPNode) to.data, (VRPNode) next.data
);
      if (Math.abs(saving - info.getSaving()) > 1e-10) {
        logger.error("2opt RTR saving not consistent");
        System.exit(1);
      }
      reverse(to,from);
      distance -= saving;
    }

  } else {
    if (to == null || t == null)
      logger.error("2opt RTR info null");
    ListNode next = (ListNode) from.getNext();
    double d1 = next==null?0:g.distance((VRPNode) from.data,
(VRPNode) next.data);
    double[] info1 = getRemainingDistance(next);
    ListNode tonext = (ListNode) to.getNext();
    // TODO : here can be optimized by directly computing
info
    double[] info2 = t.getRemainingDistance(tonext);
    double diffStime = ((int)info1[2]-(int)info2[2])*g.
getServiceTime();
    int newload1 = load - (int) info1[1] + (int) info2[1];
    int newload2 = t.load - (int) info2[1] + (int) info1[1];
    // TODO add distance constraint or other side constraint

    if (newload1 <= g.getCapacity() && newload2 <= g.
getCapacity()) {
      double d2 = tonext==null?0:g.distance((VRPNode) to.data
, (VRPNode) tonext.data);
      double d3 = tonext==null?0:g.distance((VRPNode) from.
data, (VRPNode) tonext.data);
      double d4 = next==null?0:g.distance((VRPNode) to.data,
(VRPNode) next.data);

      double newDist1 = distance - d1 - info1[0]+info2[0] +
d3-diffStime;
      double newDist2 = t.getDistance()-d2-info2[0]+info1[0]+
d4 + diffStime;
      if(newDist1>g.getDisConstraint() ||newDist2>g.
getDisConstraint()) {
        logger.info("infeasible 2opt RTR!");
        System.exit(1);
```

```
      }
      double saving = (d1 - d3) +  (d2 - d4) ;
      if (Math.abs(saving - info.getSaving()) > 1e-10) {
        logger.error("2opt RTR saving is wrong");
        System.exit(1);
      }
      //logger.info("2opt between routes "+ from +"-->"+to+"
 get
      // saving "+ saving);
      if (this != t) {
        from.setNext(tonext);
        if(tonext!=null) tonext.setPrevious(from);
        to.setNext(next);
        if(next!=null) next.setPrevious(to);
        //updateRoute(tonext);
        //t.updateRoute(next);
        // set up correct load and distance for each vehicle
        load = newload1; t.load = newload2;
        distance = newDist1;
        t.distance = newDist2;
        int num = (int) info2[2] - (int) info1[2];
        numElements += num;
        t.numElements -= num;
      }
    } else {
      logger.error("Fatal error in 2opt-RTR");
      System.exit(1);
    }
  }
}

private void updateRoute(ListNode head) {
  ListNode itr = head;
  while(itr!=null) {
    itr.setRoute(tourID);
    itr = (ListNode) itr.getNext();
  }
}

public RTRInfo getOnePointMoveRTRInfo(ListNode from, Record r
 ) {
  ListNode prev = (ListNode) from.getPrevious();
  ListNode next = (ListNode) from.getNext();
  double dold =
    g.distance((VRPNode) prev.data, (VRPNode) from.data);
```

```
  if(next!=null) dold += g.distance((VRPNode) from.data, (
 VRPNode) next.data);
  double dnew =next==null?0:g.distance((VRPNode) prev.data, (
 VRPNode) next.data);
  double bestSaving = -1e10;
  ListNode bestNode = null;
  Enumeration elm = elements();
  while(elm.hasMoreElements()) {
    ListNode to = (ListNode) elm.nextElement();
    ListNode tonext = (ListNode) to.getNext();
    if(to.equals(from)|| from.equals(tonext)) continue;
    double distold= dold ;
    if(tonext!=null) distold = dold + g.distance((VRPNode) to
 .data, (VRPNode) tonext.data);
    double distnew = dnew + g.distance((VRPNode) to.data, (
 VRPNode) from.data);
    if(tonext!=null) distnew += g.distance((VRPNode) from.
 data, (VRPNode) tonext.data);
    double saving = distold - distnew;
    if(distance - saving<=g.getDisConstraint()){
      if (r.accept(saving)) {
        if (saving > bestSaving) {
          bestSaving = saving;
          bestNode = to;
        }
      }
    }
  }
  return new RTRInfo(bestNode, this, bestSaving);
}

public RTRInfo getOnePointMoveRTRInfo(ListNode from, Tour t,
 Record r) {
  ListNode prev = (ListNode) from.getPrevious();
  ListNode next = (ListNode) from.getNext();

  int fd = ((VRPNode) from.data).getDemand();
  double dold = g.distance((VRPNode) prev.data, (VRPNode)
 from.data);
  if(next!=null) dold += g.distance((VRPNode) from.data, (
 VRPNode) next.data);
  double dnew =next==null?0:g.distance((VRPNode) prev.data, (
 VRPNode) next.data);

  double bestSaving = -1e10;
```

```
  ListNode bestNode = null;
  if (t.load + fd <= g.getCapacity()) {
    Enumeration e = t.elements();
    while (e.hasMoreElements()) {
      ListNode to = (ListNode) e.nextElement();
      ListNode tonext = (ListNode) to.getNext();
      double d1 = tonext==null?0:g.distance((VRPNode) to.data
 , (VRPNode) tonext.data);
      double distold = dold + d1;
      double d2 = g.distance((VRPNode) to.data, (VRPNode)
 from.data);
      d2 += tonext==null?0:g.distance((VRPNode) from.data, (
VRPNode) tonext.data);
      double distnew = dnew + d2 ;
      double saving = distold - distnew;
      double newDist1 = distance -dold + dnew-g.
 getServiceTime();
      double newDist2 = t.distance + d2 - d1 + g.
 getServiceTime();
      if(newDist1<=g.getDisConstraint() && newDist2<=g.
 getDisConstraint()) {
        if (r.accept(saving)) {
          if (saving > bestSaving) {
            bestSaving = saving;
            bestNode = to;
          }
        }
      }
    }
  }
  return new RTRInfo(bestNode, t, bestSaving);
}

public void doOnePointMoveRTR(ListNode from, RTRInfo info) {
  ListNode to = info.getNode();
  Tour t = info.getTour();
  ListNode prev = (ListNode) from.getPrevious();
  ListNode next = (ListNode) from.getNext();
  double dold = g.distance((VRPNode) prev.data, (VRPNode)
 from.data);
  if(next!=null) dold += g.distance((VRPNode) from.data, (
VRPNode) next.data);
  double dnew =next==null?0: g.distance((VRPNode) prev.data,
 (VRPNode) next.data);
```

```
 ListNode tonext = (ListNode) to.getNext();
 int fd = ((VRPNode) from.data).getDemand();
 double d1 = tonext==null?0:g.distance((VRPNode) to.data, (
VRPNode) tonext.data);
 double distold = dold + d1 ;
 double d2 = g.distance((VRPNode) to.data, (VRPNode) from.
data);
 d2 += tonext==null?0:g.distance((VRPNode) from.data, (
VRPNode) tonext.data);
 double distnew = dnew + d2 ;
 double saving = distold - distnew;
 if (Math.abs(saving - info.getSaving()) > 1e-10) {
   logger.error("saving is not consistent in one point move
RTR");
   System.exit(1);
 } else {
   //logger.info("one point move between routes "+ from
   // +"-->"+to+" get saving "+ saving);
   if (this==t) {
     if(distance-saving>g.getDisConstraint())
       logger.error("infeasible solution onepoint move RTR")
;
     prev.setNext(next);
     if(next!=null) next.setPrevious(prev);
     to.setNext(from);
     from.setPrevious(to);
     if(tonext!=null) tonext.setPrevious(from);
     from.setNext(tonext);
     distance -= saving;
   }
   else { // one point move between routes
     double newDist1 = distance-dold+dnew-g.getServiceTime()
;
     double newDist2 = t.distance-d1+d2+g.getServiceTime();
     if(newDist1<=g.getDisConstraint()&& newDist2<=g.
getDisConstraint() && t.load+fd<=g.getCapacity()) {
       prev.setNext(next);
       if(next!=null) next.setPrevious(prev);
       to.setNext(from);
       from.setPrevious(to);
       if(tonext!=null) tonext.setPrevious(from);
       from.setNext(tonext);
       //from.setRoute(t.getTourID());
       load -= fd;
       t.load += fd;
```

```
        distance = newDist1;
        t.distance = newDist2;
        numElements--;
        t.numElements++;
      }
      //else logger.error("infeasible one-point-RTR between
 routes!");
    }
  }
}

public RTRInfo getTwoPointMoveRTRInfo(ListNode from, Tour t,
 Record r) {
  if (this == t)
    return getTwoPointMoveRTRInfoSameTour(from, r);
  else
    return getTwoPointMoveRTRInfoDiffTour(from, t, r);
}

private RTRInfo getTwoPointMoveRTRInfoDiffTour( ListNode from
 , Tour t, Record r) {
  ListNode fprev = (ListNode) from.getPrevious();
  ListNode fnext = (ListNode) from.getNext();
  double dold = g.distance((VRPNode) fprev.data, (VRPNode)
 from.data);
  if(fnext!=null) dold += g.distance((VRPNode) from.data, (
 VRPNode) fnext.data);
  double bestSaving = -1e10;
  ListNode bestNode = null;

  Enumeration e = t.elements();
  while (e.hasMoreElements()) {
    ListNode to = (ListNode) e.nextElement();
    ListNode tprev = (ListNode) to.getPrevious();
    ListNode tnext = (ListNode) to.getNext();

    int fd = ((VRPNode) from.data).getDemand();
    int td = ((VRPNode) to.data).getDemand();
    int diff = fd - td;
    int newload1 = load - diff;
    int newload2 = t.load + diff;
    // TODO add distance constraint or other side constraint
    if (newload1 <= g.getCapacity() && newload2 <= g.
 getCapacity()) {
      double newdist1,newdist2;
```

```
      newdist1 = this.distance - dold;
      double d1 =
        g.distance((VRPNode) tprev.data, (VRPNode) to.data);
      if(tnext!=null) d1 += g.distance((VRPNode) to.data, (
 VRPNode) tnext.data);
      newdist2 = t.distance - d1;
      double distOld = dold + d1;
      double d2 = g.distance((VRPNode) tprev.data, (VRPNode)
 from.data);
      if(tnext!=null) d2 += g.distance((VRPNode) from.data, (
 VRPNode) tnext.data);
      newdist2 += d2;
      double d3 = g.distance((VRPNode) fprev.data, (VRPNode)
 to.data);
      if(fnext!=null) d3 += g.distance((VRPNode) to.data, (
 VRPNode) fnext.data);
      newdist1 += d3;
      double dnew = d2 + d3;
      if(newdist1<=g.getDisConstraint() && newdist2<=g.
 getDisConstraint()) {
        double saving = distOld - dnew;
        if (r.accept(saving)) {
          if (saving > bestSaving) {
            bestSaving = saving;
            bestNode = to;
          }
        }
      }
    }
  }
  return new RTRInfo(bestNode, t, bestSaving);
}

private RTRInfo getTwoPointMoveRTRInfoSameTour(ListNode from,
  Record r) {
  ListNode fprev = (ListNode) from.getPrevious();
  ListNode fnext = (ListNode) from.getNext();
  double dold =
    g.distance((VRPNode) fprev.data, (VRPNode) from.data);
  if(fnext!=null)
    dold += g.distance((VRPNode) from.data, (VRPNode) fnext.
 data);
  double bestSaving = -1e10;
  ListNode bestNode = null;
  Enumeration elm = elements();
```

```
  while (elm.hasMoreElements()) {
    ListNode to = (ListNode) elm.nextElement();
    if(to.equals(from) || to.equals(fprev) || to.equals(fnext
 )) continue;
    ListNode tprev = (ListNode) to.getPrevious();
    ListNode tnext = (ListNode) to.getNext();
    double distOld = dold
      + g.distance((VRPNode) tprev.data, (VRPNode) to.data);
    if(tnext!=null) distOld += g.distance((VRPNode) to.data,
 (VRPNode) tnext.data);
    double dnew =
      g.distance((VRPNode) tprev.data, (VRPNode) from.data)
      + g.distance((VRPNode) fprev.data, (VRPNode) to.data);
    if(tnext!=null) dnew += g.distance((VRPNode) from.data, (
 VRPNode) tnext.data);
    if(fnext!=null) dnew += g.distance((VRPNode) to.data, (
 VRPNode) fnext.data);
    double saving = distOld - dnew;
    if(distance-saving>g.getDisConstraint()) continue;
    else {
      if (r.accept(saving)) {
        //logger.info("same tour twoPointMove "+from+"-->" +
 to + "
        // saving = " +saving);
        if (saving > bestSaving) {
          bestSaving = saving;
          bestNode = to;
        }
      }
    }
  }
  return new RTRInfo(bestNode, this, bestSaving);
}

public void doTwoPointMoveRTR(ListNode from, RTRInfo info) {
  Tour t = info.getTour();
  ListNode fprev = (ListNode) from.getPrevious();
  ListNode fnext = (ListNode) from.getNext();
  double dold = g.distance((VRPNode) fprev.data, (VRPNode)
 from.data);
  if(fnext!=null) dold += g.distance((VRPNode) from.data, (
 VRPNode) fnext.data);
  ListNode to = info.getNode();
  ListNode tprev = (ListNode) to.getPrevious();
  ListNode tnext = (ListNode) to.getNext();
```

```
 int fd = ((VRPNode) from.data).getDemand();
 int td = ((VRPNode) to.data).getDemand();
 int diff = fd - td;
 int newload1 = load - diff;
 int newload2 = t.load + diff;
 // TODO add distance constraint or other side constraint
 double d1 = g.distance((VRPNode) tprev.data, (VRPNode) to.
data);
 if(tnext!=null) d1 += g.distance((VRPNode) to.data, (
VRPNode) tnext.data);
 double distOld = dold + d1;
 double d2 = g.distance((VRPNode) tprev.data, (VRPNode) from
.data);
 if(tnext!=null) d2 += g.distance((VRPNode) from.data, (
VRPNode) tnext.data);
 double d3 = g.distance((VRPNode) fprev.data, (VRPNode) to.
data);
 if(fnext!=null) d3 += g.distance((VRPNode) to.data, (
VRPNode) fnext.data);
 double dnew = d2 + d3;
 double saving = distOld - dnew;
 if (Math.abs(saving - info.getSaving()) > 1e-10) {
   logger.error("twopointmove RTR saving is not consisent");
   System.exit(1);
 } else {
   //logger.info("two point move between routes "+ from
+"-->"+to+"
   // get saving "+ saving);
   fprev.setNext(to);
   to.setPrevious(fprev);
   to.setNext(fnext);
   if(fnext!=null) fnext.setPrevious(to);
   tprev.setNext(from);
   from.setPrevious(tprev);
   from.setNext(tnext);
   //from.setRoute(t.getTourID());
   //to.setRoute(this.getTourID());
   if(tnext!=null) tnext.setPrevious(from);
   // set up correct load and distance for each vehicle
   if (t != this) {
     load = newload1;
     t.load = newload2;
     distance = distance - dold + d3;
     t.distance = t.distance - d1 + d2;
```

```
    } else {
      distance = distance - dold - d1 + d2 + d3;
    }
  }
}

public Node getFirst() {
  return (Node) ((ListNode) depot.getNext()).getData();
}

public Node getLast() {
  return (Node) ((ListNode) depot.getPrevious()).getData();
}

public ListNode getLastNode() {
  ListNode last = null;
  Enumeration elm = elements();
  while(elm.hasMoreElements()) {
    last = (ListNode) elm.nextElement();
  }
  return last;
}

public ListNode getFirstNode() {
  return (ListNode) depot.getNext();
}

public double getChainDistance() {
  return distance - g.distance(depot,(ListNode)depot.getNext
 ());
}

public void reloadHead() {
  ListNode first = (ListNode) depot.getNext();
  if(first==null) return;
  ListNode itr = (ListNode) first.getNext();
  ListNode last = null;
  while(itr!=null) {
    last = itr;
    itr = (ListNode) itr.getNext();
  }
  if(last!=null){
    if( !last.equals(first)) {
      double d1 = g.distance((VRPNode)depot.getData(),(
 VRPNode)first.getData());
```

```
      double d2 = g.distance((VRPNode)depot.getData(),(
 VRPNode)last.getData());
      if(d2<d1) {
        reverse(first,last);
        distance -= (d1-d2);
      }
    }
  }
}
/**
 * Do OR-opt for this node n only within this vehicle
 *
 * @param n
 * @return
 */
public void orOptMove() {
  while(orOptMoveStep()) ;

}

private boolean orOptMoveStep() {
  logger.assertLog(!isEmpty(),"in or-opt the tour is emtpy");
  boolean improved = false;
  ListNode[] list = toArray();
  for(int i=0; i<list.length;i++) {
    ListNode from = list[i];
    improved = orOptSingleStep(from);
    if(improved) break;
  }
  return improved;
}

private boolean orOptSingleStep(ListNode n) {
  boolean improved = false;
  //Node current = (Node) n.getData();
  //logger.assertLog(length()>2,"in OROptMove less than 3
 nodes");
  if(n.getNext()==null) return false; //we need at least two
 nodes to do or-opt
  else {
    ListNode prev = (ListNode) n.getPrevious();
    ListNode n1 = (ListNode)n.getNext();
    ListNode next = (ListNode)n1.getNext();
    double d1 = g.distance(prev,n);
    if(next!=null) d1+=g.distance(n1,next)-g.distance(prev,
```

```
 next);
    Enumeration e = this.elements();
    while(e.hasMoreElements()) {
      ListNode itr = (ListNode)e.nextElement();
      if(itr.equals(n) || itr.equals(n1) || itr.equals(prev)
 ) continue;
      else {
        logger.assertLog(!itr.equals(depot),"depot should not
 be in the OROpt");
        ListNode itrNext = (ListNode)itr.getNext();
        double saving = d1-g.distance(itr,n);
        if(itrNext!=null) saving +=g.distance(itr,itrNext)-g.
 distance(n1,itrNext);
        if(saving>1e-5) {
          improved = true;
          prev.setNext(next);
          if(next!=null) next.setPrevious(prev);
          itr.setNext(n);
          n.setPrevious(itr);
          n1.setNext(itrNext);
          if(itrNext!=null) itrNext.setPrevious(n1);
          distance -= saving;
          //logger.info("find a OROpt Move with saving="+
 saving);
          break;
        }
      }
    }
  }
  checkDistance();
  return improved;
}

public boolean orOptMove(Tour t) {
  boolean improved = true;
  while(improved) {
    improved = false;
    Enumeration elm = elements();
    while(elm.hasMoreElements()) {
      ListNode from = (ListNode) elm.nextElement();
      improved = orOptMove(from,t);
      if(improved) break;
    }
  }
  return improved;
```

```
}
// OR opt in tour t , trying to insert n and n1 into t
public boolean orOptMove(ListNode n, Tour t) {
  final double serviceTime = g.getServiceTime();
  boolean improved = false;
  Node current = (Node) n.getData();
  //logger.assertLog(length()>2,"in OROptMove less than 3
 nodes");
  if(n.getNext()==null) return false;
  else {
    ListNode prev = (ListNode) n.getPrevious();
    ListNode n1 = (ListNode)n.getNext();
    ListNode next = (ListNode)n1.getNext();
    int transferedLoad = current.getDemand() + ((Node)n1.data
 ).getDemand();
    if(t.getLoad() + transferedLoad>g.getCapacity()) return
 false;
    double d1 = g.distance(prev,n);
    if(next!=null) d1+= g.distance(n1,next)-g.distance(prev,
 next);
    double d2 = g.distance(n,n1);
    Enumeration e = t.elements();
    while(e.hasMoreElements()) {
      ListNode itr = (ListNode)e.nextElement();
      //if(itr.equals(n) || itr.equals(n1) || itr.equals(prev
 ) ) continue;
      if(false) continue;
      else {
        logger.assertLog(!itr.equals(depot),"depot should not
 be in the OROpt");
        ListNode itrNext = (ListNode)itr.getNext();
        double delta_d1 = d1+d2;
        double delta_d2 =0D-d2-g.distance(itr,n);
        if(itrNext!=null) delta_d2 += g.distance(itr,itrNext)
-g.distance(n1,itrNext);
        double newDist = distance-delta_d1-2*serviceTime;
        double newDist1 = t.distance - delta_d2+2*serviceTime
;
        double saving = delta_d1+delta_d2;
        if(newDist<=g.getDisConstraint() && newDist1<=g.
getDisConstraint()) {
          if((float) saving > 1e-5) {
            improved = true;
            prev.setNext(next);
            if(next!=null) next.setPrevious(prev);
```

```
                itr.setNext(n);
                n.setPrevious(itr);
                n1.setNext(itrNext);
                if(itrNext!=null) itrNext.setPrevious(n1);
                this.numElements-=2;
                if(this.numElements==0)
                  if(debug)
                    logger.info("reduce one tour "+ tourID+" in
 or-opt with saving="+saving);
                t.numElements+=2;
                //distance -= delta_d1;
                distance = newDist;
                t.distance = newDist1;
                //t.distance -= delta_d2;
                this.load-=transferedLoad;
                t.load+=transferedLoad;
                //if(saving<0) t.clean();
                //logger.info("find a OROpt Move between routes
 with saving="+saving);
              }
            }
          }
          if(improved) break;
        }
      }
      checkDistance("orOpt move between tour");
      return improved;
    }



    /**
     * Do limited 3-opt for this node n only within this vehicle
     * reloated a chain of 3 nodes into another places
     * @param n
     * @return
     */
    public void threeOptMove() {
      while(threeOptMoveStep()) ;

    }

    private boolean threeOptMoveStep() {
      boolean improved = false;
      ListNode[] list = toArray();
```

```
  for(int i=0; i<list.length;i++) {
    ListNode from = list[i];
    improved = threeOptSingleStep(from);
    if(improved) break;
  }
  return improved;
}

private boolean threeOptSingleStep(ListNode n) {
  boolean improved = false;
  //Node current = (Node) n.getData();
  //logger.assertLog(length()>2,"in OROptMove less than 3
 nodes");
  if(n.getNext()==null||n.getNext().getNext()==null) return
 false; //we need at least two nodes to do or-opt
  else {
    ListNode prev = (ListNode) n.getPrevious();
    ListNode n1 = (ListNode)n.getNext();
    ListNode next = (ListNode)n1.getNext();
    ListNode nnext = (ListNode)next.getNext();
    double d1 = g.distance(prev,n);
    if(nnext!=null) d1+=g.distance(next,nnext)-g.distance(
 prev,nnext);
    Enumeration e = this.elements();
    while(e.hasMoreElements()) {
      ListNode itr = (ListNode)e.nextElement();
      if(itr.equals(n) || itr.equals(n1) || itr.equals(prev)
 ||itr.equals(next) ) continue;
      else {
        logger.assertLog(!itr.equals(depot),"depot should not
 be in the OROpt");
        ListNode itrNext = (ListNode)itr.getNext();
        double saving = d1-g.distance(itr,n);
        if(itrNext!=null) saving +=g.distance(itr,itrNext)-g.
 distance(next,itrNext);
        if(saving>1e-5) {
          improved = true;
          prev.setNext(nnext);
          if(nnext!=null) nnext.setPrevious(prev);
          itr.setNext(n);
          n.setPrevious(itr);
          next.setNext(itrNext);
          if(itrNext!=null) itrNext.setPrevious(next);
          distance -= saving;
          //logger.info("find a OROpt Move with saving="+
```

```
 saving);
            break;
          }
        }
      }
    }
    checkDistance();
    return improved;
  }

  public boolean threeOptMove(Tour t) {
    boolean improved = true;
    while(improved) {
      improved = false;
      Enumeration elm = elements();
      while(elm.hasMoreElements()) {
        ListNode from = (ListNode) elm.nextElement();
        improved = threeOptMove(from,t);
        if(improved) break;
      }
    }
    return improved;
  }
  // OR opt in tour t , trying to insert n and n1 into t
  public boolean threeOptMove(ListNode n, Tour t) {
    double serviceTime = g.getServiceTime();
    boolean improved = false;
    Node current = (Node) n.getData();
    //logger.assertLog(length()>2,"in OROptMove less than 3
 nodes");
    if(n.getNext()==null || n.getNext().getNext()==null) return
    false;
    else {
      ListNode prev = (ListNode) n.getPrevious();
      ListNode n1 = (ListNode)n.getNext();
      ListNode next = (ListNode)n1.getNext();
      ListNode nnext = (ListNode) next.getNext();
      int transferedLoad = current.getDemand()+((Node)n1.data).
 getDemand()+((Node)next.data).getDemand();
      if(t.getLoad() + transferedLoad>g.getCapacity()) return
 false;
      double d1 = g.distance(prev,n);
      if(nnext!=null) d1+= g.distance(next,nnext)-g.distance(
 prev,nnext);
      double d2 = g.distance(n,n1)+g.distance(n1,next);
```

```
    Enumeration e = t.elements();
    while(e.hasMoreElements()) {
      ListNode itr = (ListNode)e.nextElement();
      //if(itr.equals(n) || itr.equals(n1) || itr.equals(prev
) ) continue;
      if(false) continue;
      else {
        logger.assertLog(!itr.equals(depot),"depot should not
 be in the OROpt");
        ListNode itrNext = (ListNode)itr.getNext();
        double delta_d1 = d1+d2;
        double delta_d2 =0D-d2-g.distance(itr,n);
        if(itrNext!=null) delta_d2 += g.distance(itr,itrNext)
-g.distance(next,itrNext);
        double newDist = distance-delta_d1-3*serviceTime;
        double newDist1 = t.distance - delta_d2+3*serviceTime
;
        double saving = delta_d1+delta_d2;
        if(newDist<=g.getDisConstraint() && newDist1<=g.
getDisConstraint()) {
          if((float) saving>1e-5) {
            improved = true;
            prev.setNext(nnext);
            if(nnext!=null) nnext.setPrevious(prev);
            itr.setNext(n);
            n.setPrevious(itr);
            next.setNext(itrNext);
            if(itrNext!=null) itrNext.setPrevious(next);
            this.numElements-=3;
            t.numElements+=3;
            //distance -= delta_d1;
            distance = newDist;
            t.distance = newDist1;
            //t.distance -= delta_d2;
            this.load-=transferedLoad;
            t.load+=transferedLoad;
            if(this.numElements==0){
              if(debug)
                logger.info("reduce one tour "+ tourID+" in
 3-opt! with saving="+saving);
              logger.assertLog(load==0,"the load should be
 empty!");
              logger.assertLog(Math.abs(distance)<1e-10,"the
 distance should be zero");
            }
```

```
          }
        }
      }
      if(improved) break;
    }
  }
  checkDistance();
  return improved;
}
// search if the node is in the tour
// return : the ListNode if yes
//       otherwise return null
public ListNode locate(int nodeId) {
  ListNode ret = null;
  Enumeration e = elements();
  while (e.hasMoreElements()) {
    ListNode node = (ListNode) e.nextElement();
    if (((VRPNode)node.getData()).getID()==nodeId) {
      ret = node;
      break;
    }
  }
  return ret;
}

public void clean() {
  twOpt();
  onePointMove();
  orOptMove();
}

public void optimize_tour() {
  if(numElements>11) return;
  //long startTime = System.currentTimeMillis();
  int [] bestSeq=new int[numElements];
  double bestLen = distance;
  boolean improved = false;
  PermutationGenerator x = new PermutationGenerator(
 numElements);
  int [] perm = new int[numElements];
  final int nodeIDs[] = getNodeID();
  while(x.hasMore()) {
    perm = x.getNext();
    double curDist = g.getOVRPDistance(perm,nodeIDs);
    if(curDist<bestLen-1e-5){
```

```
        System.arraycopy(perm,0,bestSeq,0,perm.length);
        bestLen = curDist;
        improved = true;
      }
    }
    if(improved) reSequence(bestSeq,nodeIDs);
  }

  private void reSequence(int[] seq, int []nodeIDs) {
    VRPNode[] nodes = g.nodes;
    depot.setPrevious(null);
    depot.setNext(null);
    numElements = 0;
    distance = 0D;
    load = 0;
    for(int i=0;i<seq.length;i++) {
      add(nodes[nodeIDs[seq[i]]]);
      //System.out.println("adding node "+ nodeIDs[seq[i]]);
    }
    //checkDistance();
  }

  private int[] getNodeID() {
    int [] id = new int[numElements];
    int cursor = 0;
    Enumeration elm = elements();
    while(elm.hasMoreElements()) {
      ListNode cust = (ListNode)elm.nextElement();
      id[cursor++] = cust.getId();
    }
    return id;
  }

  public double getOriginalDistance() {
    double ret = 0D;
    Enumeration elm = elements();
    ListNode cur = depot;
    while(elm.hasMoreElements()) {
      ListNode next = (ListNode) elm.nextElement();
      ret += g.orig_dist[cur.getId()][next.getId()];
      cur = next;
    }
    return ret;
  }
```

```
// replace one point from this tour with two other points
// from tour t
public boolean twoOneMove(Tour t) {
  final double st = g.getServiceTime();
  boolean improved = false;
  Enumeration e = elements();
  while(e.hasMoreElements()) {
    ListNode cur = (ListNode) e.nextElement();
    ListNode prev = (ListNode) cur.getPrevious();
    ListNode next = (ListNode) cur.getNext();
    // calculate the distance
    double delta1 = g.distance(prev,cur) ;
    if(next!=null) delta1 += g.distance(cur,next);

    int fd = cur.getDemand();
    Enumeration elm = t.elements();
    while(elm.hasMoreElements()){
      ListNode t1 = (ListNode) elm.nextElement();
      if(t1.getNext()==null) break;
      ListNode t2 = (ListNode) t1.getNext();
      ListNode tp = (ListNode)t1.getPrevious();
      ListNode tn = (ListNode)t2.getNext();

      int dt = t1.getDemand() + t2.getDemand();
      int newload1 = getLoad() -fd + dt;
      int newload2 = t.getLoad() + fd - dt;
      if(newload1 <= g.getCapacity() && newload2 <= g.
 getCapacity()) {
        double dd = g.distance(prev,t1)+g.distance(t1,t2);
        if(next!=null) dd+=g.distance(t2,next);
        double newdist1 = distance - delta1 + dd + st;
        double da = g.distance(tp,t1) + g.distance(t1,t2);
        if(tn!=null) da+=g.distance(t2,tn);
        double xx = g.distance(tp,cur);
        if(tn!=null) xx += g.distance(cur,tn);
        double newdist2 = t.distance - da + xx -st;
        if((float)newdist1<=g.getDisConstraint()-1e-5 && (
 float)newdist2<=g.getDisConstraint()-1e-5){
          double saving = delta1 + da - dd - xx;
          if((float)saving>1e-5) {
            //logger.info("2-1 move find a saving="+saving);
            prev.setNext(t1); t1.setPrevious(prev);
            t2.setNext(next); if(next!=null) next.setPrevious
 (t2);
            tp.setNext(cur); cur.setPrevious(tp);
```

```
            cur.setNext(tn); if(tn!=null) tn.setPrevious(cur)
 ;
            load = newload1;
            t.load = newload2;
            distance = newdist1;
            t.distance = newdist2;
            numElements++;
            t.numElements--;
            improved = true;
          }
        }
      }
      if(improved) break;
    }
    if(improved) break;
  }
  checkDistance();
  return improved;
}

public boolean threeOneMove(Tour t){
  final double st = g.getServiceTime();
  boolean improved = false;
  Enumeration e = elements();
  while(e.hasMoreElements()) {
    ListNode cur = (ListNode) e.nextElement();
    ListNode prev = (ListNode) cur.getPrevious();
    ListNode next = (ListNode) cur.getNext();
    // calculate the distance
    double delta1 = g.distance(prev,cur) ;
    if(next!=null) delta1 += g.distance(cur,next);

    int fd = cur.getDemand();
    Enumeration elm = t.elements();
    while(elm.hasMoreElements()){
      ListNode t1 = (ListNode) elm.nextElement();
      if(t1.getNext()==null) break;
      ListNode t2 = (ListNode) t1.getNext();
      ListNode tp = (ListNode)t1.getPrevious();
      ListNode t3 = (ListNode) t2.getNext();
      if(t3==null) break;
      ListNode tn = (ListNode)t3.getNext();

      int dt = t1.getDemand() + t2.getDemand() + t3.getDemand
 ();
```

```
      int newload1 = getLoad() -fd + dt;
      int newload2 = t.getLoad() + fd - dt;
      if(newload1 <= g.getCapacity() && newload2 <= g.
 getCapacity()) {
        double dd = g.distance(prev,t1)+g.distance(t1,t2) + g
 .distance(t2,t3);
        if(next!=null) dd+=g.distance(t3,next);
        double newdist1 = distance - delta1 + dd + 2*st;
        double da = g.distance(tp,t1) + g.distance(t1,t2) + g
 .distance(t2,t3);
        if(tn!=null) da+=g.distance(t3,tn);
        double xx = g.distance(tp,cur);
        if(tn!=null) xx += g.distance(cur,tn);
        double newdist2 = t.distance - da + xx -2*st;
        if((float)newdist1<=g.getDisConstraint()-1e-5 && (
 float)newdist2<=g.getDisConstraint()-1e-5){
          double saving = delta1 + da - dd - xx;
          if((float)saving>1e-5) {
            //logger.info("3-1 move find a saving="+saving);
            prev.setNext(t1); t1.setPrevious(prev);
            t3.setNext(next); if(next!=null) next.setPrevious
 (t3);
            tp.setNext(cur); cur.setPrevious(tp);
            cur.setNext(tn); if(tn!=null) tn.setPrevious(cur)
 ;
            load = newload1;
            t.load = newload2;
            distance = newdist1;
            t.distance = newdist2;
            numElements+=2;
            t.numElements-=2;
            improved = true;
          }
        }
      }
      if(improved) break;
    }
    if(improved) break;
  }
  checkDistance();
  return improved;
}

public boolean threeTwoMove(Tour t){
  final double st = g.getServiceTime();
```

```
 boolean improved = false;
 Enumeration e = elements();
 while(e.hasMoreElements()) {
   ListNode cur = (ListNode) e.nextElement();
   ListNode prev = (ListNode) cur.getPrevious();
   ListNode next = (ListNode) cur.getNext();
   if(next==null) break;
   ListNode nnext = (ListNode) next.getNext();
   // calculate the distance
   double delta1 = g.distance(prev,cur) + g.distance(cur,
next);
   if(nnext!=null) delta1 += g.distance(next,nnext);

   int fd = cur.getDemand() + next.getDemand();
   Enumeration elm = t.elements();
   while(elm.hasMoreElements()){
     ListNode t1 = (ListNode) elm.nextElement();
     if(t1.getNext()==null) break;
     ListNode t2 = (ListNode) t1.getNext();
     ListNode tp = (ListNode)t1.getPrevious();
     ListNode t3 = (ListNode) t2.getNext();
     if(t3==null) break;
     ListNode tn = (ListNode)t3.getNext();

     int dt = t1.getDemand() + t2.getDemand() + t3.getDemand
();
     int newload1 = getLoad() -fd + dt;
     int newload2 = t.getLoad() + fd - dt;
     if(newload1 <= g.getCapacity() && newload2 <= g.
getCapacity()) {
       double dd = g.distance(prev,t1)+g.distance(t1,t2) + g
.distance(t2,t3);
       if(nnext!=null) dd+=g.distance(t3,nnext);
       double newdist1 = distance - delta1 + dd + st;
       double da = g.distance(tp,t1) + g.distance(t1,t2) + g
.distance(t2,t3);
       if(tn!=null) da+=g.distance(t3,tn);
       double xx = g.distance(tp,cur) + g.distance(cur,next)
;
       if(tn!=null) xx += g.distance(next,tn);
       double newdist2 = t.distance - da + xx -st;
       if((float)newdist1<=g.getDisConstraint()-1e-5 && (
float)newdist2<=g.getDisConstraint()-1e-5){
         double saving = delta1 + da - dd - xx;
         if((float)saving>1e-5) {
```

```
                //logger.info("3-2 move find a saving="+saving);
                prev.setNext(t1); t1.setPrevious(prev);
                t3.setNext(nnext); if(nnext!=null) nnext.
    setPrevious(t3);
                tp.setNext(cur); cur.setPrevious(tp);
                next.setNext(tn); if(tn!=null) tn.setPrevious(
    next);
                load = newload1;
                t.load = newload2;
                distance = newdist1;
                t.distance = newdist2;
                numElements++;
                t.numElements--;
                improved = true;
              }
            }
          }
          if(improved) break;
        }
        if(improved) break;
      }
      checkDistance();
      t.checkDistance();
      return improved;
    }

    public void clean(Tour t){
      while(onePointMove(t)) ;
      while(twoPointMove(t)) ;
      while(orOptMove(t)) ;
      while(twOpt(t)) ;
      while(threeOneMove(t)) ;
      while(threeTwoMove(t)) ;
      while(twoOneMove(t)) ;
    }
}
```

C.4 Linkable.java

```
package ovrp;

public interface Linkable
{
  public Linkable getNext();
```

```
  public Linkable getPrevious();


  public void setNext( Linkable node );


  public void setPrevious( Linkable node );
}
```

C.5 ListNode.java

```
package ovrp;

public class ListNode implements Linkable,Cloneable,Comparable
{
  public ListNode (Object data )
  {
    this.data = data;
    this.next = this.previous = null;
  }


  protected ListNode()  {}


  public Object getData()  { return data; }


  public int getDemand() {
    return ((VRPNode)data).getDemand();
  }

  public void setRoute(int rid) {
    tid = rid;
  }

  public int getRoute() {
    return tid;
  }
  public Linkable getNext()  { return next; }


  public Linkable getPrevious()  { return previous; }
```

```
public boolean equals(Object C) {
  if(C instanceof ListNode)
    return data.equals( ((ListNode)C).getData() );
  else return false;
}


public void setData( Comparable data )  { this.data = data; }

public int compareTo(Object that) {
  if(! (that instanceof ListNode)) throw new
 IllegalArgumentException();
  else {
    int thatId = ((VRPNode) (((ListNode)that).data)).getID();
    int thisId = ((VRPNode)data).getID();
    if(thisId< thatId) return -1;
    else if(thisId>thatId) return 1;
    else return 0;
  }
}
public void setNext( Linkable node )  { next = (ListNode)node
 ; }


public void setPrevious( Linkable node )
{
  previous = (ListNode)node;
}

protected void swap() {
  ListNode tmp = previous;
  previous = next;
  next = tmp;
}

public Object clone() {
  ListNode copy = null;
  try {
    copy = (ListNode)super.clone();
  } catch(CloneNotSupportedException e) {}
  return copy;
}
```

```
  public String toString()  { return ( "" + data ); }


  protected Object data;

  private int tid; // tour id

  protected ListNode next;

  protected ListNode previous;

  public int getId() {
    return ( (VRPNode) data).getID();
  }
}
```

C.6 SolutionPlot.java

```
package ovrp;
import java.io.*;
import java.util.*;

public class SolutionPlot {
  private String data;
  PrintWriter pw;

  public SolutionPlot(String output){
    data = output;
    try {
      FileWriter fw = new FileWriter(data,false);
      pw = new PrintWriter(new BufferedWriter(fw));
    } catch(FileNotFoundException e) {}
    catch(IOException ie){System.out.println(ie);}
  }

  public void print(VRPNode n) {
    pw.println(n.x+"\t"+n.y);

  }


  public void print(Tour t){
    print(t.getDepot());
    Enumeration e = t.elements();
    while (e.hasMoreElements()) {
```

```
      ListNode n = (ListNode)e.nextElement();
      print((VRPNode) n.getData());
    }
    print("\n\n");

  }
/*
  public void print(Vehicle v) {
    print(v.getDepot());
    print(v.getTour());
    print(v.getDepot());
    print("\n\n");//required by gnuplot to separate data
  }
*/
  public void print(String s) { pw.print(s); }

    public void print(OVRPSolution s) {
      Tour[] v = s.getTours();
      for(int i=0;i<v.length;i++) {
        if(!v[i].isEmpty()) print(v[i]);
      }
    }
/*
    public void print(OVRPInstance ins) {
      print(ins.getSolution());
    }
    */
  public void close() {
    pw.flush();
    pw.close();
  }
}
```

C.7 Vehicle.java

```
/*
 * Created on 2003-10-15
 *
 * To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and
   Comments
 */
package ovrp;
//import java.lang.reflect.*;
```

```
import java.util.Enumeration;
import org.apache.log4j.*;

/**
 * @author lify
 *
 * To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Generation - Code and
   Comments
 */
public class Vehicle implements Comparable,Cloneable{
  Tour tour;
  VRPGraph g;
  //ArrayList nodes;
  int cap;
  //int load;
  //double distance;
  double maxLen;
  double serviceTime;
  //private Node depot;
  private static Logger logger =Logger.getLogger(Vehicle.class)
   ;
  /**
   * @param type
   * @param depot
   */
  public Vehicle(VRPGraph g) {
      this.g = g;
      cap = g.getCapacity();
      maxLen = g.getDisConstraint();
      serviceTime = g.getServiceTime();
      //load = 0;
      //distance = 0D;
      tour = new Tour(g);
  }

  public int compareTo(Object o)  {
    if(! (o instanceof Vehicle)) throw new
   IllegalArgumentException();
    double cap = ((Vehicle)o).getLoad();
    double thiscap = getLoad();
    if(thiscap>cap) return -1; // because we want descending
   order
    else if (thiscap==cap) return 0;
    else return 1;
```

```
}

public int getLoad() {
  return tour.getLoad();
}

/**
 * @param depot The depot to set.
 */
//public void setDepot(VRPNode depot) {
//  this.depot = depot;
//}



public int getFreeLoad() {
  return cap - tour.getLoad();
}



public boolean add(VRPNode n) {
  int load = tour.getLoad();
  double dist = tour.getDistance();
  if(leastCostInsertion(n)) {
    if(tour.getDistance()<=maxLen)
    return true;
    else {
        if(tour.getDistance()+g.getServiceTime()>=maxLen
 *0.90) return false;
        else {
      tour.optimize_tour();
      if(tour.getDistance()<=maxLen) return true;
      else {
          System.out.println("[Vehicle]removing customer "+ n
 .getID());
          tour.remove(n);
          tour.checkDistance();
          //tour.checkDistance("in vehicle.add");
          System.out.println("before load="+load+",distance="
 +dist);
          System.out.println("after load="+tour.getLoad()+",
 distance="+tour.getDistance());
          return false;
      }
```

```
        }
      }
    }
    else return false;
  }

  /**
   * @param o
   */
  public boolean leastCostInsertion(VRPNode o) {
    if(o.getDemand()+tour.getLoad()<=cap) {
      tour.leastCostInsertion(o);
      return true;
    }
    else return false;
  }

  public String toString() {
    String s = new String();
    return s+"capacity="+cap+",load="+ tour.getLoad() + ",
   distance="+tour.getDistance()+" ("+tour+")";
  }
  /**
   * @return Returns the tour.
   */
  public Tour getTour() {
    return tour;
  }

  public boolean isEmpty() {
    return tour.isEmpty();
  }

  /**
   * @return Returns the distance.
   */
  public double getDistance() {
    return tour.getDistance();
  }

  /**
   * n is a node in this vehicle
   */
  protected Object clone() throws CloneNotSupportedException {
    // TODO Auto-generated method stub
```

```
    Vehicle copy = (Vehicle) super.clone();
    copy.tour = (Tour) tour.clone();
    return copy;
  }

  public void checkSolution() {
      tour.checkDistance();
  }

  public Enumeration elements() {
      return tour.elements();
  }

}
```

C.8 PriorityQueue.java

```
//package edu.wlu.cs.levy.CG;
package ovrp;

class PriorityQueue implements java.io.Serializable {

   /**
    * This class implements a <code>PriorityQueue</code>. This
   class
    * is implemented in such a way that objects are added using
    an
    * <code>add</code> function. The <code>add</code> function
   takes
    * two parameters an object and a long.
    * <p>
    * The object represents an item in the queue, the long
   indicates
    * its priority in the queue. The remove function in this
   class
    * returns the object first in the queue and that object is
   removed
    * from the queue permanently.
    *
    * @author  Bjoern Heckel
    * @version %I%, %G%
    * @since JDK1.2
    */

   /**
```

```
     * The maximum priority possible in this priority queue.
     */
    private double maxPriority = Double.MAX_VALUE;

    /**
     * This contains the list of objects in the queue.
     */
    private Object[] data;

    /**
     * This contains the list of prioritys in the queue.
     */
    private double[] value;

    /**
     * Holds the number of elements currently in the queue.
     */
    private int count;

    /**
     * This holds the number elements this queue can have.
     */
    private int capacity;

    /**
     * Creates a new <code>PriorityQueue</code> object. The
     * <code>PriorityQueue</code> object allows objects to be
     * entered into the queue and to leave in the order of
     * priority i.e the highest priority get's to leave first.
     */
    public PriorityQueue() {
        init(20);
    }

    /**
     * Creates a new <code>PriorityQueue</code> object. The
     * <code>PriorityQueue</code> object allows objects to
     * be entered into the queue an to leave in the order of
     * priority i.e the highest priority get's to leave first.
     *
     * @param capacity the initial capacity of the queue before
     * a resize
     */
    public PriorityQueue(int capacity) {
        init(capacity);
```

```
}

/**
 * Creates a new <code>PriorityQueue</code> object. The
 * <code>PriorityQueue</code> object allows objects to
 * be entered into the queue an to leave in the order of
 * priority i.e the highest priority get's to leave first.
 *
 * @param capacity the initial capacity of the queue before
 * a resize
 * @param maxPriority is the maximum possible priority for
 * an object
 */
public PriorityQueue(int capacity, double maxPriority) {
    this.maxPriority = maxPriority;
    init(capacity);
}

/**
 * This is an initializer for the object. It basically
initializes
 * an array of long called value to represent the prioritys
of
 * the objects, it also creates an array of objects to be
used
 * in parallel with the array of longs, to represent the
objects
 * entered, these can be used to sequence the data.
 *
 * @param size the initial capacity of the queue, it can be
 * resized
 */
private void init(int size) {
    capacity = size;
    data = new Object[capacity + 1];
    value = new double[capacity + 1];
    value[0] = maxPriority;
    data[0] = null;
}

/**
 * This function adds the given object into the <code>
PriorityQueue</code>,
 * its priority is the long priority. The way in which
priority can be
```

```
 * associated with the elements of the queue is by keeping
the priority
 * and the elements array entrys parallel.
 *
 * @param element is the object that is to be entered into
this
 * <code>PriorityQueue</code>
 * @param priority this is the priority that the object
holds in the
 * <code>PriorityQueue</code>
 */
public void add(Object element, double priority) {
    if (count++ >= capacity) {
        expandCapacity();
    }
    /* put this as the last element */
    value[count] = priority;
    data[count] = element;
    bubbleUp(count);
}

/**
 * Remove is a function to remove the element in the queue
with the
 * maximum priority. Once the element is removed then it can
 never be
 * recovered from the queue with further calls. The lowest
priority
 * object will leave last.
 *
 * @return the object with the highest priority or if it's
empty
 * null
 */
public Object remove() {
    if (count == 0)
        return null;
    Object element = data[1];
    /* swap the last element into the first */
    data[1] = data[count];
    value[1] = value[count];
    /* let the GC clean up */
    data[count] = null;
    value[count] = 0L;
    count--;
```

```
    bubbleDown(1);
    return element;
}

public Object front() {
    return data[1];
}

public double getMaxPriority() {
    return value[1];
}

/**
 * Bubble down is used to put the element at subscript 'pos'
 into
 * it's rightful place in the heap (i.e heap is another name
 * for <code>PriorityQueue</code>). If the priority of an
element
 * at subscript 'pos' is less than it's children then it
must
 * be put under one of these children, i.e the ones with the
 * maximum priority must come first.
 *
 * @param pos is the position within the arrays of the
element
 * and priority
 */
private void bubbleDown(int pos) {
    Object element = data[pos];
    double priority = value[pos];
    int child;
    /* hole is position '1' */
    for (; pos * 2 <= count; pos = child) {
        child = pos * 2;
        /* if 'child' equals 'count' then there
           is only one leaf for this parent */
        if (child != count)

            /* left_child > right_child */
            if (value[child] < value[child + 1])
                child++; /* choose the biggest child */
                /* percolate down the data at 'pos', one
level
                    i.e biggest child becomes the parent */
        if (priority < value[child]) {
```

```
                value[pos] = value[child];
                data[pos] = data[child];
            }
            else {
                break;
            }
        }
        value[pos] = priority;
        data[pos] = element;
    }

    /**
     * Bubble up is used to place an element relatively low in
    the
     * queue to it's rightful place higher in the queue, but
    only
     * if it's priority allows it to do so, similar to
    bubbleDown
     * only in the other direction this swaps out its parents.
     *
     * @param pos the position in the arrays of the object
     * to be bubbled up
     */
    private void bubbleUp(int pos) {
        Object element = data[pos];
        double priority = value[pos];
        /* when the parent is not less than the child, end*/
        while (value[pos / 2] < priority) {
            /* overwrite the child with the parent */
            value[pos] = value[pos / 2];
            data[pos] = data[pos / 2];
            pos /= 2;
        }
        value[pos] = priority;
        data[pos] = element;
    }

    /**
     * This ensures that there is enough space to keep adding
    elements
     * to the priority queue. It is however advised to make the
    capacity
     * of the queue large enough so that this will not be used
    as it is
     * an expensive method. This will copy across from 0 as 'off
```

```
    ' equals
     * 0 is contains some important data.
     */
    private void expandCapacity() {
        capacity = count * 2;
        Object[] elements = new Object[capacity + 1];
        double[] prioritys = new double[capacity + 1];
        System.arraycopy(data, 0, elements, 0, data.length);
        System.arraycopy(value, 0, prioritys, 0, data.length);
        data = elements;
        value = prioritys;
    }

    /**
     * This method will empty the queue. This also helps garbage
     * collection by releasing any reference it has to the
    elements
     * in the queue. This starts from offset 1 as off equals 0
     * for the elements array.
     */
    public void clear() {
        for (int i = 1; i < count; i++) {
            data[i] = null; /* help gc */
        }
        count = 0;
    }

    /**
     * The number of elements in the queue. The length
     * indicates the number of elements that are currently
     * in the queue.
     *
     * @return the number of elements in the queue
     */
    public int length() {
        return count;
    }
}
```

Appendix D

Heterogeneous Vehicle Routing Problem Code

D.1 HVRPSolution.java

```
/*
 * Created on 2003-10-27
 *
 * To change the template for this generated file go to Window
   - Preferences -
 * Java - Code Generation - Code and Comments
 */
package hvrp;
import java.util.*;
import org.apache.log4j.*;
/**
 * @author lify
 *
 * To change the template for this generated type comment go to
    Window -
 * Preferences - Java - Code Generation - Code and Comments
 */
public class HVRPSolution implements Cloneable, Comparator,
   Visitable {

    // all the vehicles that serve this solution
    private Vehicle[] vehicles;

    static Logger logger = Logger.getLogger(HVRPSolution.class)
   ;
    static MyRandom rand = new MyRandom();
    /**
     * @return Returns the fleet.
     */
```

```
  //private double length;

  public Vehicle[] getVehicles() {
return vehicles;
  }

  /**
   * @param fleet
   *                    The fleet to set.
   */
  public void setVehicles(Vehicle[] fleet) {
this.vehicles = fleet;
  }

  // the default constructor should not be invoked
  private HVRPSolution(){ }

  public HVRPSolution(Vehicle[] v) {
vehicles = v;
  }

  public void add(Node n) {
// need some criterion for insert node into vehicles
logger.error("HVRPSolution.add() should not be called");
  }

  public String toString() {
double totalDist = 0D;
String s = "\n";
for (int i = 0; i < vehicles.length; i++) {
    s += vehicles[i].toString() + "\n";
    totalDist += vehicles[i].getDistance();
}
return s + " total distance =" + totalDist + "\n";
  }

  public void twOpt() {
while (twOptStep());
//this.checkSolution("after twopt");
  }

  private boolean twOptStep() {
boolean improved = false;
for (int i = 0; i < vehicles.length; i++) {
    //first do local twopt
```

```
    Enumeration e = vehicles[i].getTour().elements();
    while (e.hasMoreElements()) {
  ListNode current = (ListNode) e.nextElement();
  for (int j = 0; j < vehicles.length; j++) {
      if (j == i) {
    improved = vehicles[i].twOpt(current);
      } else {
    improved = vehicles[i].twOpt(current, vehicles[j]);
      }
      if (improved)
    break;
  }
  if (improved)
      break;
    }
    if (improved)
  break;
}
return improved;
  }

  public void onePointMove() {
while (onePointMoveStep());
//this.checkSolution("after onepoint move");
  }

  private boolean onePointMoveStep() {
boolean improved = false;
for (int i = 0; i < vehicles.length; i++) {
    Enumeration e = vehicles[i].getTour().elements();
    while (e.hasMoreElements()) {
  ListNode current = (ListNode) e.nextElement();
  for (int j = 0; j < vehicles.length; j++) {
      if (j == i) {
    improved = vehicles[i].onePointMove(current);
      } else {
    improved =
        vehicles[i].onePointMove(current, vehicles[j]);
      }
      if (improved)
    break;
  }
  if (improved)
      break;
    }
```

```
if (improved)
  break;
}
return improved;
  }

  public void chainOpt(int size) {
while (chainOptStep(size));
  }

  public boolean chainOptStep(int chainSize) {
boolean improved = false;
for (int i = 0; i < vehicles.length; i++) {
}
return false;
  }
  /*
   * (non-Javadoc)
   *
   * @see java.lang.Object#clone()
   */
  public Object clone() throws CloneNotSupportedException {
// TODO Auto-generated method stub
HVRPSolution copy = (HVRPSolution) super.clone();
copy.vehicles = (Vehicle[]) vehicles.clone();
for (int i = 0; i < vehicles.length; i++)
    copy.vehicles[i] = (Vehicle) vehicles[i].clone();
return copy;
  }

  public void twoPointMove() {
//     for(int i=0;i<vehicles.length;i++)
//       vehicles[i].twoPointMove();
while (twoPointMoveStep());
//this.checkSolution("after twopointmove");
  }

  private boolean twoPointMoveStep() {
boolean improved = false;
for (int i = 0; i < vehicles.length; i++) {
    Enumeration e = vehicles[i].getTour().elements();
    while (e.hasMoreElements()) {
  ListNode current = (ListNode) e.nextElement();
  for (int j = 0; j < vehicles.length; j++) {
      if (j == i) {
```

```
    improved = vehicles[i].twoPointMove(current);
      } else {
    improved =
        vehicles[i].twoPointMove(current, vehicles[j]);
      }
      if (improved)
    break;
  }
  if (improved)
      break;
    }
    if (improved)
  break;
}
return improved;
  }

  public void OROpt() {
while(OROptStep());
  }

  private boolean OROptStep() {
boolean improved = false;
for(int i=0;i< vehicles.length;i++) {
    if(vehicles[i].length()<=2) continue;
    else {
  Enumeration e = vehicles[i].getTour().elements();
  while(e.hasMoreElements()) {
      ListNode current = (ListNode)e.nextElement();
      for(int j=0;j<vehicles.length;j++) {
    if(j==i) {
        improved = vehicles[i].OROptMove(current);
    } else {
        improved = vehicles[i].OROptMove(current,vehicles[j])
 ;
    }
    if(improved) break;
      }
      if(improved) break;
  }
  if(improved) break;
    }
}
return improved;
  }
```

```
  public double getDistance() {
double length = 0D;
for (int i = 0; i < vehicles.length; i++) {
    length += vehicles[i].getDistance();
}
return length;
  }

  public void cleanUp() {
double d1, d2, d3;
do {
    twOpt();
    d1 = getDistance();
    onePointMove();
    d2 = getDistance();
    twoPointMove();
    d3 = getDistance();
} while (Math.abs(d1 - d2) > 1e-10 || Math.abs(d2 - d3) > 1e
 -10);
  }

  public void clean() {
twOpt();
onePointMove();
twoPointMove();
OROpt();
  }

  public boolean twOptRTR(Record r) {
//logger.info("enter 2opt RTR");
//logger.info(this);
allOff();
boolean moved = false;
int total = GraphFactory.getGraph().size() - 1;
// because we include the depot
int count = 0;
while (count < total) {
    for (int i = 0; i < vehicles.length; i++) {
  Enumeration e = vehicles[i].getTour().elements();
  while (e.hasMoreElements()) {
      ListNode current = (ListNode) e.nextElement();
      if (!current.isDone()) {
    current.setDone(true);
    count++;
```

```
    //logger.info(current);
    RTRInfo bestInfo = new RTRInfo(null, null, -1e8);
    for (int j = 0; j < vehicles.length; j++) {
        RTRInfo info;
        //if(vehicles[j].getTour().contains(current.getData()
))
        // info = vehicles[i].get2OptRTRInfo(current,r);
        if (i == j)
      info = vehicles[i].get2OptRTRInfo(current, r);
        // here we have a problem, current maybe no longer
        // in vehicle i after 2opt RTR
        else
      info =
          vehicles[i].get2OptRTRInfo(
            current,
            vehicles[j],
            r);
        if (info.getSaving() > bestInfo.getSaving())
      bestInfo = info;
    }
    if (bestInfo.getNode() != null) {
        vehicles[i].do2OptRTR(current, bestInfo);
        r.setCurrent(getDistance());
        moved = true;
        break;
    }
      }
  }
    }
}
//logger.info("finish 2opt RTR");
return moved;
  }

  public boolean onePointMoveRTR(Record r) {
//logger.info(this);
//logger.info("enter one point move RTR");
allOff();
boolean moved = false;
int total = GraphFactory.getGraph().size() - 1;
// because we include the depot
int count = 0;
while (count < total) {
    //logger.info("[opmRTR]count="+count+",total="+total);
    for (int i = 0; i < vehicles.length; i++) {
```

```
 Enumeration e = vehicles[i].getTour().elements();
 while (e.hasMoreElements()) {
     ListNode current = (ListNode) e.nextElement();
     if (!current.isDone()) {
   current.setDone(true);
   count++;
   //logger.info("[opmRTR]count="+count+",total="+total+",
checking
   // node "+current.getId());
   //logger.info(current);
   RTRInfo bestInfo = new RTRInfo(null, null, -1e8);
   for (int j = 0; j < vehicles.length; j++) {
       RTRInfo info;
       //if(vehicles[j].getTour().contains(current.getData()
))
       // info = vehicles[i].get2OptRTRInfo(current,r);
       if (i == j)
     info =
         vehicles[i].getOnePointMoveRTRInfo(
           current,
           r);
       // here we have a problem, current maybe no longer
       // in vehicle i after 2opt RTR
       else
     info =
         vehicles[i].getOnePointMoveRTRInfo(
           current,
           vehicles[j],
           r);
       if (info.getSaving() > bestInfo.getSaving())
     bestInfo = info;
   }
   if (bestInfo.getNode() != null) {
       vehicles[i].doOnePointMoveRTR(current, bestInfo);
       r.setCurrent(getDistance());
       moved = true;
       break;
   }
     }
 }
   }
}
//logger.info("finish one point move RTR");
return moved;
  }
```

```
  public boolean twoPointMoveRTR(Record r) {
//logger.info(this);
//logger.info("enter 2 point move RTR");
allOff();
boolean moved = false;
int total = GraphFactory.getGraph().size() - 1;
// because we include the depot
int count = 0;
while (count < total) {
    for (int i = 0; i < vehicles.length; i++) {
  Enumeration e = vehicles[i].getTour().elements();
  while (e.hasMoreElements()) {
      ListNode current = (ListNode) e.nextElement();
      if (!current.isDone()) {
    current.setDone(true);
    count++;
    //logger.info(current);
    RTRInfo bestInfo = new RTRInfo(null, null, -1e8);
    for (int j = 0; j < vehicles.length; j++) {
        //if(vehicles[j].getTour().contains(current.getData()
 ))
        // info = vehicles[i].get2OptRTRInfo(current,r);
        RTRInfo info =
      vehicles[i].getTwoPointMoveRTRInfo(
        current,
        vehicles[j],
        r);
        if (info.getSaving() > bestInfo.getSaving())
      bestInfo = info;
    }
    if (bestInfo.getNode() != null) {
        vehicles[i].doTwoPointMoveRTR(current, bestInfo);
        r.setCurrent(getDistance());
        moved = true;
        break;
    }
      }
  }
    }
}
//logger.info("finish 2 point move RTR");
return moved;
  }
```

```
  private void allOff() {
for (int i = 0; i < vehicles.length; i++) {
    Enumeration e = vehicles[i].getTour().elements();
    while (e.hasMoreElements()) {
  ((ListNode) e.nextElement()).setDone(false);
    }
}
  }

  public ListNode[] toArray() {
ArrayList list = new ArrayList();
for (int i = 0; i < vehicles.length; i++) {
    Enumeration e = vehicles[i].getTour().elements();
    while (e.hasMoreElements()) {
  list.add(e.nextElement());
    }
}
return (ListNode[]) list.toArray(new ListNode[0]);
  }

  public HVRPSolution RTR() {
HVRPSolution bestSol = null;
double minLen = 1e10;
int count=0;
while(count<5) {
    HVRPSolution sol = RTR1();
    if(sol.getDistance()<minLen) {
  try {
  bestSol = (HVRPSolution) sol.clone();
  minLen = bestSol.getDistance();
  } catch(CloneNotSupportedException e){}
    }
    HVRPSolution bclone = null;
    try {
  bclone = (HVRPSolution) sol.clone();
    } catch (CloneNotSupportedException e) {
  logger.info(e);
    }
    //bclone.perturb();
    bclone.ruinRecreate();
    HVRPSolution pSol = bclone.RTR1();
    if (pSol.getDistance() < minLen){
  try {
  bestSol = (HVRPSolution) pSol.clone();
  minLen = bestSol.getDistance();
```

```
  } catch(CloneNotSupportedException e) {}
    }
    count++;
}
return bestSol;
  }

  public HVRPSolution RTR1() {
Random rand = new Random();
int counter = 1;
int I =30;
clean();
HVRPSolution bestSol = null;
try {
    bestSol = (HVRPSolution) clone();
} catch (CloneNotSupportedException e) {
}
double bestLength = bestSol.getDistance();
Record r = new Record(bestLength);
r.setCurrent(bestLength);

while (true) {
    boolean improved = false;
    for (int i = 0; i < I; i++) {
  boolean opm = onePointMoveRTR(r);
  boolean tpm = twoPointMoveRTR(r);
  boolean tom = twOptRTR(r);
  if (!opm & !tpm & !tom) {
      logger.info("no movement in loop I. Quit loop I");
      break;
  } else {
      if (getDistance() < bestSol.getDistance()) {
    improved = true;
    try {
        bestSol = (HVRPSolution) clone();
    } catch (CloneNotSupportedException e) {
    }
    r.setCurrent(getDistance());
      }
  }
    }
    clean();
    double currentLength = getDistance();
    if (currentLength < bestSol.getDistance() - 1e-5) {
  //logger.info("find new best solution");
```

```
    try {
        bestSol = (HVRPSolution) clone();
    } catch (CloneNotSupportedException e) {
    }
    r.setRecord(currentLength);
    counter = 0;
      }
      if(currentLength > r.getRecord() && rand.nextDouble()
   >0.6)  r.setRecord(currentLength);
      //if(Math.exp(-(currentLength-r.getRecord())/r.
   getDeviation()) > 0.9)  r.setRecord(currentLength);
      r.setCurrent(currentLength);
      //logger.info("counter=" + counter +" best solution =" +
      // bestSol.getDistance() + "current solution="
      //  + currentLength);
      counter++;
      if (counter>5)
    break;
  }
  return bestSol;
    }

    public void leastCostInitSolution() {
  Vehicle[] v = getVehicles();
  VRPGraph g = GraphFactory.getGraph();
  Node[] nodes = new Node[g.size() - 1];
  System.arraycopy(g.nodes, 1, nodes, 0, g.size() - 1);
  Arrays.sort(nodes);
  for (int i = 0; i < nodes.length; i++) {
      this.leastCostInsertNode(nodes[i]);
  }
    }

private void leastCostInsertNode(Node n) {
    double insCost = 1e8;
    Vehicle insVeh = null;
    for (int k = 0; k < vehicles.length; k++) {
  double theCost = vehicles[k].getLeastCost(n);
  if (theCost < insCost) {
      insCost = theCost;
      insVeh = vehicles[k];
  }
    }
    if (insVeh != null) {
  insVeh.leastCostInsertion(n);
```

```
    } else {
  logger.error(
    "cannot insert node " + n + " with demand=" + n.getDemand()
   );
  logger.error("This should not happend");
    }
}

public void initSolution() {
    Vehicle[] v = getVehicles();
    //sol = new HVRPSolution(v);
    VRPGraph g = GraphFactory.getGraph();
    Node[] nodes = new Node[g.size() - 1];
    System.arraycopy(g.nodes, 1, nodes, 0, g.size() - 1);
    Arrays.sort(nodes);
    Arrays.sort(v);
    for (int i = 0; i < nodes.length; i++) {
  //Arrays.sort(v);
  for (int k = 0; k < v.length; k++) {
      if (v[k].leastCostInsertion(nodes[i]))
    //this.checkSolution();
    break;
      //        else {
      //          logger.info("can't insert node "+i);
      //        }
  }
    }
    logger.info("check solution for initSolution");
    this.checkSolution();
    //buildNBList();
}

public void sweepInitSolution(double initTheta) {
    Vehicle[] v = getVehicles();
    VRPGraph g = GraphFactory.getGraph();
    Node depot = g.getDepot();
    Node[] nodes = new Node[g.size() - 1];
    System.arraycopy(g.nodes, 1, nodes, 0, g.size() - 1);
    //SweepNode [] snodes = new SweepNode[nodes.length];
    for (int i = 0; i < nodes.length; i++)
  nodes[i].setTheta(depot, initTheta);
    Arrays.sort(nodes, this);
    Arrays.sort(v);
    for (int nid = 0; nid < nodes.length; nid++) {
  int vid = -1;
```

```
    double leastCost = 1e10;
    for (int k = 0; k < v.length; k++) {
        double cost = v[k].getLeastCost(nodes[nid]);
        if (cost < leastCost) {
      leastCost = cost;
      vid = k;
        }
    }
    if (vid == -1) {
        throw new IllegalArgumentException();
        /*
         * logger.error( "can't insert node " + nodes[nid] + "
     into the
         * solution,demand=" + nodes[nid].getDemand());
         * //logger.info(this);
         throw new IllegalArgumentException();
         * //System.exit(1);
         */
    } else
        v[vid].leastCostInsertion(nodes[nid]);
      }
      //checkSolution();
}


      public void maxSolution() {
    Vehicle[] v = getVehicles();
    VRPGraph g = GraphFactory.getGraph();
    Node depot = g.getDepot();
    Node[] nodes = new Node[g.size() - 1];
    System.arraycopy(g.nodes, 1, nodes, 0, g.size() - 1);
    Arrays.sort(nodes, new Comparator() {
         public int compare(Object obj1, Object obj2) {
        Node n1 = (Node) obj1;
         Node n2 = (Node) obj2;
        if (n1.getDemand() > n2.getDemand())
             return -1;
         else if (n1.getDemand() == n2.getDemand())
             return 0;
          else return 1;
         }
    });
    Arrays.sort(v); // remember we sort it decreasingly
     java.util.List nodeList = new ArrayList(nodes.length);
     for(int i=0;i<nodes.length;i++) nodeList.add(nodes[i]);
```

```
    for(int i=v.length-1;i>=0;i--) {
        int cap = v[i].getCapacity();
        if(!nodeList.isEmpty()) {
      ListIterator itr = nodeList.listIterator();
      Node cur = null;
      while(itr.hasNext()) {
          cur = (Node)itr.next();
          if(cur.getDemand()<=cap)
         break;
      }
      int currentLoad = cur.getDemand();
      v[i].leastCostInsertion(cur);
      //logger.info("inserting node "+ cur +" into vehicle "+v[
   i]);
      itr.remove();
      while(itr.hasNext()) {
          Node nextNode = (Node)itr.next();
          if(v[i].leastCostInsertion(nextNode)){
        //logger.info("inserting node "+ nextNode +" into
   vehicle "+v[i]);
        itr.remove();
          }
      }
        }
    }
    }

public void sweepInitSolution1(double theta) {
    Vehicle[] v = getVehicles();
    VRPGraph g = GraphFactory.getGraph();
    Node depot = g.getDepot();
    Node[] nodes = new Node[g.size() - 1];
    System.arraycopy(g.nodes, 1, nodes, 0, g.size() - 1);
    //     Arrays.sort(nodes, new Comparator() {
    //       public int compare(Object obj1, Object obj2) {
    //         Node n1 = (Node) obj1;
    //         Node n2 = (Node) obj2;
    //         if (n1.getDemand() > n2.getDemand())
    //           return -1;
    //         else if (n1.getDemand() == n2.getDemand())
    //           return 0;
    //         else
    //           return 1;
    //       }
    //     });
```

```
  ArrayList nodeList = new ArrayList(nodes.length);
  for (int i = 0; i < nodes.length; i++)
nodeList.add(nodes[i]);

  Arrays.sort(v);
  ArrayList vehicleList = new ArrayList(v.length);
  for (int i = 0; i < v.length; i++)
vehicleList.add(v[i]);

  while (!nodeList.isEmpty()) {
Node[] nodeSoFar = (Node[]) nodeList.toArray(new Node[0]);
Arrays.sort(nodeSoFar, new Comparator() {
    public int compare(Object obj1, Object obj2) {
  Node n1 = (Node) obj1;
  Node n2 = (Node) obj2;
  if (n1.getDemand() > n2.getDemand())
      return -1;
  else if (n1.getDemand() == n2.getDemand())
      return 0;
  else
    return 1;
    }
});
nodeList.clear();
for(int i=0;i<nodeSoFar.length;i++) nodeList.add(nodeSoFar[i
 ]);

Node current = (Node) nodeList.remove(0);
// get the node with the largest demand
Vehicle veh = null;
if (!vehicleList.isEmpty())
    veh = (Vehicle) vehicleList.remove(0);
else {
    logger.warn("no enough vehicle to serve customers");
    nodeList.add(0, current);
    break;
    //System.exit(1);
}
veh.add(current);
Node[] remainingNodes = (Node[]) nodeList.toArray(new Node
 [0]);
theta = Math.atan2(current.y - depot.y, current.x - depot.x);
for (int i = 0; i < remainingNodes.length; i++)
    remainingNodes[i].setTheta(depot, theta);
```

```
Arrays.sort(remainingNodes, new Comparator() {
    public int compare(Object obj1, Object obj2) {
  Node n1 = (Node) obj1;
  Node n2 = (Node) obj2;
  double val1 = Math.abs(n1.getTheta());
  double val2 = Math.abs(n2.getTheta());
  if (val1 < val2)
      return -1;
  else if (val1 == val2)
      return 0;
  else
    return 1;
    }
});
nodeList.clear();
for (int i = 0; i < remainingNodes.length; i++)
    nodeList.add(remainingNodes[i]);
ListIterator itr = nodeList.listIterator();
while (itr.hasNext()) {
    Node next = (Node) itr.next();
    if (next.getDemand() <= veh.getFreeLoad()) {
  veh.leastCostInsertion(next);
  itr.remove();
    } else
  break;
}
// now we are done with this vehicle
  }

  // now check if all nodes has been assigned
  if (!nodeList.isEmpty()) {
ListIterator itr = nodeList.listIterator();
while (itr.hasNext()) {
    Node next = (Node) itr.next();
    boolean inserted = false;
    for (int i = 0; i < v.length; i++) {
  if (v[i].getFreeLoad() >= next.getDemand()) {
      v[i].leastCostInsertion(next);
      inserted = true;
  }
    }
    if (!inserted)
  logger.warn(
    "sweep cannot insert customer "
    + next
```

```
      + " with demand "
      + next.getDemand());
  }
    }

    checkSolution();
}

// used in sweep init solution
public int compare(Object obj1, Object obj2) {
    if (!(obj1 instanceof Node && obj2 instanceof Node))
  throw new IllegalArgumentException();
    else {
  double theta1 = ((Node) obj1).getTheta();
  double theta2 = ((Node) obj2).getTheta();
  if (Math.abs(theta1 - theta2) < 1e-10)
      return 0;
  else if (theta1 < theta2)
      return -1;
  else
      return 1;
    }
}

private class NodeCost {
    Node n;
    double cost;
    boolean marked;
    NodeCost(Node n, double cost) {
  this.n = n;
  this.cost = cost;
  marked = false;
    }
    public Node getNode() {
  return n;
    }
    public double getCost() {
  return cost;
    }
    public void markTrue() {
  marked = true;
    }
    public boolean isMarked() {
  return marked;
    }
```

```
}

// on problem 13 of HVRP, node 42 can't be inserted
public void bestFitInitSolution() {
    Vehicle[] v = getVehicles();
    Arrays.sort(v); // sort in the descending order of vehicle
    // capacity
    //sol = new HVRPSolution(v);
    VRPGraph g = GraphFactory.getGraph();
    Node[] nodes = new Node[g.size() - 1];
    System.arraycopy(g.nodes, 1, nodes, 0, g.size() - 1);
    Arrays.sort(nodes); // sort customer in the descending
   order
    // of demand

    ArrayList nodeList = new ArrayList(nodes.length);
    VehicleType[] fleet = g.getFleet();
    Node depot = g.getDepot();
    Arrays.sort(fleet); // make sure that from largest to
   smallest
    if (fleet[0].getCapacity() < fleet[1].getCapacity()) {
  logger.error("fleet soring is false");
  System.exit(1);
    }
    for (int i = 0; i < nodes.length; i++) {
  double cost = 0;
  for (int j = 0; j < fleet.length; j++) {
      if (fleet[j].getCapacity() >= nodes[i].getDemand()) {
    cost = fleet[j].getPenalty() * g.distance(depot, nodes[i]);
    break;
      }
  }
  nodeList.add(new NodeCost(nodes[i], cost));
    }

    ArrayList vehicleList = new ArrayList(v.length);
    for (int i = 0; i < v.length; i++)
  vehicleList.add(v[i]);

    while (!nodeList.isEmpty()) {
  Vehicle vk = null;
  // we fill one vehicle at a time
  Node current = ((NodeCost) nodeList.remove(0)).getNode();
  for (int i = 0; i < v.length; i++) {
      if (v[i].getFreeLoad() >= current.getDemand()) {
```

```
  vk = v[i];
  break;
    }
}
// vk is the first vehicle that can serve current customer
if (vk == null)
    logger.error(
      "customer "
      + current
      + "  can't be served by any vehicle!");
//else logger.info("serving customer "+ current);
if (vk.getCapacity() < current.getDemand()) {
    logger.error("fatal error in bestFitInitSolution!!!");
    System.exit(1);
} else {
    // NT: if later the total load can be fit into a smaller
    // vehicle
    // we should remove all the customers
    vk.add(current);
    if (vk.getFreeLoad() == 0) {
  //assign this node to vehicle k and continue
  // make sure it really remove this vehicle
  //vehicleList.remove(vk);
  continue;
    } else {
  // try to add more nodes if there are positive saving
  // finally if this vehicle is not fully loaded, can we
  // move
  // to a smaller vehicle
  NodeCost[] remainOrders =
      (NodeCost[]) nodeList.toArray(new NodeCost[0]);
  for (int i = 0; i < remainOrders.length; i++) {
      Node next = remainOrders[i].getNode();
      if (next.getDemand() <= vk.getFreeLoad()) {
    double saving =
        remainOrders[i].getCost()
        - vk.getLeastCost(next);
    if (saving > 1e-10) {
        //logger.info(
        //  "bestFitInitSolution get a positive saving
        // ="
        //    + saving);
        vk.leastCostInsertion(next);
        remainOrders[i].markTrue();
    }
```

```
            }
        }
        // these customers are already served so we delete them
        for (int i = 0; i < remainOrders.length; i++) {
            if (remainOrders[i].isMarked())
          nodeList.remove(remainOrders[i]);
        }
        //check if we can move this load to a smaller vehicle

        for (int i = 0; i < v.length; i++) {
            if (v[i].getCapacity() < vk.getCapacity()
              && vk.getLoad() <= v[i].getFreeLoad()) {
          vk.transferNode(v[i]);
          break;
              }
        }
          }
      }

        }
        checkSolution();
        //buildNBList();
    }

    //  public void bestFitInitSolution() {
    //      Vehicle[] v = getVehicles();
    //      Arrays.sort(v); // sort in the descending order of
        vehicle
    //      // capacity
    //      //sol = new HVRPSolution(v);
    //      VRPGraph g = GraphFactory.getGraph();
    //      Node[] nodes = new Node[g.size() - 1];
    //      System.arraycopy(g.nodes, 1, nodes, 0, g.size() - 1);
    //      Arrays.sort(nodes); // sort customer in the descending
        order
    //      // of demand
    //
    //      ArrayList nodeList = new ArrayList(nodes.length);
    //      VehicleType[] fleet = g.getFleet();
    //      Node depot = g.getDepot();
    //      Arrays.sort(fleet); // make sure that from largest to
        smallest
    //      if (fleet[0].getCapacity() < fleet[1].getCapacity()) {
    //        logger.error("fleet soring is false");
    //        System.exit(1);
```

```
//      }
//      for (int i = 0; i < nodes.length; i++) {
//        double cost = 0;
//        for (int j = 0; j < fleet.length; j++) {
//        if (fleet[j].getCapacity() >= nodes[i].getDemand()) {
//          cost = fleet[j].getPenalty() * g.distance(depot,
    nodes[i]);
//          break;
//        }
//        }
//        nodeList.add(new NodeCost(nodes[i], cost));
//      }
//
//      ArrayList vehicleList = new ArrayList(v.length);
//      for (int i = 0; i < v.length; i++)
//        vehicleList.add(v[i]);
//
//      while (!vehicleList.isEmpty()) {
//        Vehicle vk = (Vehicle) vehicleList.get(0);
//        // we fill one vehicle at a time
//        Node current = ((NodeCost) nodeList.remove(0)).getNode
    ();
//        if (vk.getCapacity() < current.getDemand()) {
//        logger.error("fatal error in bestFitInitSolution!!!");
//        System.exit(1);
//        } else {
//        // NT: if later the total load can be fit into a
    smaller
//        // vehicle
//        // we should remove all the customers
//        vk.add(current);
//        if (vk.getCapacity() == current.getDemand()) {
//          //assign this node to vehicle k and continue
//          // make sure it really remove this vehicle
//          vehicleList.remove(vk);
//          continue;
//        } else {
//          // try to add more nodes if there are positive
    saving
//          // finally if this vehicle is not fully loaded, can
    we
//          // move
//          // to a smaller vehicle
//          NodeCost[] remainOrders =
//          (NodeCost[]) nodeList.toArray(new NodeCost[0]);
```

```
//          for (int i = 0; i < remainOrders.length; i++) {
//          Node next = remainOrders[i].getNode();
//          if (next.getDemand() <= vk.getFreeLoad()) {
//            double saving =
//            remainOrders[i].getCost()
//            - vk.getLeastCost(next);
//            if (saving > 1e-10) {
//            //logger.info(
//            // "bestFitInitSolution get a positive saving ="
//            // + saving);
//            vk.leastCostInsertion(next);
//            remainOrders[i].markTrue();
//            }
//          }
//          }
//          // these customers are already served so we delete
    them
//          for (int i = 0; i < remainOrders.length; i++) {
//          if (remainOrders[i].isMarked())
//            nodeList.remove(remainOrders[i]);
//          }
//          //check if we can move this load to a smaller
    vehicle
//          boolean transfered = false;
//          Vehicle[] veh =
//          (Vehicle[]) vehicleList.toArray(new Vehicle[0]);
//          for (int i = 1; i < veh.length; i++) {
//          logger.assertLog(
//            veh[i].getLoad() == 0,
//            "Vehicle should be empty");
//          if (vk.getLoad() <= veh[i].getCapacity()) {
//            vk.transferNode(veh[i]);
//            vehicleList.remove(veh[i]);
//            transfered = true;
//            break;
//          }
//          }
//          if (!transfered)
//          vehicleList.remove(vk);
//        }
//        }
//
//      }
//      checkSolution();
//      //buildNBList();
```

```
//     }
public void checkSolution() {
    int expectedSize = GraphFactory.getGraph().size() - 1;
    //excluding the depot
    int totalSize = 0;
    for (int i = 0; i < vehicles.length; i++)
  totalSize += vehicles[i].length();
    checkMissingNode();
    logger.assertLog(
      totalSize == expectedSize,
      "the solution misses some customer");
    checkDistance();
    //checkLoad();
}

    private void checkMissingNode() {
  VRPGraph g = GraphFactory.getGraph();
  for(int i=1;i<g.size();i++)
      if(missingNode(g.nodes[i])) {
    logger.error("node "+ i+" is missing from the solution!");
    System.exit(1);
      }
    }

    private boolean missingNode(Node n) {
  boolean missing = true;
  for(Vehicle v : vehicles) {
      if(v.contains(n)){
    missing = false;
    break;
      }
  }
  return missing;
    }
    public void checkSolution(String where) {
    int expectedSize = GraphFactory.getGraph().size() - 1;
    //excluding the depot
    int totalSize = 0;
    for (int i = 0; i < vehicles.length; i++)
  totalSize += vehicles[i].length();
    logger.assertLog(
      totalSize == expectedSize,"["+where+"]"+
      "the solution misses some customer");
    checkDistance();
    //checkLoad();
```

```
}
private void checkDistance() {
    for (int i = 0; i < vehicles.length; i++)
  vehicles[i].checkSolution();
}

    public void CWSeqSaving(double lamda) {

  VRPGraph g = GraphFactory.getGraph();
  Node depot = g.getDepot();
  Node[] nodes = new Node[g.size() - 1];
  System.arraycopy(g.nodes, 1, nodes, 0, g.size() - 1);
  // sort the customers according to their rank decreasingly
  Arrays.sort(nodes);
  Vehicle[] v = vehicles;
  // sort the vehicle according to its capacity decreasingly
  Arrays.sort(v);

  VehicleType[] fleet = g.getFleet();
  Arrays.sort(fleet);

  ArrayList nodeList = new ArrayList(nodes.length);
  for (int i = 0; i < nodes.length; i++)
      nodeList.add(nodes[i]);

  ArrayList vehicleList = new ArrayList(v.length);
  for (int i = 0; i < v.length; i++)
      vehicleList.add(v[i]);

  // build the save list
  ArrayList savingList = new ArrayList(g.size() * g.size() * v.
   length);

  for (int i = 0; i < nodes.length; i++) {
      for (int j = i + 1; j < nodes.length; j++) {
      //for (int j = 0; j < nodes.length; j++) {
    if(i==j) continue;
    for (int ki = 0; ki < fleet.length; ki++) {
        if (fleet[ki].getCapacity() < nodes[i].getDemand())
      break;
        else {
      for (int kj = 0; kj < fleet.length; kj++) {
          if (fleet[kj].getCapacity() < nodes[j].getDemand()
            || fleet[ki].getCapacity()
           < nodes[i].getDemand()
```

```
            + nodes[j].getDemand())
        break;
          else {
        double saving =
            fleet[ki].getPenalty()
            * (g.distance(nodes[i], depot)
              - g.distance(nodes[j], nodes[i])
              - g.distance(depot, nodes[j]))
            + fleet[kj].getPenalty()
            * (g.distance(depot, nodes[j])
              + g.distance(nodes[j], depot));

        if (saving > 1e-5)
            savingList.add(
              new HVRPSaving(
            nodes[i],
            nodes[j],
            fleet[ki],
            fleet[kj],
            fleet[ki],
            saving));
          }
      }
        }
    }
      }
}

HVRPSaving[] savings =
    (HVRPSaving[]) savingList.toArray(new HVRPSaving[0]);
Arrays.sort(savings);
savingList.clear();
for (int i = savings.length - 1; i >= 0; i--)
    savingList.add(savings[i]);

boolean[] flag = new boolean[g.size()];
for (int i = 0; i < flag.length; i++)
    flag[i] = false;

while (!nodeList.isEmpty()) {

    if (vehicleList.isEmpty()) {
  logger.error("not enough vehicle in Clarke and Wright
 Seqential Saving");
```

```
break;
  }

  // get the customer w ith highest rank

  else {
Node current = (Node) nodeList.remove(0);
Vehicle veh = (Vehicle) vehicleList.remove(0);
if (veh.getCapacity() < current.getDemand()) {
    //          logger.error(
    //            "customer "
    //              + current.getId()
    //              + " can't be added to any vehicle");
    // put it back and quit since we sort the vehicles by
    // capacity
    nodeList.add(nodeList.size(), current);
    break;
} else {
    //logger.info("adding customer " + current.getId());
    //            logger.assertLog(
    //              current.getId() != 0,
    //              "can't adding depot shoot!");

    veh.add(current);
    flag[current.getId()] = true;
    Node first = veh.getFirst();
    Node last = veh.getLast();
    ListIterator lit = savingList.listIterator();
    while (lit.hasNext()) {
  HVRPSaving asaving = (HVRPSaving) lit.next();
  if (flag[asaving.j.getId()] == true)
      lit.remove();
  else if (
    !(first.equals(asaving.i)
        || last.equals(asaving.i)))
      continue;
  else {
      if (veh.isTypeOf(asaving.vi)
        && veh.getFreeLoad() >= asaving.j.getDemand()) {
    if (first.equals(asaving.i)) {
        //we can insert j before current
        veh.addFirst(asaving.j);
        first = asaving.j;
        nodeList.remove(first);
        flag[first.getId()] = true;
```

```
            continue;
        } else if (last.equals(asaving.i)) {
            veh.append(asaving.j);
            last = asaving.j;
            nodeList.remove(last);
            flag[last.getId()] = true;
            continue;
        } else
            logger.info(
              "this really should not happend in CWSeqSaving!")
;
            }
    }
      }

      //try to assgin this vehicle to a smaller one to save
 free
      // load
      ListIterator itr = vehicleList.listIterator();
      boolean transfered = false;
      while (itr.hasNext()) {
    Vehicle nextVehicle = (Vehicle) itr.next();
    if (nextVehicle.getCapacity() < veh.getCapacity()
     && nextVehicle.getCapacity() >= veh.getLoad()) {
        veh.transferNode(nextVehicle);
        itr.remove();
        transfered = true;
        break;
      }
      }
      if (transfered) {
    logger.assertLog(
      veh.isEmpty(),
      "the vehicle should be empty after transferNode");
    logger.assertLog(
      veh.getLoad() == 0,
      "there should be no load after transfer");
    vehicleList.add(0, veh);
      }
  }
    } // end while loop
}
// there are some nodes remaining, so we use cheapest
 insertion
```

```
Node[] remainNodes = (Node[]) nodeList.toArray(new Node[0]);
Arrays.sort(remainNodes);
//ListIterator iter = nodeList.listIterator();
//while (iter.hasNext()) {
for (int i = 0; i < remainNodes.length; i++) {
    this.leastCostInsertNode(remainNodes[i]);
}
this.checkSolution();
logger.info("finish CWSeqSaving");
  }

  public void perturb() {
// we try to free some nodes with smallest ration
// r = demand / insertion_cost
class Ratio implements Comparable {
    Node from, current, to;
    Vehicle v;
    double value;
    public Ratio(
      Node from,
      Node current,
      Node to,
      Vehicle v,
      double value) {
  this.from = from;
  this.current = current;
  this.to = to;
  this.v = v;
  this.value = value;
      }

    public int compareTo(Object obj) {
  if (value < ((Ratio) obj).value)
      return -1;
  else if (value > ((Ratio) obj).value)
      return 1;
  else
      return 0;
    }

    public String toString() {
  return from + "->" + current + "->" + to + "(" + value + ")
 ";
    }
}
```

```
VRPGraph g = GraphFactory.getGraph();
Node[] nodes = new Node[g.size() - 1];
System.arraycopy(g.nodes, 1, nodes, 0, g.size() - 1);

Ratio[] ratio = new Ratio[nodes.length];
for (int i = 0; i < vehicles.length; i++) {
    Enumeration e = vehicles[i].elements();
    while (e.hasMoreElements()) {
  ListNode current = (ListNode) e.nextElement();
  if (current.getId() > 0) {
      Node curr = (Node) current.getData();
      Node prev =
    (Node) ((ListNode) current.getPrevious()).getData();
      Node next = (Node) ((ListNode) current.getNext()).
 getData();
      double r =
    curr.getDemand()
    / ((g.distance(prev, curr)
          + g.distance(curr, next)
          - g.distance(prev, next))
      * vehicles[i].getPenalty());
      ratio[curr.getId() - 1] =
    new Ratio(prev, curr, next, vehicles[i], r);
  }
    }
}

Arrays.sort(ratio);
int totalPurturbation = Math.min((int) g.size() / 5, 20);
logger.assertLog(totalPurturbation > 0, "purturbation number
 is wrong");
int numPurb = 0;
for (int i = 0; i < g.size(); i++) {
    if(numPurb>=totalPurturbation) break;
    Vehicle v = ratio[i].v;
    v.remove(ratio[i].current);
    double leastCost = 1e10;
    Vehicle insVeh = null;
    ListNode insNode = null;
  for (int k = 0; k < vehicles.length; k++) {
      if (vehicles[k].getFreeLoad()
        >= ratio[i].current.getDemand()) {
    Enumeration e = vehicles[k].elements();
    while (e.hasMoreElements()) {
        ListNode listnode = (ListNode) e.nextElement();
```

```
        Node from = (Node) listnode.getData();
        Node to =
      (Node) ((ListNode) listnode.getNext()).getData();
        if (from.equals(ratio[i].from)
          && to.equals(ratio[i].to))
      continue; //this is the old place
        else {
      double cost =
          g.distance(from, ratio[i].current)
          + g.distance(ratio[i].current, to)
          - g.distance(from, to);
      cost *= vehicles[k].getPenalty();
      if (cost < leastCost) {
          leastCost = cost;
          insVeh = vehicles[k];
          insNode = listnode;
      }
        }
    }
        }
  }

    if (insVeh == null) {
  // usually this happens when we remove the only customer
 from a
  // vehicle
  //logger.error("cannot purturn a solution");
  v.add(ratio[i].current);
  //System.exit(1);
    } else { // do the purturbation
  //logger.info("reinsert node" + ratio[i].current +" after
 "+
  // insNode.getId());
  insVeh.addAfter(insNode, ratio[i].current);
  numPurb++;
    }
}
  }

  public void ruinRecreate() {
VRPGraph g = GraphFactory.getGraph();
final int p = Math.min(g.size()/5,8);
int nodeToPerturb = rand.nextInt(1,g.size());
int [] nodeToDelete = new int[p+1];
int maxIndex = Math.min(p,g.NBCount[nodeToPerturb]);
```

```
int count=0;
nodeToDelete[count++] = nodeToPerturb;
for(int i=0;i<maxIndex;i++)
    if(g.NBList[nodeToPerturb][i]!=0) // not the depot
  nodeToDelete[count++] = g.NBList[nodeToPerturb][i];
for(int i=0;i<count;i++){
    logger.assertLog(nodeToDelete[i]!=0,"depot can't be
 deleted!");
    //logger.info("deleting node "+ nodeToDelete[i]);
    deleteNode(nodeToDelete[i]);
}

clean();
// we want to insert node with large demand first,
 decreasing order
class NodeSortByDemand implements Comparable{
    int id, q;
    public NodeSortByDemand(int id,int q) {
  this.id = id; this.q = q;
    }
    public int compareTo(Object o) {
  NodeSortByDemand node = (NodeSortByDemand) o;
  if(this.q > node.q) return -1;
  else if(this.q==node.q) return 0;
  else return 1;
    }
}
NodeSortByDemand [] dnode = new NodeSortByDemand[count];
for(int i=0;i<count;i++) dnode[i] = new NodeSortByDemand(
 nodeToDelete[i], g.nodes[nodeToDelete[i]].getDemand());
Arrays.sort(dnode);
for(int i=0; i<count;i++) {
    //insertNode(nodeToDelete[i]);
    insertNode(dnode[i].id);
    //logger.info("inserting node "+ dnode[i].id +" with
 demand ="+ dnode[i].q);
}
checkSolution();
//logger.info("done R&R");
  }

  private void deleteNode(int nid) {
VRPGraph g = GraphFactory.getGraph();
Node node = g.nodes[nid];
for(Vehicle v : vehicles) {
```

```
        if(v.contains(node))
      v.remove(node);

    }
      }

      private void insertNode(int nid) {
    Node n = GraphFactory.getGraph().nodes[nid];
    //logger.info("tryint to insert node "+ n +" with demand "+ n
     .getDemand());
    Vehicle insVehicle = null;
    double minCost = 1e5;
    for(Vehicle v : vehicles) {
        double cost = v.getLeastCost(n);
        //logger.info("trying vehicle "+ v +" with cost ="+ cost)
     ;
        if(cost < minCost) {
      minCost = cost;
      insVehicle = v;
        }
    }
    if(insVehicle==null) {
        logger.info("can't insert node "+nid +" with demand "+ n.
     getDemand());
        // try to swap a node in the tour
        boolean done = false;
        while(!done) {

        }
    }
    else insVehicle.add(n);
      }

      public void accept(Visitor visitor) {
    visitor.visit(this);
    for(Vehicle v: vehicles)
       if(!v.isEmpty()) v.accept(visitor);
      }
}
```

D.2 HVRPInstance.java

```
/*
 * Created on 2003-10-27
 *
```

```
 * To change the template for this generated file go to Window
   - Preferences -
 * Java - Code Generation - Code and Comments
 */
package hvrp;
import java.util.*;
import org.apache.log4j.*;
/**
 * @author lify
 *
 * To change the template for this generated type comment go to
    Window -
 * Preferences - Java - Code Generation - Code and Comments
 */
public class HVRPInstance implements Cloneable, Comparator {

  private VRPGraph g;
  //static ListNode nodes[];
  private HVRPSolution sol;
  static Logger logger = Logger.getLogger(HVRPInstance.class);
  public HVRPInstance(VRPGraph g) {
    this.g = g;
    sol = new HVRPSolution(g.getVehicles());
    // leastCostInitSolution();
  }

  //    public ListNode[] getNodes() {
  //  return nodes;
  //    }

  public void initSolution() {
    Vehicle[] v = sol.getVehicles();
    //sol = new HVRPSolution(v);
    Node[] nodes = new Node[g.size() - 1];
    System.arraycopy(g.nodes, 1, nodes, 0, g.size() - 1);
    Arrays.sort(nodes);
    //Arrays.sort(v);
    for (int i = 0; i < nodes.length; i++) {
      Arrays.sort(v);
      for (int k = 0; k < v.length; k++) {
        if (v[k].leastCostInsertion(nodes[i]))
          break;
      }
    }
    buildNBList();
```

```
}

public void leastCostInitSolution() {
  Vehicle[] v = sol.getVehicles();
  //sol = new HVRPSolution(v);
  Node[] nodes = new Node[g.size() - 1];
  System.arraycopy(g.nodes, 1, nodes, 0, g.size() - 1);
  Arrays.sort(nodes);
  for (int i = 0; i < nodes.length; i++) {
    double insCost = 1e8;
    Vehicle insVeh = null;
    for (int k = 0; k < v.length; k++) {
      double theCost = v[k].getLeastCost(nodes[i]);
      if (theCost < insCost) {
        insCost = theCost;
        insVeh = v[k];
      }
    }
    if (insVeh != null)
      insVeh.leastCostInsertion(nodes[i]);
    else {
      logger.error(
        "cannot insert node "
          + nodes[i]
          + " with demand="
          + nodes[i].getDemand());
      logger.error(
        "in leastcost initsolution This should not happend");
    }
  }
  buildNBList();
}

public void leastCostInitSolution(HVRPSolution sol) {
  Vehicle[] v = sol.getVehicles();
  //sol = new HVRPSolution(v);
  Node[] nodes = new Node[g.size() - 1];
  System.arraycopy(g.nodes, 1, nodes, 0, g.size() - 1);
  //Arrays.sort(nodes);
  for (int i = 0; i < nodes.length; i++) {
    double insCost = 1e8;
    Vehicle insVeh = null;
    for (int k = 0; k < v.length; k++) {
      double theCost = v[k].getLeastCost(nodes[i]);
      if (theCost < insCost) {
```

```
        insCost = theCost;
        insVeh = v[k];
      }
    }
    if (insVeh != null)
      insVeh.leastCostInsertion(nodes[i]);
    else
      logger.error("This should not happend");
  }
  buildNBList();
}

public void sweepInitSolution(double initTheta) {
  Vehicle[] v = sol.getVehicles();
  Node depot = g.getDepot();
  Node[] nodes = new Node[g.size() - 1];
  System.arraycopy(g.nodes, 1, nodes, 0, g.size() - 1);
  //SweepNode [] snodes = new SweepNode[nodes.length];
  for (int i = 0; i < nodes.length; i++)
    nodes[i].setTheta(depot, initTheta);
  Arrays.sort(nodes, this);
  Arrays.sort(v);
  for (int nid = 0; nid < nodes.length; nid++) {
    int vid = -1;
    double leastCost = 1e10;
    for (int k = 0; k < v.length; k++) {
      double cost = v[k].getLeastCost(nodes[nid]);
      if (cost < leastCost) {
        leastCost = cost;
        vid = k;
      }
    }
    v[vid].leastCostInsertion(nodes[nid]);
  }
  //    for(int i=0;i<v.length;i++) {
  //        // we are going to fill up this vehicle
  //        for(;nid<nodes.length;nid++) {
  //      if(!v[i].leastCostInsertion(nodes[nid]))
  //          break;
  //        }
  //    }
  //    // we may have some customers left
  //    while(nid<nodes.length) {
  //      Node n = nodes[nid];
  //      logger.info("we have leftover " + n + ",demand="+n.
```

```
 getDemand() );
  //        int vid = -1;
  //        double leastCost = 1e10;
  //        for(int i=0;i<v.length;i++) {
  //          double cost = v[i].getLeastCost(n);
  //          if(cost<leastCost) {
  //            leastCost = cost;
  //            vid = i;
  //          }
  //        }
  //        if(vid>=0) {
  //          v[vid].leastCostInsertion(n);
  //          nid++;
  //        }
  //        else throw new RuntimeException("we are stuck here
 ");
  //    }
}
public void sweepInitSolution(HVRPSolution sol, double
 initTheta) {
  Vehicle[] v = sol.getVehicles();
  Node depot = g.getDepot();
  Node[] nodes = new Node[g.size() - 1];
  System.arraycopy(g.nodes, 1, nodes, 0, g.size() - 1);
  //SweepNode [] snodes = new SweepNode[nodes.length];
  for (int i = 0; i < nodes.length; i++)
    nodes[i].setTheta(depot, initTheta);
  Arrays.sort(nodes, this);
  Arrays.sort(v);
  for (int nid = 0; nid < nodes.length; nid++) {
    int vid = -1;
    double leastCost = 1e10;
    for (int k = 0; k < v.length; k++) {
      double cost = v[k].getLeastCost(nodes[nid]);
      if (cost < leastCost) {
        leastCost = cost;
        vid = k;
      }
    }
    if (vid == -1) {
      logger.error(
        "can't insert node "
          + nodes[nid]
          + " into the solution ,demand="
          + nodes[nid].getDemand());
```

```
        logger.info(sol);
        System.exit(1);
      } else
        v[vid].leastCostInsertion(nodes[nid]);
    }
  }
  public int compare(Object obj1, Object obj2) {
    if (!(obj1 instanceof Node && obj2 instanceof Node))
      throw new IllegalArgumentException();
    else {
      double theta1 = ((Node) obj1).getTheta();
      double theta2 = ((Node) obj2).getTheta();
      if (Math.abs(theta1 - theta2) < 1e-10)
        return 0;
      else if (theta1 < theta2)
        return -1;
      else
        return 1;
    }
  }
  private void buildNBList() {
//    nodes = sol.toArray();
//    Arrays.sort(nodes);
  }

  public String toString() {
    return sol.toString();
  }

  public HVRPSolution getSolution() {
    return sol;
  }

  public void twOpt() {
    sol.twOpt();
  }

  public void onePointMove() {
    sol.onePointMove();
  }

  /*
   * (non-Javadoc)
   *
   * @see java.lang.Object#clone()
```

```
 */
protected Object clone() throws CloneNotSupportedException {
  // TODO Auto-generated method stub
  HVRPInstance hvrp = (HVRPInstance) super.clone();
  hvrp.sol = (HVRPSolution) sol.clone();
  return hvrp;
}

/**
 *
 */
public void twoPointMove() {
  sol.twoPointMove();
}

/**
 *
 */
public void clean() {
  sol.clean();
}

public HVRPSolution CWSolver() {
  HVRPSolution bestSol = null;
  double minLen = 1e10;
  for(int i=0;i<10;i++) {
    g.setParameter(0.1*i);
    HVRPSolution sol = new HVRPSolution(g.getVehicles());
    sol.CWSeqSaving(1.0);
    logger.info("alpha="+0.1*i+",distance="+sol.getDistance()
 );
    sol.RTR();
    logger.info("after RTR, distance=" + sol.getDistance());
    if(sol.getDistance() < minLen) {
      bestSol = sol;
    }
  }
  return bestSol;
}

public HVRPSolution SweepSolver() {
  int numTrial = 100;
  HVRPSolution [] best5 = new HVRPSolution[5];
  HVRPSolution bestSol = null;
  double minLen = 1e10;
```

```
  for (int i = 0; i < numTrial; i++) {
    double theta = 2 * Math.PI * i / numTrial;
    sol = new HVRPSolution(g.getVehicles());
    try {
        sol.sweepInitSolution(theta);
          //logger.info("sweep distance=" + sol.getDistance()
);
        } catch (IllegalArgumentException e) {
          continue;
         }
    for (int k = 0; k < 5; k++) {
            if (best5[k] != null) {
              if (best5[k].getDistance() > sol.getDistance())
{
                for (int j = 4; j > k; j--)
                  best5[j] = best5[j - 1];
                best5[k] = sol;
                //break;
              }
            }else {
              best5[k] = sol;
              //break;
            }
          }
  }
  for(int i=0;i<5;i++) {
    HVRPSolution sol = best5[i];
    sol.checkSolution();
    logger.info("sweep " + i + " distance = " + sol.
getDistance());
    sol.RTR();
    logger.info("sweep " + i + " RTR distance=" + sol.
getDistance());
    if(sol.getDistance() < minLen) {
      minLen = sol.getDistance();
      try {
        bestSol = (HVRPSolution)sol.clone();
      }catch(CloneNotSupportedException e) { logger.error(e);
}
    }
  }
  return bestSol;
}
```

```
private HVRPSolution bestFive(HVRPSolution sol) {
  int repetition = 5;
  HVRPSolution copy[] = new HVRPSolution[repetition];
  for(int i=0;i<repetition;i++) copy[i] = sol;
  double minLen = 1e10;
  HVRPSolution best = null;
  for(int i=0;i<repetition;i++) {
    HVRPSolution rtr = sol.RTR();
    if(rtr.getDistance()<minLen) {
      minLen = rtr.getDistance();
      best = rtr;
    }
  }
  return best;
}

public HVRPSolution solve() {
  final int INIT =0;
  final int LS = 1;
  final int BF = 2;
  final int SW = 3;
  final int CW = 4;
  final int MAX = 5;
  int flag = -1;

  HVRPSolution bestSol = null;
  double minLen = 1e10;
  int numTrial = 100;
  HVRPSolution[] best5 = new HVRPSolution[5];

  HVRPSolution sol = new HVRPSolution(g.getVehicles());
  sol.maxSolution();
  logger.info("verifying maxsolution");
  sol.checkSolution();
  logger.info("maxSolution distance="+sol.getDistance());

  HVRPSolution RTRmax = bestFive(sol);
  RTRmax.checkSolution();
  logger.info("verifying maxRTRsolution");
  RTRmax.checkSolution();
  if(RTRmax.getDistance()< minLen) {
      bestSol = RTRmax;
      minLen = RTRmax.getDistance();
      flag = MAX;
  }
```

```
    logger.info("maxRTR distance="+RTRmax.getDistance());


    long start = System.currentTimeMillis();
    for (int i = 0; i < numTrial; i++) {
      double theta = 2 * Math.PI * i / numTrial;
      sol = new HVRPSolution(g.getVehicles());
      try {
        sol.sweepInitSolution(theta);
        //logger.info("sweep distance=" + sol.getDistance());
      } catch (IllegalArgumentException e) {
        logger.info("infeasible solution for " + i);
        continue;
      }
      for (int k = 0; k < 5; k++) {
        if (best5[k] != null) {
          if (best5[k].getDistance() > sol.getDistance()) {
            for (int j = 4; j > k; j--)
              best5[j] = best5[j - 1];
            best5[k] = sol;
            //break;
          }
        } else {
          best5[k] = sol;
          //break;
        }
      }
    }
    long end = System.currentTimeMillis();
//    logger.info(
//      "the running time for "
//        + numTrial
//        + " soluition = "
//        + (end - start) / 1e3
//        + " seconds");
    HVRPSolution result = null;
    for (int i = 0; i < 5; i++) {
      //HVRPSolution sol = best5[i];
      sol = best5[i];
      logger.info("check sweep solution " + i);
      sol.checkSolution();
      //logger.info("sweep "+i+",tour length=" + sol.
   getDistance());
      //HVRPSolution result = sol.RTR();
      result = sol.RTR();
```

```
    logger.info("sweep" +i+" RTR distance=" + result.
 getDistance());
    if (result.getDistance() < minLen) {
      bestSol = result;
      minLen = bestSol.getDistance();
      flag = SW;
    }
  }

  sol = new HVRPSolution(g.getVehicles());
  sol.CWSeqSaving(1.0);

  HVRPSolution saving = this.bestFive(sol);
  logger.info("CW solution=" + saving.getDistance());
  if(saving.getDistance()<minLen) {
    minLen = saving.getDistance();
    bestSol = saving;
    flag = CW;
  }

  switch(flag) {
    case INIT:
      logger.info("best solution from INIT");
      break;
    case LS:
      logger.info("best solution from LS");
      break;
    case BF:
      logger.info("best solution from BF");
      break;
    case SW:
      logger.info("best solution from SW");
      break;
    case CW:
      logger.info("best solution from CW");
      break;
    case MAX :
      logger.info("best solution from MAX");
      break;
    default:
      logger.error("Fator error");
      break;
  }
  return bestSol;
}
```

```
//     public HVRPSolution RTR() throws
 CloneNotSupportedException {
//   int counter = 1;
//   int I = 30;
//   sol.clean();
//   HVRPSolution bestSol = (HVRPSolution)sol.clone();
//   double bestLength = bestSol.getDistance();
//   Record r = new Record(bestLength);
//   r.setCurrent(bestLength);
//
//   while(true) {
//       boolean improved = false;
//       for(int i=0;i<I;i++) {
//     if( !sol.onePointMoveRTR(r) & !sol.twoPointMoveRTR(r) &
// !sol.twOptRTR(r)) {
//         logger.info("no movement in loop I. Quit loop I");
//         break;
//     }
//     else {
//         if(sol.getDistance() < bestSol.getDistance()) {
//       improved = true;
//       bestSol = (HVRPSolution) sol.clone();
//       r.setCurrent(sol.getDistance());
//         }
//     }
//       }
//       sol.clean();
//       double currentLength = sol.getDistance();
//       if(currentLength<bestSol.getDistance() - 1e-10) {
//     logger.info("find new best solution");
//     bestSol = (HVRPSolution) sol.clone();
//     r.setRecord(currentLength);
//     //r.setCurrent(currentLength);
//     counter = 0;
//       }
//
//       r.setCurrent(currentLength);
//       logger.info("counter=" + counter +" best solution ="
 +
// bestSol.getDistance() + "current solution="
//         + currentLength);
//       counter++;
//       if(counter>5) break;
//   }
//   return bestSol;
```

```
  //    }

  public HVRPSolution RTR() {
    return sol.RTR();
  }

//  public HVRPSolution CWSaving() {
//    Node[] nodes = new Node[g.size() - 1];
//    System.arraycopy(g.nodes, 1, nodes, 0, g.size() - 1);
//    VehicleType[] fleet = g.getFleet();
//    Arrays.sort(fleet);
//    //Arrays.sort(fleet);
//    Node depot = g.getDepot();
//    // in the beginning each node use a single vehicle, which
    of course
//    // with
//    // the smallest penalty
//    Vehicle[] v = new Vehicle[nodes.length];
//    for (int i = 0; i < v.length; i++) {
//      v[i] = new Vehicle(fleet[0], depot);
//      v[i].add(nodes[i]);
//    }
//
//    ArrayList savingList =
//      new ArrayList(
//        g.size() * g.size() * (int) Math.pow(fleet.length, 3)
   );
//    for (int i = 0; i < nodes.length; i++) {
//      for (int j = i + 1; j < nodes.length; j++) {
//        for (int ki = 0; ki < fleet.length; ki++) {
//          for (int kj = 0; kj < fleet.length; kj++) {
//            for (int kk = 0; kk < fleet.length; kk++) {
//              double saving =
//                g.distance(nodes[i], depot)
//                  * fleet[ki].getPenalty()
//                  + g.distance(nodes[j], depot)
//                    * fleet[kj].getPenalty()
//                  - g.distance(nodes[i], nodes[j])
//                    * fleet[kk].getPenalty();
//              if (saving > 0D) {
//                savingList.add(
//                  new HVRPSaving(
//                    i,
//                    j,
//                    fleet[ki],
```

```
//                      fleet[kj],
//                      fleet[kk],
//                      saving));
//                }
//              }
//            }
//          }
//        }
//
//    logger.info("there are " + savingList.size() + " savings
    in the list");
//    HVRPSaving[] list =
//      (HVRPSaving[]) savingList.toArray(new HVRPSaving[0]);
//    Arrays.sort(list);
//    for (int i = 0; i < list.length; i++) {
//      HVRPSaving saving = list[i];
//      Node from = nodes[saving.i];
//      Node to = nodes[saving.j];
//
//    }
//    return null;
//  }
}
```

D.3 HVRPSaving.java

```
/*
 * Created on 2003-11-6
 *
 * To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and
   Comments
 */
package hvrp;

/**
 * @author lify
 *
 * To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Generation - Code and
   Comments
 */
public class HVRPSaving implements Comparable {
  Node i,j;
```

```
  VehicleType vi,vj,vk;
  private double saving;



  /**
   * @param vi
   * @param vj
   * @param vk
   * @param saving
   */

  // initially node i from vehicletype vi
  // node j from vehicletype vj
  // after saving, they are combined in vehicletype vk
  public HVRPSaving(Node i,Node  j ,VehicleType vi, VehicleType
    vj, VehicleType vk, double saving) {
    this.i = i;
    this.j = j;
    this.vi = vi;
    this.vj = vj;
    this.vk = vk;
    this.saving = saving;
  }

  public int compareTo(Object other) {
    if(!(other instanceof HVRPSaving)) throw new
   IllegalArgumentException();
    else {
      double osaving = ((HVRPSaving) other).saving;
      if(saving<osaving) return -1;
      else if(saving>osaving) return 1;
      else return 0;
    }
  }

  public String toString() {
    return ""+i+"-->"+j+"saving="+saving;
  }
}
```

D.4 SolutionPlot.java

```
package hvrp;
import java.io.*;
```

```
import java.util.*;

public class SolutionPlot {
  private String data;
  PrintWriter pw;

  public SolutionPlot(String output){
    data = output;
    try {
      FileWriter fw = new FileWriter(data,false);
      pw = new PrintWriter(new BufferedWriter(fw));
    } catch(FileNotFoundException e) {}
    catch(IOException ie){}
  }

  public void print(Node n) {
    pw.println(n.x+"\t"+n.y);

  }


  public void print(Tour t){
    Enumeration e = t.elements();
    while (e.hasMoreElements()) {
      ListNode n = (ListNode)e.nextElement();
      print((Node) n.getData());
    }

  }

  public void print(Vehicle v) {
    print(v.getDepot());
    print(v.getTour());
    print(v.getDepot());
    print("\n\n");//required by gnuplot to separate data
  }

  public void print(String s) { pw.print(s); }

    public void print(HVRPSolution s) {
      Vehicle[] v = s.getVehicles();
      for(int i=0;i<v.length;i++) {
        if(!v[i].isEmpty()) print(v[i]);
      }
    }
```

```
    public void print(HVRPInstance ins) {
      print(ins.getSolution());
    }
  public void close() {
    pw.close();
  }
}
```

D.5 SolutionWriter.java

```
package hvrp;
import java.io.*;

public class SolutionWriter extends reflectiveVisitor {

    private PrintWriter pw;

    public SolutionWriter(String fn) {
  try {
      FileWriter fw = new FileWriter(fn,false);
      //System.out.println("the solution writer file is "+fn);
      pw = new PrintWriter(new BufferedWriter(fw));
  } catch(FileNotFoundException e) { System.out.println(e); }
  catch(IOException e) {}
    }

    public void close() {
  pw.close();
    }
    public void visitTour(Tour t) {
  ListNode [] node = t.toArray();
  for(ListNode n : node) {
  //for(int i=0; i<node.length;i++){
      pw.print(n+",");
  }
  pw.println();
    }

    public void visitVehicle(Vehicle v) {
  VehicleType vt = v.getVehicleType();
  Tour t = v.getTour();
  pw.print(vt+"load="+t.getLoad()+",distance="+t.getDistance())
   ;
  pw.println();
```

```
    visitTour(t);
      }
      public void visitHVRPSolution(HVRPSolution sol) {
    pw.println("total distance="+sol.getDistance());
      }

}
```

D.6 Vehicle.java

```
/*
 * Created on 2003-10-15
 *
 * To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and
   Comments
 */
package hvrp;
import java.lang.reflect.*;
import java.util.Enumeration;
import org.apache.log4j.*;

/**
 * @author lify
 *
 * To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Generation - Code and
   Comments
 */
public class Vehicle implements Comparable,Cloneable,Visitable{
  private VehicleType type;
  private Tour tour;
  //private Node depot;
  private static Logger logger =Logger.getLogger(Vehicle.class)
   ;
  static int GVID = 0; // global vehicle ID
  int vid;
  /**
   * @param type
   * @param depot
   */
  public Vehicle(VehicleType type, Node depot) {
    this.type = type;
    tour = new Tour(depot);
    tour.setVehicle(this);
```

```
  vid = GVID++;
  //this.depot = depot;
}
// compare vehicle based on its capacity
public int compareTo(Object o)  {
  if(! (o instanceof Vehicle)) throw new
 IllegalArgumentException();
  double cap = ((Vehicle)o).getScore();
  double thiscap = getScore();
  if(thiscap>cap) return -1; // because we want descending
 order
  else if (thiscap==cap) return 0;
  else return 1;
}

/**
 * @param type
 */
private Vehicle(VehicleType type) {
  logger.warn("this constructor of Vehicle should not be
 called");
  this.type = type;
}
/**
 * @return Returns the depot.
 */
public Node getDepot() {
  return tour.getDepot();
}

/**
 * @param depot The depot to set.
 */
//public void setDepot(VRPNode depot) {
//  this.depot = depot;
//}

public VehicleType getVehicleType() {
    return type;
}

/**
 * @return
 */
public int getCapacity() {
```

```
  return type.getCapacity();
}

public int getFreeLoad() {
  return type.getCapacity() - getLoad();
}


public int getLoad() {
  return tour.getLoad();
}


// this is for cheapest insertion
// but we don't consider distance constraint here
public boolean add(Node n) {
  if( n.getDemand()+getLoad()<=type.getCapacity() ) {
    //load+=n.getDemand();
    tour.leastCostInsertion(n);
    return true;
  }
  else return false;
}

public void append(Node n) {
  //load += n.getDemand();
  tour.append(n);
}

public void addFirst(Node n) {
  tour.addFirst(n);
}
/**
 * @param o
 */
public boolean leastCostInsertion(Node o) {
  if(o.getDemand()+getLoad()<=type.getCapacity()) {
    tour.leastCostInsertion(o);
    //load+=o.getDemand();
    return true;
  }
  else return false;
}

public double getLeastCost(Node o) {
```

```
    if(o.getDemand()+getLoad()<=type.getCapacity())
      return tour.getLestCost(o) * type.getPenalty();
    else return 1e10;
  }
  public double getScore() {
    //return type.getCapacity()/Math.pow(type.getPenalty(),
   FACTOR);
    return type.getCapacity()-getLoad();
    //return type.getCapacity();
  }

  final private double FACTOR = 0D;

  public String toString() {
    String s = type.toString();
    return s+"("+vid+"),load="+getLoad() + ",distance="+
   getDistance()+" ("+tour+")";
  }
  /**
   * @return Returns the tour.
   */
  public Tour getTour() {
    return tour;
  }

  public boolean isEmpty() {
    return tour.length()==0;
  }

  /**
   * @return Returns the distance.
   */
  public double getDistance() {
    return tour.getDistance()*type.getPenalty();
  }

  public void twOpt() {
    tour.twOpt();
  }
  /**
   * n is a node in this vehicle
   */
  public boolean twOpt(ListNode n) {
    return tour.twOpt(n);
  }
```

```
  public boolean OROptMove(ListNode n) {
    return tour.OROptMove(n);
  }

  public boolean OROptMove(ListNode n, Vehicle v) {
    return tour.OROptMove(n,v.getTour());
  }
  /**
   * n is a node in this vehicle
   * v is another vehicle
   */
  public boolean twOpt(ListNode n, Vehicle v) {
    return tour.twOpt(n,v.getTour());
  }
  /**
   *
   */
  public void onePointMove() {
    tour.onePointMove();
  }

  /**
   * @return
   */
  public double getPenalty() {
    return type.getPenalty();
  }

  /**
   * @param load The load to set.
   */
//  public void setLoad(int load) {
//    this.load = load;
//  }

  public boolean onePointMove(ListNode n) {
    return tour.onePointMove(n);
  }

  public boolean onePointMove(ListNode n, Vehicle v) {
    return tour.onePointMove(n,v.getTour());
  }
  /* (non-Javadoc)
   * @see java.lang.Object#clone()
```

```
 */
protected Object clone() throws CloneNotSupportedException {
  // TODO Auto-generated method stub
  Vehicle copy = (Vehicle) super.clone();
  copy.tour = (Tour) tour.clone();
  return copy;
}

/**
 *
 */
public void twoPointMove() {
  tour.twoPointMove();
}

public boolean twoPointMove(ListNode n) {
  return tour.twoPointMove(n);
}

public boolean twoPointMove(ListNode n, Vehicle v) {
  return tour.twoPointMove(n, v.getTour());
}

public RTRInfo get2OptRTRInfo(ListNode n,Record r) {
  return tour.get2OptRTRInfo(n,r);
}

public RTRInfo get2OptRTRInfo(ListNode n, Vehicle v, Record r
 ) {
  return tour.get2OptRTRInfo(n,v.getTour(),r);
}

public void do2OptRTR(ListNode from, RTRInfo info) {
  tour.do2OptRTR(from,info);
}

public RTRInfo getOnePointMoveRTRInfo(ListNode n, Vehicle v,
 Record r) {
  return tour.getOnePointMoveRTRInfo(n,v.getTour(),r);
}

public RTRInfo getOnePointMoveRTRInfo(ListNode n, Record r) {
  return tour.getOnePointMoveRTRInfo(n,r);
}
```

```
public void doOnePointMoveRTR(ListNode n,RTRInfo info) {
  tour.doOnePointMoveRTR(n,info);
}

public RTRInfo getTwoPointMoveRTRInfo(ListNode n, Vehicle v,
 Record r) {
  return tour.getTwoPointMoveRTRInfo(n,v.getTour(),r);
}

public void doTwoPointMoveRTR(ListNode n, RTRInfo info) {
  tour.doTwoPointMoveRTR(n,info);
}
/**
 * @return
 */
public int length() {
  return tour.length();
}

// transfer all customers in this vehicle to other
public void transferNode(Vehicle other) {
  // precodition: other is an emtpy vehicle
  logger.assertLog(other.getFreeLoad()>=this.getLoad(),"fatal
 error in transferNode,infeasible");
  tour.transferNode(other.getTour());
  //postcodition: this.tour should be emtpy
  logger.assertLog(getLoad()==0,"fatal error after transfer
 Node this tour is not empty");
}

public boolean isTypeOf(VehicleType vt) {
  return type.equals(vt);
}

public Node getFirst() {
  return tour.getFirst();
}

public Node getLast() {
  return tour.getLast();
}
/**
 * @return
 */
public Enumeration elements() {
```

```
    return tour.elements();
  }

  public ListNode remove(Object n) {
    return tour.remove(n);
  }

  public boolean contains(Object n) {
      return tour.contains(n);
  }

  public void addAfter(ListNode node,Object data) {
    tour.addAfter(node,data);
  }

  public void checkSolution() {
    tour.checktLoad();
    tour.checkDistance();
  }

  public void accept(Visitor v) {
      //System.out.println("visitor is "+ v.getClass());
      v.visit(this);
  }
}
```

D.7 VehicleType.java

```
/*
 * Created on 2003-10-14
 *
 * To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and
   Comments
 */
package hvrp;

/**
 * @author lify
 *
 * To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Generation - Code and
   Comments
 */
public class VehicleType implements Comparable{
```

```
  /**
   * the vehicle capacity
   */
  private int capacity;
  /**
   * the distance penalty coefficient
   */
  private double penalty;
  /**
   * the max route length
   */
  private double length;
  /**
   * number of vehicles of this type
   */
  private int size;

  /**
   * fixed cost
   */
  private int cost;

  /**
   * VehicleType Identifier
   */
  private String id;

  /**
   * @return Returns the capacity.
   */
  public int getCapacity() {
    return capacity;
  }

//  // sort according to penalty, used for clarke and wright
    algorithm
//  public int compareTo(Object obj) {
//    if(! (obj instanceof VehicleType)) throw new
    IllegalArgumentException();
//    else {
//      double op = ((VehicleType)obj).penalty;
//      if(penalty<op) return -1;
//      else if (penalty>op) return 1;
//      else return 0;
//    }
```

```
//  }
//  sort dscendingly according to capacity, used for
    bestFitInitSolution
    public int compareTo(Object obj) {
      if(! (obj instanceof VehicleType)) throw new
    IllegalArgumentException();
      else {
        int op = ((VehicleType)obj).getCapacity();
        if(capacity > op) return -1;
        else if (penalty < op) return 1;
        else return 0;
      }
    }

  /**
   * @param capacity The capacity to set.
   */
  public void setCapacity(int capacity) {
    this.capacity = capacity;
  }

  /**
   * @return Returns the length.
   */
  public double getLength() {
    return length;
  }

  /**
   * @param length The length to set.
   */
  public void setLength(double length) {
    this.length = length;
  }

  /**
   * @return Returns the penalty.
   */
  public double getPenalty() {
    return penalty;
  }

  /**
   * @param penalty The penalty to set.
   */
```

```
public void setPenalty(double penalty) {
  this.penalty = penalty;
}

/**
 * @return Returns the size.
 */
public int getSize() {
  return size;
}

/**
 * @param size The size to set.
 */
public void setSize(int size) {
  this.size = size;
}


/**
 * @return Returns the cost.
 */
public int getCost() {
  return cost;
}

/**
 * @param cost The cost to set.
 */
public void setCost(int cost) {
  this.cost = cost;
}

/**
 * @param capacity
 * @param penalty
 * @param length
 * @param size
 * @param cost
 */
public VehicleType(String id, int capacity, double penalty,
 double length, int cost, int size) {
  this.id = id;
  this.capacity = capacity;
  this.penalty = penalty;
```

```
    this.length = length;
    this.size = size;
    this.cost = cost;
  }
  /**
   * @param capacity
   * @param penalty
   * @param size
   * @param cost
   */
  public VehicleType(String id,int capacity, int cost,double
   penalty, int size) {
    this.id = id;
    this.capacity = capacity;
    this.penalty = penalty;
    this.size = size;
    this.cost = cost;
  }

  public String toString() {
      /*
    StringBuffer sb = new StringBuffer();
    sb.append("capacity="+capacity);
    sb.append(",penalty="+penalty);
    sb.append(",size="+size);
    return sb.toString();
    */
      return id+"("+capacity+")";
  }

  public boolean equals(Object obj) {
    if(obj instanceof VehicleType) {
      if(capacity == ((VehicleType)obj).getCapacity()
          && Math.abs(penalty-((VehicleType)obj).getPenalty())
   < 1e-10)
        return true;
      else return false;
    }
    else return false;
  }
}
```

D.8 VRPConfigure.java

```
package hvrp;
```

```
import java.io.*;
import org.apache.log4j.Logger;


public class VRPConfigure {
   static Logger logger= Logger.getLogger(VRPConfigure.class);
    public static VRPGraph readInfo(String data) throws
   IOException,FileNotFoundException{
  BufferedReader in = new BufferedReader(new FileReader(data));
  String content;
  VRPGraph g=null;
  VehicleType [] fleet=null;
  Node nodes [] = null;
  while( (content=in.readLine())!=null ) {
      if( content.trim().startsWith("#")) continue;
      String[] format = content.trim().split("=");
      if(format[0].trim().equalsIgnoreCase("vrpdata")) {
    try {
        logger.info("read node data " + format[1]);
        nodes = VRPFileReader.readNode(format[1]);
    } catch(IOException io) {
        logger.error(io);
        throw io;
    }
      }
      if(format[0].trim().equalsIgnoreCase("fleetdata")){
    try {
        logger.info("read fleet data "+ format[1]);
        fleet = VRPFileReader.getFleet(format[1]);

    }catch(IOException io) {
        logger.error(io);
        throw io;
    }
      }
  }
  return new VRPGraph(nodes,fleet);
    }
}
```

D.9 Visitor.java

```
/*
 * Created on 2003-10-27
 *
```

```
 * To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and
   Comments
 */
package hvrp;

/**
 * @author lify
 *
 * To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Generation - Code and
   Comments
 */
public interface Visitor {
  public void visit(Object o);
}
```

D.10 Visitable.java

```
/*
 * Created on 2003-10-27
 *
 * To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and
   Comments
 */
package hvrp;

/**
 * @author lify
 *
 * To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Generation - Code and
   Comments
 */
public interface Visitable {
  public void accept(Visitor v);
}
```

D.11 HVRPTest.java

```
package hvrp;
import junit.framework.*;
import java.util.*;
```

```
import org.apache.log4j.Logger;
import org.apache.log4j.BasicConfigurator;

public class HVRPTest extends TestCase {
  private VRPGraph g;
  Random rand;
  Node[] nodes;
  int problem_size;
  static String pid = "15";
  String fn = "/home/lify/workspace/hvrp/data/" + pid + ".info"
   ;
  Node n1, n2, n3, depot;
  Vehicle[] vehicles;
  SolutionPlot plot;
  GnuplotWriter gw;
  String output = "/home/lify/workspace/hvrp/data/" + pid + ".
   sol";
  String gnuplotOutput = "/home/lify/workspace/hvrp/data/" +
   pid + ".gp";
  SolutionWriter sw ;
  String solFile = "/home/lify/workspace/hvrp/data/"+pid+".
   solution";
  /*
   * static logger
   */
  static Logger logger;
  static {
    logger = Logger.getLogger(HVRPTest.class);
    BasicConfigurator.configure();
  }

  protected void setUp() throws Exception {
    logger.info("enter setup");
    g = VRPConfigure.readInfo(fn);
    GraphFactory.setGraph(g);
    n1 = new Node(0, 0, 10);
    n2 = new Node(1, 1, 5);
    n3 = new Node(2, 2, 7);
    depot = new Node(-1, -1, 3);
    vehicles = g.getVehicles();
    assertTrue(vehicles != null);
    rand = new Random();
    plot = new SolutionPlot(output);
    gw = new GnuplotWriter(gnuplotOutput);
    sw = new SolutionWriter(solFile);
```

```
  logger.info("leave setup");
}

public void __testVisitPattern() {
    VehicleType vt = new VehicleType("A",10,20,50,3);
    Vehicle v = new Vehicle(vt,n1);
    v.add(n2); v.add(n3);
    v.accept(sw);
    sw.close();
}
//public void testGraph() {
//logger.info(g);
//g.dump();
//}

//    public void testLinkedList() {
//      LinkedList list = new LinkedList();
//
//      list.add(n1);
//      list.add(n2);
//      list.add(n3);
//      logger.info(list);
//      list.remove(1);
//      logger.info(list);
//      list.add(n3);
//      logger.info(list);
//      list.remove(1);
//      logger.info(list);
//    }

//    public void testTour() {
//      Tour tour = new Tour(depot);
//      tour.add(n1);
//      //logger.info(tour);
//      tour.add(n2);
//      //logger.info(tour);
//      tour.add(n3);
//      logger.info(tour);
//      tour.reverse(n1,n2);
//      logger.info(tour);
//      tour.remove(2);
//      logger.info(tour);
//    }

//    public void testHGraph() {
```

```
//        int [] initSeq = MyRandom.getRandPerm(g.size());
//        Tour tour = new Tour(g.getDepot());
//        for(int i=0;i<g.size();i++) {
//          if(initSeq[i]!=0) tour.add(g.getNodeById(initSeq[i
 ]));
//        }
//        logger.info(tour);
//        tour.reverse(1,4);
//        logger.info(tour);
//    }

/*
 * public void test13() { assertTrue(vehicles[0][0]!=null);
 * vehicles[0][0].add(g.nodes[26]); assertTrue(vehicles
 [0][1]!=null);
 * vehicles[0][1].add(g.nodes[17]); assertTrue(vehicles
 [0][2]!=null);
 * vehicles[0][2].add(g.nodes[6]); vehicles[0][3].add(g.nodes
 [16]);
 * vehicles[1][0].add(g.nodes[4]); vehicles[1][1].add(g.nodes
 [33]);
 * vehicles[2][0].add(g.nodes[40]); int [] cust = { 7,35,19
 }; for(int
 * i=0;i <cust.length;i++) vehicles[2][1].add(g.nodes[cust[i
 ]]); int
 * []cust1 = { 27,15,13}; for(int i=0;i
 * <cust1.length;i++) vehicles[2][2].add(g.nodes[cust1[i]]);
 int cust2[] = {
 * 25,31 }; for(int i=0;i
 * <cust2.length;i++) vehicles[2][3].add(g.nodes[cust2[i]]);
 int []cust3 = {
 * 49,24,18,50 }; for(int i=0;i
 * <cust3.length;i++) vehicles[3][0].add(g.nodes[cust3[i]]);
 int []cust4 = {
 * 22,28,2}; for(int i=0;i
 * <cust4.length;i++) vehicles[3][1].add(g.nodes[cust4[i]]);
 int [] cust5 = {
 * 8,46,34 }; for(int i=0;i
 * <cust5.length;i++) vehicles[3][2].add(g.nodes[cust5[i]]);
 int [] cust6 = {
 * 1,43,42,41,23 }; for(int i=0;i
 * <cust6.length;i++) vehicles[3][3].add(g.nodes[cust6[i]]);
 int []cust7 = {
 * 10,38,11,14 }; for(int i=0;i
 * <cust7.length;i++) vehicles[4][0].add(g.nodes[cust7[i]]);
```

```
int [] cust8 = {
* 12,39,9,32,44,3}; for(int i=0;i
* <cust8.length;i++) vehicles[4][1].add(g.nodes[cust8[i]]);
int []cust9 = {
* 45,29,5,37,20,36,47,21,48,30 }; for(int i=0;i
* <cust9.length;i++) vehicles[5][0].add(g.nodes[cust9[i]]);
*
* for(int i=0;i <vehicles.length;i++) { for(int j=0;j
* <vehicles[i].length;j++) { System.out.println("load =" +
* vehicles[i][j].getLoad()+ "capacity="+vehicles[i][j].
getCapacity()); } } }
*/

//     public void test15() {
//     HVRPInstance hvrp = new HVRPInstance(g);
//     Vehicle[] vehicles = g.getVehicles();
//  int [] cust = { 5,38,46};
//  for(int i=0;i<cust.length;i++) vehicles[0].append(g.nodes
 [cust[i]]);
//  int []cust1 = { 24,43,7,26};
//  for(int i=0;i<cust1.length;i++) vehicles[1].append(g.
 nodes[cust1[i]]);
//  int cust2[] = { 4,19,40,42,37 };
//  for(int i=0;i<cust2.length;i++) vehicles[2].append(g.
 nodes[cust2[i]]);
//  int []cust3 = { 48,23,6};
//  for(int i=0;i<cust3.length;i++) vehicles[3].append(g.
 nodes[cust3[i]]);
//  int []cust4 = { 41,13,25,14};
//  for(int i=0;i<cust4.length;i++) vehicles[4].append(g.
 nodes[cust4[i]]);
//  int [] cust5 = { 18,47,12};
//  for(int i=0;i<cust5.length;i++) vehicles[5].append(g.
 nodes[cust5[i]]);
//  int [] cust6 = { 17,44,15,45,33,39,10,49};
//  for(int i=0;i<cust6.length;i++) vehicles[6].append(g.
 nodes[cust6[i]]);
//  int []cust7 = { 27,8,31,28,3,36,35,20,22,1};
//  for(int i=0;i<cust7.length;i++) vehicles[7].append(g.
 nodes[cust7[i]]);
//  int [] cust8 = { 32,2,29,21,34,30,9,50,16,11};
//  for(int i=0;i<cust8.length;i++) vehicles[8].append(g.
 nodes[cust8[i]]);
//
//  logger.info(hvrp);
```

```
//   plot.print(hvrp);
//   plot.close();
//   gw.setTerm("postscript color eps");
//       gw.setOutput("/home/lify/workspace/hvrp/data/15.eps")
 ;
//       gw.setKey("box");
//       gw.setXLabel("X");
//       gw.setYLabel("Y");
//       gw.setTitle("HVRP Initial Solution: Problem 15");
//       gw.plot(hvrp.getSolution(),output);
//       gw.close();
//   }

//     public void testInsertion() {
//       int i = rand.nextInt(g.size()-1)+1;
//       int j = rand.nextInt(g.size()-1)+1;
//       Arrays.sort(vehicles);
//       for(int k=0;k<vehicles.length;k++)
//         logger.info(vehicles[k]);
//       for(int k=0;k<vehicles.length;k++) {
//         if(vehicles[k].leastCostInsertion(g.getNodeById(i))
 )
//           break;
//       }
//       for(int k=0;k<vehicles.length;k++) {
//         if(vehicles[k].leastCostInsertion(g.getNodeById(j))
 )
//           break;
//       }
//     for(int k=0;k<vehicles.length;k++)
//           logger.info(vehicles[k]);
//     }

//     public void testInstance() throws
 CloneNotSupportedException {
//       HVRPInstance hvrp = new HVRPInstance(g);
//       hvrp.initSolution();
//       //hvrp.leastCostInitSolution();
//       HVRPSolution s = hvrp.RTR();
//       logger.info("best solution\n"+s);
//       plot.print(s);
//       plot.close();
//       gw.setTerm("postscript color eps");
//       gw.setOutput("/home/lify/workspace/hvrp/data/15.eps")
 ;
```

```
//        gw.setKey(" outside box");
//        gw.setXLabel("X");
//        gw.setYLabel("Y");
//        gw.setTitle("HVRP Initial Solution: Problem 15");
//        gw.plot(s,output);
//        gw.close();
//    }

private void plotSolution(HVRPSolution sol, String out) {
  plot.print(sol); plot.close();
      gw.setTerm("postscript color eps");
  gw.setOutput(out);
      gw.setNoKey();
  gw.setSizeSquare();
      gw.setXLabel("X");
      gw.setYLabel("Y");
  double value = Math.floor(sol.getDistance()*100+0.5)/100;
      gw.setTitle("Problem "+pid +" solution value = " +
 value);
      gw.plot(sol,output);
      gw.close();
}

public void testSweep() throws CloneNotSupportedException {
  HVRPInstance hvrp = new HVRPInstance(g);
  HVRPSolution best = hvrp.solve();
  logger.info("best solution for "+pid +" is "+ best);
  plotSolution(best,"/home/lify/workspace/hvrp/data/"+pid+".
 eps");
  best.accept(sw);
  sw.close();
  /*
  HVRPSolution sol = new HVRPSolution(g.getVehicles());
  sol.maxSolution();
  String where = "/home/lify/workspace/hvrp/data/"+pid+"-init
 .eps";
  plotSolution(sol,where);
  //HVRPSolution s = hvrp.solve();
  logger.info("max solution for problem " + pid + "\n" + sol)
 ;
  HVRPSolution best = sol.RTR1();
  logger.info("after RTR solution="+best);
  plotSolution(best, "/home/lify/workspace/hvrp/data/"+pid+".
 eps");
  */
```

```
  }


  //     public void testRank() {
  //       Node n[] = new Node[g.size()-1];
  //       System.arraycopy(g.nodes,1,n,0,g.size()-1);
  //       Arrays.sort(n);
  //       for(int i=0;i<n.length;i++)
  //         logger.info(n[i]+"\t"+n[i].getDemand() +"\t"+
  // g.distance(0,n[i].getId()));
  //     }

//   public void testCWSaving() {
//     HVRPSolution sol = new HVRPSolution(g.getVehicles());
//     //sol.sweepInitSolution(0D);
//     sol.CWSeqSaving();
//     //       Vehicle[] v = sol.getVehicles();
//     //       v[8].remove(g.nodes[8]);
//     //       v[8].remove(g.nodes[7]);
//     logger.info("CWSeqSaving solution distance=" + sol.
    getDistance());
//     logger.info("check solution before purturbation");
//     HVRPSolution initSol[] = new HVRPSolution[5];
//     for (int i = 0; i < 5; i++)
//       initSol[i] = sol;
//     double minLen = 1e10;
//     HVRPSolution bestSol = null;
//     for (int i = 0; i < 5; i++) {
//       HVRPSolution asol = initSol[i].RTR();
//       if (asol.getDistance() < minLen);
//       minLen = asol.getDistance();
//       try {
//         bestSol = (HVRPSolution) asol.clone();
//       } catch (CloneNotSupportedException e) {
//       }
//     }
//
//     //RPSolution bestSol = sol.RTR();
//     //asol.OROpt();
//     //bestsol.checkSolution();
//     bestSol.checkSolution();
//     logger.info("after RTR distance=" + bestSol.getDistance()
    );
//     //       sol.checkSolution();
//     //       sol.RTR();
```

```
//     //     logger.info("after RTR CW distance=" + sol.
    getDistance());
//     //       sol.perturb();
//     //       logger.info("after purturbation distance =" + sol
    .getDistance());
//     //       sol.RTR();
//     //       logger.info("after RTR CW distance="+ sol.
    getDistance());
//     //       sol.checkSolution();
//   }
  public static Test suite() {
    //suite.addTest(VRPTest.class);
    TestSuite suite = new TestSuite(HVRPTest.class);
    return suite;
  }

  public static void main(String args[]) throws Exception {
    HVRPTest.pid = args[0];
    junit.textui.TestRunner.run(suite());
  }
}
```

Appendix E

LandFill Routing Problem Code

E.1 Customer.java

```
package edu.umd.math.lify.vrp;
import edu.umd.math.lify.graph.*;
import java.util.ArrayList;

public class Customer extends Node implements Cloneable {

    /**
     * <code>q</code> is the demand of this node
     *
     */
    private int q;
  /**
   * lastdrop is the landfill that minimizes the distance from
   * the customer to it and it to the depot
   */
    private LandFill lastdrop;

  /**
   * lastDropDist is the distance described above
   */
    private double lastDropDist;

    /**
     * <code>stime</code> is the service time/waiting time.
     */
  private int stime;

  /**
   * <code>g</code> is the VRPGraph
```

```
 */

private VRPGraph g;


/**
 * candLF is the candidate landfill
 * one way to define a candidate is according its distance to
  the node
 * we might consider cost later
 */

 private ArrayList candLF;

  public Customer(double x,double y) {
  super(x,y);
  q = 0;
  stime = 0;
  candLF = new ArrayList(3);
  }

  public Customer(double x,double y, int i) {
  super(x,y,i);
  q = 0;
  stime = 0;
  candLF = new ArrayList(3);
  }

  public Customer(double x,double y,int i,int demand)
  {
  super(x,y,i);
  q = demand;
  stime = 0;
  candLF = new ArrayList(3);
  }

  public Customer(double x, double y, int i, int demand, int
 t)
  {
  super(x,y,i);
  q = demand;
  stime = t;
  candLF = new ArrayList(3);
  }
```

```
  public int getDemand() { return q; }
  public void setDemand(int demand) { q = demand;}

  public int getServiceTime() { return stime;}
  public void setServiceTime(int t) { stime = t;}

  public String toString() {
    StringBuffer sb = new StringBuffer();
  sb.append(super.toString()+"demand("+q+")\tserviceTime("+
 stime+")");
  return sb.toString();
  }

  public int getCandLandFillSize() { return candLF.size(); }
public LandFill getCanLandFill(int i) { return (LandFill)
 candLF.get(i); }
  public LandFill getLastdrop() { return lastdrop; }

  public void setLastdropDist(double d) { lastDropDist = d; }

  public double getLastdropDist() { return lastDropDist;}

  public void setLastdrop(LandFill l) { lastdrop = l; }

public void addLandFill(LandFill l) { candLF.add(l); }

public boolean contains(LandFill l) { return candLF.contains(
 l); }

public double getDist2Node(Node v) {
  if(v instanceof Customer) return g.distance(this,v);
  else if(v instanceof Depot) return lastDropDist;
  else return v.getDist2Node(this);
}

  public Object clone() {
    Customer v = null;
  v = (Customer)super.clone();
  return v;
  }

 public boolean hasCandLandFill(LandFill lf) { return candLF.
 contains(lf); }
```

```
    public boolean equals(Object o) {
     if(o instanceof Customer) {
       if(id==((Customer)o).id) return true;
       else return false;
     }
     else return false;
    }

    public int hashCode() { return id; }

}
```

E.2 LandFill.java

```
package edu.umd.math.lify.vrp;
import edu.umd.math.lify.graph.*;
import java.util.*;

public class LandFill extends Node
{
     static private int custSize;
     private int id;
     /**
       * a list of potential (nearest) customer
       */
     private ArrayList customer;
     private double [] dist;
     public LandFill(double x,double y,int id)
     {
   super(x,y,id);
   customer = new ArrayList();
   dist = new double[custSize];
     }

    /**
      * add one potential customer
      */
     public void register(Node n) throws
    DuplicateCustomerException{
   if(customer.contains(n)) throw new DuplicateCustomerException
    ("customer "+n+ " is already registered!");
   else {
     customer.add(n);
        }
     }
```

```
 /**
  * remove one potential customer
  */
  public void unregister(Node n) throws
 CustomerNotFoundException {
    if(customer.contains(n)) {
  customer.remove(n);
}
else throw new CustomerNotFoundException("customer "+n+" is
 not found!");
  }

 /**
   * set the distance from customer n to this landfill equal
 d
   */
  public void setDist2Cust(Node n,double d) {
    dist[n.getId()] = d;
  }
  /**
    * retrieve the distance from this landfill to customer
 with id i
    */
  public double getDist2Cust(int i) {
    return dist[i];
  }

  public double getDist2Node(Node n) {
    return getDist2Cust(n);
  }
 /**
   * retrieve the distance from this landfill to customer n
   */
  public double getDist2Cust(Node n) {
    return dist[n.getId()];
  }

  /**
    * the total number of potential customer
    */
  public int getCusCount() { return customer.size();}

  public String toString() {
    return "[LF]"+super.toString();
  }
```

```
    /**
     * ugly hack to set the total number of customer
     */
    public static final void setCustCount(int n) { custSize = n
   ;}
    public boolean isLandFill() { return true; }
    public boolean isDepot() { return false; }

    public boolean equals(Object o) {
      if(o instanceof LandFill) {
    if(getId()==((LandFill)o).getId()) return true;
    else return false;
  }
  else return false;
    }

    public int hashCode() { return getId(); }
}
```

E.3 LRSolution.java

```
package edu.umd.math.lify.vrp;
import java.util.*;
import edu.umd.math.lify.util.*;
import edu.umd.math.lify.graph.*;
/**
  * the Landfill Routing Problem Solution
  */
public class LRSolution implements Solution , Cloneable{
  private boolean flag = false;

  public double perc = 0.01;
  static MyRandom rand = new MyRandom(123456789);
  class Chain {
    private int demand;
    private double length;
    private ArrayList nodes;

    public Chain() { nodes = new ArrayList(); demand=0; length
   =0D;}
    public void add(VRPNode n) {
      demand+= n.getDemand();
      length+=n.getServiceTime();
      if(size()>=1) length+=getLast().getDist2Node(n);
      nodes.add(n);
```

```
  }
  public boolean contains(VRPNode n) { return nodes.contains(
 n); }
  public int getDemand() { return demand; }
  public VRPNode getFirst() { return (VRPNode)nodes.get(0); }
  public VRPNode getLast() { return (VRPNode)nodes.get(nodes.
 size()-1); }
  public double getLength() { return length; }
  public VRPNode[] getNodes(){ return (VRPNode[]) nodes.
 toArray(new VRPNode[0]); }
  public boolean inTheSameTour(VRPNode n) { return getFirst()
 .getTourId()==n.getTourId(); }
  public boolean inTheSameTrip(VRPNode n) { return getFirst()
 .getTripId()==n.getTripId(); }
  private void setup() {
    demand = 0;
    length = 0D;
    VRPNode[] v = (VRPNode [])nodes.toArray(new VRPNode[0]);
    //System.out.println("v.length="+v.length+",size()="+size
 ());
    for(int i=0;i<v.length-1;i++){
      demand+=v[i].getDemand();
      length+=v[i].getDist2Node(v[i+1]);
    }
    demand+=v[v.length-1].getDemand();
    if(v.length>1) length+=v[v.length-2].getDist2Node(v[v.
 length-1]);
  }
  public int size() { return nodes.size(); }
  public String toString() {
    StringBuffer sb = new StringBuffer();
    for(int i=0;i<size();i++) sb.append(((Node)nodes.get(i)).
 getId()+",");
    return sb.toString();
  }
}
public boolean debug = false;//true;
private VRPGraph g;
private double length;
private ArrayList tours;

public VRPGraph getGraph() { return g; }

/**
 * return all the nodes in the solution
```

```
 * */
public VRPNode[] getNodes() {
  ArrayList list = new ArrayList(g.size());
  Tour[] t = getTours();
  for(int i=0;i<t.length;i++) {
    Trip[] trips = t[i].getTrips();
    for(int j=0;j<trips.length;j++){
      VRPNode [] n = trips[j].getNodes();
      for(int k=0;k<n.length;k++) list.add(n[k]);
    }
  }
  return (VRPNode[]) list.toArray(new VRPNode[0]);
}

public void dumpNode(){
  g.dump();
}

public LRSolution(Tour[] initTour,VRPGraph g) {
  tours = new ArrayList();
  length=0D;
  for(int i=0;i<initTour.length;i++){
    tours.add(initTour[i]);
    length+=initTour[i].getLength();
  }
  this.g=g;
}

public Object clone() {
  LRSolution copy = null;
  try {
    copy = (LRSolution)super.clone();
  } catch(CloneNotSupportedException e) {System.out.println(e
 ); }
  copy.g = (VRPGraph)g.clone();
  Tour [] copyTour = new Tour[tours.size()];
  Tour [] t = (Tour[])tours.toArray(new Tour[0]);
  for(int i=0;i<t.length;i++){
    copyTour[i] = (Tour) t[i].clone();
    copyTour[i].setGraph(copy.g);
    Trip[] trips = copyTour[i].getTrips();
    for(int j=0;j<trips.length;j++) {
      trips[j].setGraph(copy.g);
      trips[j].fixNode();
    }
```

```
  }
  copy.tours = new ArrayList();
  for(int i=0;i<copyTour.length;i++)
    copy.tours.add(copyTour[i]);
  copy.evaluate();
  return copy;
}

// one-point-move heuristic
// delete one customer and insert it into another place
public double chain(int chainSize) {
  while(chainInsertionStep(chainSize)) ;
  return length;
}

public double chainOne() {
  //debug = true;
  return chain(1);
}

private void setRTR() {
  for(int i=0;i<g.size();i++)
    g.getCustomer(i).setRTR(true);
}
// some observed problems with record-to-record travel
// 1. a node travel back and forth to the original place
public boolean chain_RTR(int chainSize,double record,double
 deviation) {
  //System.out.println("[chain_RTR]record="+record+",
 deviation="+deviation);
  boolean ret = false;
  boolean moved = false;
  setRTR();
  int count=0;
  while(count<g.size()) {
    for(int i=0;i<tours.size();i++) {
      Tour t = (Tour)tours.get(i);
      Trip[] trips = t.getTrips();
      for(int m=0;m<trips.length;m++) {
        boolean special_chain_seek = false;
        VRPNode [] n = trips[m].getNodes();
        for(int j=0;j<n.length-chainSize+1;j++) {
          if(n[j].getRTR()==true) {
            n[j].setRTR(false);
            count++;
```

```
            Chain chain = new Chain();
            int tripSize = trips[m].size();
            if( tripSize < chainSize ) continue;
            else if(tripSize==chainSize) special_chain_seek =
 true;
            for(int k=0;k<chainSize;k++) chain.add(n[j+k]);//
 chain[k]=n[j+k];
            if(special_chain_seek) moved =
 special_chain_seek_rtr(chain,record,deviation);
            else moved = chain_seek_rtr(chain,record,
 deviation);
            if(length<record) {
              record = length;
              deviation = perc*(record-g.getServiceTime());
            }
            if(moved){
              ret = true;
              j+=chainSize-1;
          //break;// if one chain is moved into other trips,
 we should not go on the trip
            }
          //if(improved) break;
          }
        }
        //if(improved) break;
      }
      //if(improved) break;
    }
  }
  //System.out.println("length="+evaluate());
  safe_check("chain_RTR");
  //System.out.println("leaving [chain_RTR]");
  return ret;

}


private boolean chain_seek(Chain chain) {
  if(debug) {
    System.out.print("enter_chain_seek\t");
    System.out.println("chain:"+chain);
  }
  boolean improved = false;
  VRPNode first = chain.getFirst();
  Node prior = first.getPrior();
```

```
int c2 = first.getId();
VRPNode last = chain.getLast();
Node next = last.getNext();
for(int m=0;m<g.NBCount[c2];m++) {
  int c3 = g.NBList[c2][m];
  VRPNode C3 = g.getCustomer(c3);
  Node C4 = C3.getNext();
  //Node prior = first.getPrior();
  /*
  System.out.println("prior:"+prior);
  System.out.println("next:"+next);
  System.out.println("C3:"+C3);
  System.out.println("C4:"+C4);
  */
  // prior-->first.....last->next.......C3->C4
  // prior->next......C3->first.....->last->C4
  boolean cond = true;
  if(prior instanceof VRPNode) {
    if(C3.equals((VRPNode)prior)) cond = false;
  }
  if(C4 instanceof VRPNode) {
    if(chain.contains((VRPNode)C4)) cond = false;
  }
  if((!chain.contains(C3)) && cond ) {
    // fitfall: when the chain is taken out, the trip
becomes empty
    // I WILL WORK ON IT TOMORROW
    double saving = prior.getDist2Node(first)+last.
getDist2Node(next)+C3.getDist2Node(C4);
    saving -= (prior.getDist2Node(next)+C3.getDist2Node(
first)+last.getDist2Node(C4));
    if(saving >1e-10) {
      if(chain.inTheSameTour(C3)) {
        if(chain.inTheSameTrip(C3)) {
          // don't need to do anything except for the max
trip length
          if(debug) System.out.println("[chain same trip] find
 a positive saving "+saving +". chain size "+ chain.size());
          if(debug){
            System.out.println("chain:"+chain);
            System.out.println("insert into:"+C3.getId()+"
->"+C4.getId());
          }
          if(prior instanceof VRPNode) ((VRPNode)prior).
setNext(next);
```

```
            if(next instanceof VRPNode) ((VRPNode)next).
setPrior(prior);
            C3.setNext(first); first.setPrior(C3);
            last.setNext(C4);
            if(C4 instanceof VRPNode) ((VRPNode)C4).setPrior(
last);
            Trip t = first.getTrip();
            //if(debug) System.out.println("before:"+t);
            t.rebuildTrip();
            first.getTour().evaluate();
            evaluate();
            improved = true;
            if(debug) {
              g.dump();
              dump();
            }
            //if(debug) System.out.println("after:"+t);
            break;
          }
          else { //now same tour different trip
            if(C3.getTrip().getDemand()+chain.getDemand()<=g.
getCapacity()) {
              if(debug) System.out.println("[chain same tour]
saving "+saving +".chain size "+ chain.size());
              if(prior instanceof VRPNode) ((VRPNode)prior).
setNext(next);
              if(next instanceof VRPNode) ((VRPNode)next).
setPrior(prior);
              C3.setNext(first); first.setPrior(C3);
              last.setNext(C4);
              if(C4 instanceof VRPNode) ((VRPNode)C4).
setPrior(last);
              Trip t = first.getTrip();
              Trip s = C3.getTrip();
              //if(debug) System.out.println("before:\n"+t+"\
n"+s);
              t.delNode(chain.getNodes());
              t.evaluate();
              //t.getTour().evaluate();
              s.addNode(chain.getNodes());
              s.rebuildTrip();
              s.getTour().evaluate();
              evaluate();
              improved = true;
            }
```

```
          }
        }
        else {
          if(C3.getTrip().getDemand()+chain.getDemand()<=g.
getCapacity()){
            double newTourLength = C3.getTour().getLength()-
C3.getDist2Node(C4)+C3.getDist2Node(first)+
                  last.getDist2Node(C4) + chain.getLength();
            double newLength1 = first.getTour().getLength()-
prior.getDist2Node(first)
                      -chain.getLength()-last.getDist2Node(
next)+ prior.getDist2Node(next);
            if(newTourLength<=g.getDisConstraint()&&
newLength1<=g.getDisConstraint()) {
              if(debug) System.out.println("[chain␣diff␣tour]
find␣a␣positive␣saving␣"+saving +".chain␣size␣"+ chain.size
());

              if(debug){
                System.out.println("chain:"+chain);
                System.out.println("chain.length="+chain.
getLength());
                System.out.println("insert␣into:"+C3.getId()+
"-->"+C4.getId());
                System.out.println("first="+first.getId()+",
last="+last.getId());
                System.out.println("prior="+prior.getId()+",
next="+next.getId());
                System.out.println("expected␣newlength1="+
newLength1);
                System.out.println("expected␣newLength2="+
newTourLength);
                System.out.println("first.getTour.getLength="
+first.getTour().getLength());
                System.out.println("prior.getDist2Node(first)
="+prior.getDist2Node(first));
                System.out.println("last.getDist2Node(next)="
+last.getDist2Node(next));
                System.out.println("prior.getDist2Node(next)=
"+prior.getDist2Node(next));
                System.out.println("before␣insertion");
                dump();
              }
              if(prior instanceof VRPNode) ((VRPNode)prior).
setNext(next);
```

```
                if(next instanceof VRPNode) ((VRPNode)next).
 setPrior(prior);
               C3.setNext(first); first.setPrior(C3);
                last.setNext(C4);
                if(C4 instanceof VRPNode) ((VRPNode)C4).
 setPrior(last);
                Trip t = first.getTrip();
                Trip s = C3.getTrip();
                //if(debug) System.out.println("before:\n"+t+"\
 n"+s);
                t.delNode(chain.getNodes());
                t.evaluate();
                t.getTour().evaluate();

                s.addNode(chain.getNodes());
                s.rebuildTrip();
                s.getTour().evaluate();
              //finally update the solution
                evaluate();
                //if(debug) System.out.println("after:\n"+t+"\n
 "+s);
                improved = true;
              //System.out.println("after:"+t);
                if(debug) {
                  System.out.println("after_insertion");
                  dump();
                }
              }
            }
          }
        }
      }
      if(improved) break;
    }
    //System.out.println("leave chain_seek");
    //if(improved) dump();
    check_data_consistency("after_chain_seek");
    return improved;
  }

  private boolean chain_seek_rtr(Chain chain,double record,
   double deviation) {
    //System.out.println("enter chain_seek_rtr");
    boolean moved = false;
    VRPNode first = chain.getFirst();
```

```
Node prior = first.getPrior();
int c2 = first.getId();
VRPNode last = chain.getLast();
Node next = last.getNext();
double max_saving = -1e10;
VRPNode max_C3 = null;
if(prior instanceof LandFill && next instanceof LandFill){
  System.out.println("this should never happen now!");
  return false;
}
//System.out.println("[rtr]chain:"+chain);
//System.out.println("priro="+prior+",next="+next.getId());
for(int m=0;m<g.NBCount[c2];m++) {
  int c3 = g.NBList[c2][m];
  VRPNode C3 = g.getCustomer(c3);
  Node C4 = C3.getNext();
  //Node prior = first.getPrior();
  // prior-->first.....last->next.......C3->C4
  // prior->next......C3->first.....->last->C4
  boolean cond = true;
  if(prior instanceof VRPNode) {
    if(C3.equals((VRPNode)prior)) cond = false;
  }
  if(C4 instanceof VRPNode) {
    if(chain.contains((VRPNode)C4)) cond = false;
  }
  if((!chain.contains(C3)) && cond ) {
    // fitfall: when the chain is taken out, the trip
becomes empty
    // I WILL WORK ON IT TOMORROW
    double saving = prior.getDist2Node(first)+last.
getDist2Node(next)+C3.getDist2Node(C4);
    saving -= (prior.getDist2Node(next)+C3.getDist2Node(
first)+last.getDist2Node(C4));
    //System.out.println("saving="+saving);
    if(saving>max_saving) {
      if(chain.inTheSameTour(C3)) {
        double newTourLength = C3.getTour().getLength() -
saving;
        if(newTourLength <= g.getDisConstraint()) {
          if(chain.inTheSameTrip(C3)) {
            // don't need to do anything except for the max
 trip length
            //System.out.println("[chain same trip]find a
positive saving "+saving +". chain size "+ chain.size());
```

```
                if(length-saving<=record+deviation) {
                    max_saving = saving;
                    max_C3 = C3;
                }
                if(debug){
                  System.out.println("chain:"+chain);
                  System.out.println("insert into:"+C3.getId()+
"->"+C4.getId());
                }
              }
              else { //now same tour different trip
                  //need to check feasibility now
                if(C3.getTrip().getDemand()+chain.getDemand()<=
g.getCapacity()){
                  if(length-saving<=record+deviation) {
                    max_saving = saving;
                    max_C3 = C3;
                  }
                }
              }
            }
          }
          else {
            //double newTourLength1 = first.getTour().getLength
()- chain.getLength()-prior.getDist2Node(first)
                        //-last.getDist2Node(next) + prior.
getDist2Node(next);
            //double newTourLength2 = C3.getTour().getLength()
+ chain.getLength() + C3.getDist2Node(first)
                        //+ last.getDist2Node(C4) - C3.
getDist2Node(C4);
            if(C3.getTrip().getDemand()+chain.getDemand()<=g.
getCapacity()){
              double newTourLength = C3.getTour().getLength()-
C3.getDist2Node(C4)+C3.getDist2Node(first)+
                    last.getDist2Node(C4) + chain.getLength();
              double newLength1 = first.getTour().getLength()-
prior.getDist2Node(first)
                        -chain.getLength()-last.getDist2Node(
next)+ prior.getDist2Node(next);
              if(newTourLength<=g.getDisConstraint()&&
newLength1<=g.getDisConstraint()) {
                if(length - saving <= record + deviation) {
                  max_saving = saving;
```

```
                max_C3 = C3;
              }
            }
          }
        }
      }
    }
  }
  if(max_C3!=null) { // here we do the record-to-record
travel
    //dump();
    Node max_C4 = max_C3.getNext();
    if(debug) System.out.println("[doing rtr] in tour("+max_C3
.getTourId()+")C3="+max_C3.getId()+",C4="+max_C4.getId()+"
chain="+chain+" prior="+prior.getId()+" next="+next.getId()+"
from tour("+max_C3.getTourId()+")");
    //dump();
    if(chain.inTheSameTour(max_C3)) {
      if(chain.inTheSameTrip(max_C3)) {
        if(prior instanceof VRPNode) ((VRPNode)prior).setNext
(next);
        if(next instanceof VRPNode) ((VRPNode)next).setPrior(
prior);
        max_C3.setNext(first); first.setPrior(max_C3);
        last.setNext(max_C4);
        if(max_C4 instanceof VRPNode) ((VRPNode)max_C4).
setPrior(last);
        Trip t = first.getTrip();
        t.rebuildTrip();
        t.getTour().evaluate();
        evaluate();
        if(length > record+deviation) System.out.println("
wrong in chain_seek_rtr");
        moved=true;
        //     System.out.println("after rtr");
        //dump();
      }
      else {  // same tour diff trip
          //  System.out.println("[chain same tour]saving "+
saving +".chain size "+ chain.size());
        if(prior instanceof VRPNode) ((VRPNode)prior).setNext
(next);
        if(next instanceof VRPNode) ((VRPNode)next).setPrior(
prior);
        max_C3.setNext(first); first.setPrior(max_C3);
```

```
        last.setNext(max_C4);
        if(max_C4 instanceof VRPNode) ((VRPNode)max_C4).
setPrior(last);
        Trip t = first.getTrip();
        Trip s = max_C3.getTrip();
        //if(debug) System.out.println("before:\n"+t+"\n"+s);
        t.delNode(chain.getNodes());
        t.evaluate();
        //t.getTour().evaluate();
        s.addNode(chain.getNodes());
        s.rebuildTrip();
        s.getTour().evaluate();
        evaluate();
        if(length > record+deviation) System.out.println("
wroing same tour diff trip in chain seek rtr");
        //System.out.println("after rtr");
        //dump();
        moved = true;
      }
    }
    else { // diff tour

        if(prior instanceof VRPNode) ((VRPNode)prior).setNext
(next);
        if(next instanceof VRPNode) ((VRPNode)next).setPrior(
prior);
        max_C3.setNext(first); first.setPrior(max_C3);
        last.setNext(max_C4);
        if(max_C4 instanceof VRPNode) ((VRPNode)max_C4).
setPrior(last);
        Trip t = first.getTrip();
        Trip s = max_C3.getTrip();
        //if(debug) System.out.println("before:\n"+t+"\n"+s);
        t.delNode(chain.getNodes());
        t.evaluate();
        t.getTour().evaluate();
        //t.getTour().evaluate();
        s.addNode(chain.getNodes());
        s.rebuildTrip();
        s.getTour().evaluate();
        evaluate();
        if(length > record+deviation) System.out.println("
wrong diff tour chain seek rtr");
        moved = true;
        //System.out.println("after rtr");
```

```
        //dump();
      }
    safe_check("chain_seek_rtr()C3="+max_C3.getId()+",C4="+
 max_C4.getId()+"chain="+chain+"\n");
    }
    //System.out.println("finish for chain:"+chain);
    //System.out.println("leave_chain_seek_rtr");
    return moved;

  }

private boolean chainInsertionStep(int chainSize) {
  boolean improved = false;
  for(int i=0;i<tours.size();i++) {
    Tour t = (Tour)tours.get(i);
    Trip[] trips = t.getTrips();
    for(int m=0;m<trips.length;m++) {
      boolean special_chain_seek = false;
      VRPNode [] n = trips[m].getNodes();
      int tripSize = trips[m].size();
      if( tripSize < chainSize ) continue;
      else if(tripSize==chainSize) special_chain_seek = true;
      for(int j=0;j<n.length-chainSize+1;j++) {
        Chain chain = new Chain();
        for(int k=0;k<chainSize;k++) chain.add(n[j+k]);//
 chain[k]=n[j+k];
        if(debug) System.out.println("chain="+chain);
        if(special_chain_seek) improved = special_chain_seek(
 chain);
        else improved = chain_seek(chain);
        if(improved) {
          if(debug) System.out.println("done_something_on_
 chain_"+chain);
          break;
        }
      }
      if(improved) break;
    }
    if(improved) break;
  }
  //System.out.println("length="+evaluate());
  safe_check("after_chainInserionStep");
  return improved;
}
```

```
private boolean check_feasibility() {
  Tour[] t = getTours();
  for(int i=0;i<t.length;i++)
    if(t[i].evaluate()>g.getDisConstraint())
      return false;
  return true;
}

// precodition: the orientation is prior->from->...->to->next
// after this procedure, the tour becomes prior->to->..->from
 ->next
private boolean check_feasibility(VRPNode prior,VRPNode from,
  VRPNode to, VRPNode next) {
  // since this is from two tours
  // we first need to check individal trip is feasible, then
 the new tour is feasible w.r.t
  // distance constraint
  int tourId1 = prior.getTourId();
  int tourId2 = next.getTourId();
  int partial_load1 = 0;
  int partial_load2 = 0;
  double partial_dist1 = 0D;
  double partial_dist2 = 0D;
  Node itr = from;
  while(itr instanceof VRPNode) {
    partial_load1+=((VRPNode)itr).getDemand();
    partial_dist1+=((VRPNode)itr).getServiceTime();
    Node tmp = ((VRPNode)itr).getNext();
    if(tmp instanceof VRPNode) {
      partial_dist1+=g.distance(itr.getId(),tmp.getId());
    }
    else { // tmp is a landfill
      partial_dist1 += ((VRPNode)itr).getDist2LandFill((
 LandFill)tmp);
    }
    itr = tmp;
  }
  itr = to;
  while(itr instanceof VRPNode) {
    partial_load2+=((VRPNode)itr).getDemand();
    partial_load2+=((VRPNode)itr).getServiceTime();
    Node tmp = ((VRPNode)itr).getPrior();
    if(tmp instanceof VRPNode) partial_dist2 +=g.distance(itr
 .getId(),tmp.getId());
    else partial_dist2 += ((LandFill)tmp).getDist2Cust(itr);
```

```
    itr = tmp;
  }

  //first check capacity constraint

  int demand1 = from.getTrip().getDemand();
  int demand2 = to.getTrip().getDemand();

  if(demand1-partial_load1+partial_load2<=g.getCapacity() &&
    demand2-partial_load2+partial_load1<=g.getCapacity()) {

  // we need add the remaining distance from other trips
    partial_dist1 += prior.getTour().getRemainDistance(prior.
 getTrip());
    partial_dist2 += next.getTour().getBeforeDistance(next.
 getTrip());

    double dist1 = from.getTour().getLength();
    double dist2 = to.getTour().getLength();
    double newdist1 = dist1-partial_dist1+g.distance(prior,to
 );
    double newdist2 = dist2-partial_dist2+g.distance(from,
 next);

    if(newdist1<=g.getDisConstraint() && newdist2<=g.
 getDisConstraint())
      return true;
    else return false;

  }
  else return false;
}

private void delTour(Tour t) {
  if(t.size()!=0) {
    System.out.println("tour not empty! should not be deleted
 ");
    System.exit(1);
  }
  tours.remove(t);
  check_data_consistency("after delTour");
  evaluate();
}

// c1->c2  => c1->c3
```

```
// c4->c3  => c4->c2
private boolean diff_tour_seek(VRPNode C1,Node C2,VRPNode C3,
 Node C4) {
  boolean improved = false;
  int c1=C1.getId();int c2=C2.getId(); int c3 = C3.getId();
 int c4=C4.getId();
  //System.out.println("diff_tour_seek("+c1+","+c2+","+c3
 +","+c4+")");
  //System.out.println("find a 2opt move in the tour with
 saving "+saving);
  if(feasible_twopt_move(C1,C2,C3,C4)) {
    if(debug) System.out.println("[global_2opt_seek]find a 2
 opt move ");
    //System.out.print("before reverse ");
    //System.out.println("("+c1+","+c2+","+c4+","+c3+")");
    //System.out.println(this);
    //dump();
    //reverse(C1,C2,C4,C3);
    do_2opt_move(C1,C2,C3,C4);
    //System.out.println("after reverse");
    //System.out.println(this);
    //dump();
    improved = true;
    safe_check("diff_tour_seek");
  }
  return improved;

}

private void do_2opt_move(VRPNode C1, Node C2, VRPNode C3,
 Node C4) {

  int tripId1 = C1.getTripId();
  int tripId2 = C3.getTripId();
  int partial_load1 = 0;
  int partial_load2 = 0;
  LandFill lf1 = C1.getTrip().getLastLandFill();
  LandFill lf2 = C3.getTrip().getFirstLandFill();
  //1. set up the prior and next pointer for nodes following
 from and before to
  Trip t1 = C1.getTrip();
  Trip t2 = C3.getTrip();
  VRPNode [] n1 = t1.getNodes();
  VRPNode [] n2 = t2.getNodes();
```

```
ArrayList a = new ArrayList();
ArrayList b = new ArrayList();

Node itr = C2;
while(itr instanceof VRPNode) {
  Node tmp = ((VRPNode)itr).getNext();
  a.add((VRPNode)itr);
  t1.delNode((VRPNode)itr);
  //partial_load1+=((VRPNode)itr).getDemand();
  itr = tmp;
}
itr = C3;
while(itr instanceof VRPNode) {
  Node tmp = ((VRPNode)itr).getNext();
  b.add((VRPNode)itr);
  t2.delNode((VRPNode)itr);
  //partial_load2+=((VRPNode)itr).getDemand();
  itr = tmp;

}
C1.setNext(C3);
C3.setPrior(C1);
if(C4 instanceof VRPNode) ((VRPNode)C4).setNext(C2);
if(C2 instanceof VRPNode) ((VRPNode)C2).setPrior(C4);
t1.addNode((VRPNode[])b.toArray(new VRPNode[0]));
t2.addNode((VRPNode[])a.toArray(new VRPNode[0]));
// change the landfill for these two trip
t1.setLastLandFill(lf2);
t2.setFirstLandFill(lf1);

t1.rebuildTrip();
t2.rebuildTrip();
// assume the tours are properly ordered in the ArrayList
Trip[] t1after = t1.getTour().getTripsAfter(t1);
Trip[] t2after = t2.getTour().getTripsAfter(t2);
if(t1after!=null){
  t1.getTour().delTrip(t1after);
  t2.getTour().addTrip(t2after);
}
if(t2after!=null){
  t2.getTour().delTrip(t2after);
  t1.getTour().addTrip(t1after);
}
t1.getTour().evaluate();
t2.getTour().evaluate();
```

```
  evaluate();
}

public void dump() {
  Tour[] t = getTours();
  for(int i=0;i<t.length;i++)
    System.out.println(t[i]);
}

public double evaluate() {
  length=0D;
  Tour [] initTour = (Tour[])tours.toArray(new Tour[0]);
  for(int i=0;i<initTour.length;i++){
    length+=initTour[i].getLength();
    //length+=initTour[i].evaluate();
  }
  return length;
}

public double exchange() {
  while(exchange_move_step()) ;
  return length;
}

private boolean exchange_move_step() {
  Tour [] t = getTours();
  boolean improved = false;
  for(int i=0;i<t.length;i++) {
    VRPNode[] n = t[i].getNodes();
    for(int j=0;j<n.length;j++) {
      improved = exchange_seek(n[j]);
      if(improved) break;
    }
    if(improved) break;
  }
  return improved;
}

private boolean exchange_seek(VRPNode v) {
    //System.out.println("enter exchange_seek");
    boolean improved = false;
    int id = v.getId();
    for(int i=0;i<g.NBCount[id];i++) {
  int c2 = g.NBList[id][i];
  VRPNode C2 = g.getCustomer(c2);
```

```
 if(C2.getTourId()!=v.getTourId()) {
      Node vprior = v.getPrior();
      Node vnext = v.getNext();
      Node C2prior = C2.getPrior();
      Node C2next = C2.getNext();
      Trip vt = v.getTrip();
      Trip ct = C2.getTrip();
      double d1 = vprior.getDist2Node(C2)+C2.getDist2Node(
vnext);
      double d2 = C2prior.getDist2Node(v)+v.getDist2Node(
C2next);
      double d3 = vprior.getDist2Node(v)+v.getDist2Node(vnext
);
      double d4 = C2prior.getDist2Node(C2)+C2.getDist2Node(
C2next);
      double d5 = C2.getServiceTime()-v.getServiceTime();
      double saving = d3+d4-(d1+d2);
      if(saving>1e-10) {
    if(vt.getDemand()-v.getDemand()+C2.getDemand()<=g.
getCapacity()
      && ct.getDemand()-C2.getDemand()+v.getDemand()<=g.
getCapacity()) {
        double newLength1 = v.getTour().getLength()-d3+d1+d5;
        double newLength2 = C2.getTour().getLength()-d4+d2-d5
;
        if(newLength1<=g.getDisConstraint() && newLength2<=g.
getDisConstraint()){
      if(debug) System.out.println("[exchange_seek]find_a_
positive_saving_"+saving);
      v.setPrior(C2prior); if(C2prior instanceof VRPNode) ((
VRPNode)C2prior).setNext(v);
      v.setNext(C2next); if(C2next instanceof VRPNode) ((
VRPNode)C2next).setPrior(v);
      C2.setPrior(vprior); if(vprior instanceof VRPNode) ((
VRPNode)vprior).setNext(C2);
      C2.setNext(vnext);    if(vnext instanceof VRPNode) ((
VRPNode)vnext).setPrior(C2);
      vt.delNode(v);
      ct.delNode(C2);
      vt.addNode(C2);
      ct.addNode(v);
      vt.rebuildTrip();
      ct.rebuildTrip();
      vt.getTour().evaluate(); ct.getTour().evaluate();
      evaluate();
```

```
      improved = true;
        }
      }
      }
  }
  if(improved) break;
    }
    safe_check("exchange_seek");
    //System.out.println("leave exchange_seek");
    return improved;
}

public boolean exchange_rtr(double record, double deviation)
 {
    boolean ret = false;
    boolean moved = false;
    int [] seq = MyRandom.getRandPerm(g.size());
    setRTR();
    int count=0;
    //System.out.println("begin exchage_rtr()");

    while(count<g.size()) {
  Tour[] t = getTours();
  for(int i=0;i<t.length;i++) {
      VRPNode[] v = t[i].getNodes();
      for(int j=0;j<v.length;j++) {
    if(v[j].getRTR()==true){
        v[j].setRTR(false);
        moved = exchange_seek_rtr(v[j],record,deviation);
        count++;
        if(length<record) {
      record = length;
      deviation = (record-g.getServiceTime())*perc;
        }
        if(moved) ret = true;
    }
      }

  }
  //System.out.println("count="+count);
    }

    //System.out.println("done exchange_rtr()");
    /*
       for(int i=0;i<g.size();i++) {
```

```
        VRPNode v = g.getCustomer(seq[i]);
        moved = exchange_seek_rtr(v,record,deviation);
        if(length<record) {
        record = length;
        deviation = (record-g.getServiceTime())*perc;
        }
        if(moved) ret = true;
        }
        */
    return ret;
}

private boolean exchange_seek_rtr(VRPNode v, double record,
 double deviation) {
    boolean moved = false;
    int id = v.getId();
    VRPNode max_C2=null;
    double max_saving = -1e10;
    for(int i=0;i<g.NBCount[id];i++) {
  int c2 = g.NBList[id][i];
  VRPNode C2 = g.getCustomer(c2);
  if(C2.getTourId()!=v.getTourId()) {
      Node vprior = v.getPrior();
      Node vnext = v.getNext();
      Node C2prior = C2.getPrior();
      Node C2next = C2.getNext();
      Trip vt = v.getTrip();
      Trip ct = C2.getTrip();
      double d1 = vprior.getDist2Node(C2)+C2.getDist2Node(
 vnext);
      double d2 = C2prior.getDist2Node(v)+v.getDist2Node(
 C2next);
      double d3 = vprior.getDist2Node(v)+v.getDist2Node(vnext
 );
      double d4 = C2prior.getDist2Node(C2)+C2.getDist2Node(
 C2next);
      double d5 = C2.getServiceTime()-v.getServiceTime();
      double saving = d3+d4-(d1+d2);
      if(saving>max_saving) {
    if(vt.getDemand()-v.getDemand()+C2.getDemand()<=g.
 getCapacity()
     && ct.getDemand()-C2.getDemand()+v.getDemand()<=g.
 getCapacity()) {
        double newLength1 = v.getTour().getLength()-d3+d1+d5;
        double newLength2 = C2.getTour().getLength()-d4+d2-d5
```

```
;
        if(newLength1<=g.getDisConstraint() && newLength2<=g.
getDisConstraint()){
      if(length-saving <=record+deviation) {
          max_saving = saving;
          max_C2 = C2;
      }
        }
      }
      }
 }
   }
   if(max_C2!=null) {
 Node vprior = v.getPrior();
 Node vnext = v.getNext();
 Node C2prior = max_C2.getPrior();
 Node C2next = max_C2.getNext();
 Trip vt = v.getTrip();
 Trip ct = max_C2.getTrip();
 if(debug) System.out.println("[exchange_seek_rtr]exchage_
node("+v.getId()+")_with_node("+max_C2.getId()+")" +"_saving
="+max_saving);
 v.setPrior(C2prior); if(C2prior instanceof VRPNode) ((
VRPNode)C2prior).setNext(v);
 v.setNext(C2next); if(C2next instanceof VRPNode) ((VRPNode)
C2next).setPrior(v);
 max_C2.setPrior(vprior); if(vprior instanceof VRPNode) ((
VRPNode)vprior).setNext(max_C2);
 max_C2.setNext(vnext);    if(vnext instanceof VRPNode) ((
VRPNode)vnext).setPrior(max_C2);
 vt.delNode(v);
 ct.delNode(max_C2);
 vt.addNode(max_C2);
 ct.addNode(v);
 vt.rebuildTrip();
 ct.rebuildTrip();
 vt.getTour().evaluate(); ct.getTour().evaluate();
 evaluate();
 moved = true;
   }
   safe_check("exchage_seek_rtr");
   return moved;
}

private boolean feasible_twopt_move(VRPNode C1,Node C2,
```

```
VRPNode C3, Node C4) {
   // since this is from two tours
   // we first need to check individal trip is feasible,
then the new tour is feasible w.r.t
   // distance constraint
   int tourId1 = C1.getTourId();
   int tourId2 = C3.getTourId();
   int partial_load1 = 0;
   int partial_load2 = 0;
   double partial_dist1 = 0D;
   double partial_dist2 = 0D;
   Node itr = C2;
   while(itr instanceof VRPNode) {
 partial_load1+=((VRPNode)itr).getDemand();
 partial_dist1+=((VRPNode)itr).getServiceTime();
 Node tmp = ((VRPNode)itr).getNext();
 partial_dist1+=itr.getDist2Node(tmp);
 itr = tmp;
   }
   itr = C3;
   while(itr instanceof VRPNode) {
 partial_load2+=((VRPNode)itr).getDemand();
 partial_dist2+=((VRPNode)itr).getServiceTime();
 Node tmp = ((VRPNode)itr).getNext();
 partial_dist2 +=((VRPNode)itr).getDist2Node(tmp);
 itr = tmp;
   }

   //first check capacity constraint

   int demand1 = C1.getTrip().getDemand();
   int demand2 = C3.getTrip().getDemand();

   if(demand1-partial_load1+partial_load2<=g.getCapacity()
&&
     demand2-partial_load2+partial_load1<=g.getCapacity()) {

 // we need add the remaining distance from other trips
 partial_dist1 += C1.getTour().getRemainDistance(C1.getTrip
());
 partial_dist2 += C3.getTour().getRemainDistance(C3.getTrip
());

 double dist1 = C1.getTour().getLength();
 double dist2 = C3.getTour().getLength();
```

```
  double newdist1 = dist1-partial_dist1+C1.getDist2Node(C3)-
 C1.getDist2Node(C2)+partial_dist2;
  double newdist2 = dist2-partial_dist2+C4.getDist2Node(C2)-
 C4.getDist2Node(C3)+partial_dist1;

  if(newdist1<=g.getDisConstraint() && newdist2<=g.
 getDisConstraint()){
      //System.out.println("RemainDistance1="+C1.getTour().
 getRemainDistance(C1.getTrip());
      //System.out.println("RemainDistance2="+C3.getTour().
 getRemainDistance(C3.getTrip());
      //System.out.println("expected newlength1="+newdist1);
      //System.out.println("expected newlength2="+newdist2);
      return true;
  }
  else return false;

      }
    else return false;
}

public double getLength() { return length; }

public Tour[] getTours() { return (Tour[])tours.toArray(new
 Tour[0]); }

private double global_twOpt() {
    while(global_twopt_step()) ;
    return length;
}


private boolean global_twopt_seek(VRPNode C1) {
    //System.out.println("[global_twopt_seek]"+C1.getId());
    boolean improved = false;
    int c1 = C1.getId();
    Node C2 = C1.getNext();
    int c2 = C2.getId();
    double d12 = C1.getDist2Node(C2);
    for(int m=0;m<g.NBCount[c1];m++) {
  int c3 = g.NBList[c1][m];
  VRPNode C3 = g.getCustomer(c3);
  if(C3.getTourId()==C1.getTourId()) continue;
  double d13 = g.distance(c1,c3);
  //if(d23>=d12) break;
```

```
  //if(c3==c2) continue;
  Node C4 = C3.getPrior();
  int c4 = C4.getId();
  //System.out.println("[global_twopt_seek]("+c1+","+c2+","+
 c4+","+c3+")");
  //if(c4==c1) continue;
  if(C2 instanceof VRPNode || C4 instanceof VRPNode) {
      double d43 = C4.getDist2Node(C3);
      double d42 = C4.getDist2Node(C2);
      //System.out.println("d12="+d12+",d43="+d43+",d13="+d13
 +",d42="+d42);
      double saving = d12+d43-d13-d42;
      //System.out.println("saving="+saving);
      if(saving>1e-10) {
    //if(C3.inTheSameTrip(C1)) improved = same_trip_seek(C1,(
 VRPNode)C2,C3,(VRPNode)C4);
    //else if(C3.inTheSameTour(C1)) improved = same_tour_seek
 (C1,(VRPNode)C2,C3,(VRPNode)C4);
    //else //different tour
    improved = diff_tour_seek(C1,C2,C3,C4);
    if(improved) System.out.println("saving="+saving);
      }
      if(improved) break;
  }
    }
    return improved;
}

private boolean global_twopt_step() {
    boolean improved = false;
    for(int i=0;i<tours.size();i++) {
  Tour t = (Tour)tours.get(i);
  VRPNode [] n = t.getNodes();
  for(int j=0;j<n.length;j++) {
      improved = global_twopt_seek(n[j]);
      if(improved) break;
  }
  if(improved) break;
    }
    return improved;
}

public void clean() {
    //System.out.println("enter clean");
    double len1,len2,len3,len4,len5;
```

```
    //do {
    len2 = local_twOpt();
    check_data_consistency("after_local_2opt");
    len3 = chain(1);
    len4 = chain(2);
    //len5 = chain(3);
    len1 = exchange();
    //} while(Math.abs(len1-len5)>1e-10);
    //System.out.println("leave clean");
}
// return true if n1 is ahead of n2, false otherwise
private boolean isAhead(VRPNode n1,VRPNode n2) {
    Tour[] t = (Tour[])tours.toArray(new Tour[0]);
    int[] tid = new int[t.length];
    for(int i=0;i<tid.length;i++) tid[i] = t[i].getId();
    int id1 = n1.getTourId();
    int id2 = n2.getTourId();
    for(int i=0;i<tid.length;i++) {
  if(tid[i]==id1) return true;
  else if(tid[i]==id2) return false;
    }
    return false;
}



private double local_twOpt() {
    Tour[] t = (Tour[])tours.toArray(new Tour[0]);
    for(int i=0;i<t.length;i++)
  t[i].twOpt();
    evaluate();
    return length;
}


// we need to do the following when reverse a chain of nodes
// 1. setup the prior and next pointer for each customer
// 2. set the first and last landfill for each trip
// 3. set the trip/tripid for each node(this is done
 implicitly when adding node to trip)
// 4. update the length and demand for each trip and the tour
// 5. add/delete the nodes for each trip
// 6. change the tours(ArrayList) for proper ordering
private void reverse(VRPNode prior,VRPNode from, VRPNode to,
 VRPNode next) {
```

```
    /*
        int tripId1 = prior.getTripId();
        int tripId2 = next.getTripId();
        int partial_load1 = 0;
        int partial_load2 = 0;
        LandFill lf1 = from.getTrip().getLastLandFill();
        LandFill lf2 = to.getTrip().getFirstLandFill();
    //1. set up the prior and next pointer for nodes
following from and before to
    Trip t1 = prior.getTrip();
    Trip t2 = next.getTrip();
    VRPNode [] n1 = t1.getNodes();
    VRPNode [] n2 = t2.getNodes();

    ArrayList a = new ArrayList();
    ArrayList b = new ArrayList();

    Node itr = from;
    while(itr instanceof VRPNode) {
    Node tmp = ((VRPNode)itr).getNext();
    ((VRPNode)itr).swap();
    a.add((VRPNode)itr);
    t1.delNode((VRPNode)itr);
    //partial_load1+=((VRPNode)itr).getDemand();
    itr = tmp;
    }
    itr = to;
    while(itr instanceof VRPNode) {
    Node tmp = ((VRPNode)itr).getPrior();
    ((VRPNode)itr).swap();
    b.add((VRPNode)itr);
    t2.delNode((VRPNode)itr);
    //partial_load2+=((VRPNode)itr).getDemand();
    itr = tmp;

    }
    prior.setNext(to);
    to.setPrior(prior);
    next.setPrior(from);
    from.setNext(next);

    t1.addNode((VRPNode[])b.toArray(new VRPNode[0]));
    //something could be wrong!
    int asize = a.size();
    VRPNode [] rev = new VRPNode[a.size()];
```

```
    for(int i=asize-1;i>=0;i--) rev[asize-1-i] = (VRPNode)a.
 get(i);
    t2.addNodeBefore(rev);
    // change the landfill for these two trip
    t1.setLastLandFill(lf2);
    t2.setFirstLandFill(lf1);
    t1.evaluate();
    t2.evaluate();
    */
}

public boolean RTR(double record,double deviation) {
    //System.out.println("enter RTR, record="+record+",
 deviation="+deviation);
    /*
        boolean ret = twOpt_RTR(record,deviation);
        ret |= chain_RTR(1,record,deviation);
        if(flag) System.out.println("**********************
 AFTER chain_RTR(1,)**************************");
        check_data_consistency();
        ret  |= exchange_rtr(record,deviation);
        if(flag) System.out.println("**********************
 AFTER exchange_rtr***************************");
        check_length("after rtr");
    //System.out.println("leave RTR");
    */

    return chain_RTR(1,record,deviation) || exchange_rtr(
 record,deviation) || twOpt_RTR(record,deviation);
    //return exchange_rtr(record,deviation);// && chain_RTR
 (1,record,deviation);

}

public Solution improve() {
    int counter=1; int I = 30;
    //Path best_record = (Path)clone();
    //clean();
    LRSolution best_sol =(LRSolution) clone();
    System.out.println("entering improve:"+best_sol);
    double record = best_sol.getLength();
    double deviation = perc * (record-g.getServiceTime());
    //int [] I={30,25,20,15,10,5};
    while(true) {
  boolean improved = false;
```

```
  for(int i=0;i<I;i++) {
      if(RTR(record,deviation)==false) {
    System.out.println("no movement in I loop, quit I loop");
    break;
      }
      else{
    if(length<best_sol.getLength()) {
        System.out.println("[RRT] find a improving move, record
 is "+length);
        improved=true;
        best_sol = (LRSolution) clone();
        record = best_sol.getLength();
        deviation=perc*(record-g.getServiceTime());
    }
      }
  } // I loop
  //System.out.println("counter="+counter);
  //checkDistance("before clean");
  clean();
  check_data_consistency("after clean");
  check_length("after clean");
  //checkDistance("after clean");
  if(length>=best_sol.getLength()) {
      System.out.println("no improvement "+length);
      improved = false;
  }
  else {
      if(Math.abs(length-best_sol.getLength())>1e-10) {
    improved = true;
    best_sol= (LRSolution) clone();
    record = best_sol.getLength();
    deviation=perc*(record-g.getServiceTime());
    System.out.println("after clean best record =="+best_sol.
 getLength());
    counter=0;
      }
  }
  counter++;
  //System.out.println("best record is "+ best_record.
 getLength());
  if(counter>5)
      break;
    }
    return best_sol;
}
```

```
private void safe_check(String where) {
    if(check_feasibility()==false){
  System.out.println(where+"******************DISTANCE_
 CONSTRAINT_VIOLATED**********************");
  dump();
  System.out.println("******************END_DUMP
 ************************");
  System.exit(0);
    }
}


private boolean same_tour_seek(VRPNode C1,VRPNode C2,VRPNode
 C3,VRPNode C4) {
    boolean improved = false;
    return improved;
}

private boolean same_trip_seek(VRPNode C1,VRPNode C2,VRPNode
 C3,VRPNode C4) {
    boolean improved = false;
    return improved;
}
//this chain is acutally a trip
// it might be the first trip(in this case the first landfill
  is a depot
// last trip ( the last landfill is the depot)
// or simply a tour with only one trip(both landfills are
 depot)
private boolean special_chain_seek(Chain chain) {
    check_data_consistency("before_special_chain_seek");
    int nid = 106;
    if(debug) {
  System.out.println(g.getCustomer(nid));
  System.out.println(g.getCustomer(nid).getTour());
    }
    if(debug) System.out.println("enter_special_chain_seek:"+
 chain);
    boolean improved = false;
    VRPNode first = chain.getFirst();
    LandFill prior = (LandFill)first.getPrior();
    int c2 = first.getId();
    VRPNode last = chain.getLast();
    LandFill next = (LandFill)last.getNext();
```

```
   for(int m=0;m<g.NBCount[c2];m++) {
 int c3 = g.NBList[c2][m];
 VRPNode C3 = g.getCustomer(c3);
 Node C4 = C3.getNext();
 //Node prior = first.getPrior();
 /*
    System.out.println("prior:"+prior);
    System.out.println("next:"+next);
    System.out.println("C3:"+C3);
    System.out.println("C4:"+C4);
    */
 // prior-->first.....last->next.......C3->C4
 // prior->next......C3->first.....->last->C4

 boolean cond = true;
 if(C4 instanceof VRPNode) {
     if(chain.contains((VRPNode)C4)) cond = false;
 }
 // since this is moving the whole trip, we might first
check
 // the capacity constraint cause this is most likely to be
violated
 if(C3.getTrip().getDemand()+chain.getDemand()>g.getCapacity
()) continue;
 if((!chain.contains(C3)) && cond ) {
     double saving;
     if(prior.isDepot() && next.isDepot()) {
   if(debug) {
       System.out.println("find a single tour with a single
trip");
   }
   saving = prior.getDist2Node(first)+last.getDist2Node(next
)+C3.getDist2Node(C4);
   saving -= (C3.getDist2Node(first)+last.getDist2Node(C4));
   double newLength;
   if(C3.getTourId()==first.getTourId()) {
       System.out.println("**************************FATAL
ERROR IN SPECIAL_CHAIN_SEEK");
       newLength = C3.getTour().getLength()-saving;
   }

   else {
       newLength = C3.getTour().getLength()+chain.getLength
()+C3.getDist2Node(first)+
     last.getDist2Node(C4)-C3.getDist2Node(C4);
```

```
    }
    if(saving>1e-10 && newLength<=g.getDisConstraint()) {
        if(C3.getTrip().getDemand()+chain.getDemand()<=g.
getCapacity()){
      System.out.println("[special chain seek single trip]
removing one tour saving "+saving);
      System.out.println("[special chain seek] chain="+chain);
      C3.setNext(first);
      first.setPrior(C3);
      last.setNext(C4);
      if(C4 instanceof VRPNode) ((VRPNode)C4).setPrior(last);
      Trip t = first.getTrip();
      Trip s = C3.getTrip();
      t.clearNode();
      delTour(t.getTour());
      s.addNode(chain.getNodes());
      s.rebuildTrip();
      s.getTour().evaluate();
      evaluate();
      improved = true;
        }
    }
      }
      else { // the chain has at least one trip in the same
tour
    if(prior.isDepot()) {
        Trip t = first.getTrip();
        Tour w = first.getTour();
        VRPNode x = w.getNodeAfter(t);
        saving = prior.getDist2Node(first)+last.getDist2Node(
next)+C3.getDist2Node(C4);
        saving -= (prior.getDist2Node(x)+ C3.getDist2Node(
first)+last.getDist2Node(C4));
        double newLength1,newLength2;

        if(C3.getTourId()==first.getTourId()) {
      newLength1 = C3.getTour().getLength()-saving;
      newLength2 = newLength1;
        }
        // newlength1 C3->first->...->last->C4
        // newlength2 prior->chain->next->x==>prior->x
        else {
      newLength1 = C3.getTour().getLength()+chain.getLength()
+C3.getDist2Node(first)+
          last.getDist2Node(C4)-C3.getDist2Node(C4);
```

```
      newLength2 = w.getLength()-prior.getDist2Node(first)-
chain.getLength()
          -last.getDist2Node(next)-next.getDist2Node(x)+prior
.getDist2Node(x);
        }
        if(saving>1e-10 && newLength1<=g.getDisConstraint()&&
newLength2<=g.getDisConstraint() ) {
      if(debug) System.out.println("[special chain seek first
 depot] removing one trip saving "+saving);
      C3.setNext(first);
      first.setPrior(C3);
      last.setNext(C4);
      if(C4 instanceof VRPNode) ((VRPNode)C4).setPrior(last);
      w.delTrip(t);
      x.setPrior(prior);
      x.getTrip().setFirstLandFill(prior);
      x.getTrip().evaluate();
      w.evaluate();
      Trip s = C3.getTrip();
      s.addNode(chain.getNodes());
      s.rebuildTrip();
      s.getTour().evaluate();
      evaluate();
      improved = true;
        }
    }
    else {
        if(next.isDepot()) {
      if(debug) System.out.println("next is a depot, chain="+
chain+",C3="+C3.getId()+",C4="+C4.getId()+" in tour "+C3.
getTour().getId());
      //dump();
      Trip t = first.getTrip();
      Tour w = first.getTour();
      VRPNode x = w.getNodeBefore(t);
      saving = prior.getDist2Node(first)+last.getDist2Node(
next)+C3.getDist2Node(C4);
      saving -= (x.getDist2Node(next)+ C3.getDist2Node(first)
+last.getDist2Node(C4));
      if(debug) System.out.println("saving="+saving);
      double newLength1,newLength2;

      if(C3.getTourId()==first.getTourId()) {
          newLength1 = C3.getTour().getLength()-saving;
          newLength2 = newLength1;
```

```
      }
      //newLength2 x->prior->chain->next ==>x->next
      else {
          newLength1 = C3.getTour().getLength()+chain.
getLength()+C3.getDist2Node(first)+
        last.getDist2Node(C4)-C3.getDist2Node(C4);
          newLength2 = w.getLength()-prior.getDist2Node(first
)-chain.getLength()
        -last.getDist2Node(next)-x.getDist2Node(prior)+x.
getDist2Node(next);
      }

      if(saving>1e-10 && newLength1<=g.getDisConstraint()&&
newLength2<=g.getDisConstraint()) {
          if(debug) {
        System.out.println("[special chain seek last depot]
removing one trip saving "+saving);
        System.out.println("C3="+C3.getId()+",C4="+C4.getId()
+",x="+x.getId()+",chain="+chain);
          }
          //System.out.println("before");
          //dump();
          C3.setNext(first);
          first.setPrior(C3);
          last.setNext(C4);
          if(C4 instanceof VRPNode) ((VRPNode)C4).setPrior(
last);
          w.delTrip(t);
          x.setNext(next);
          x.getTrip().setLastLandFill(next);
          x.getTour().evaluate();
          w.evaluate();
          Trip s = C3.getTrip();
          s.addNode(chain.getNodes());
          s.rebuildTrip();
          s.getTour().evaluate();
          evaluate();
          //System.out.println("after");dump();
          improved = true;
      }
        }
        else {//now both of prior and next are real landfill
      // x->prior->chain->next->y ==> x->z->y
      Trip t = first.getTrip();
      Tour w = first.getTour();
```

```
        VRPNode x = w.getNodeBefore(t);
        VRPNode y = w.getNodeAfter(t);
        LandFill z = g.getOptLandFill(x,y);

        saving = prior.getDist2Node(first)+last.getDist2Node(
next)+C3.getDist2Node(C4)
            + x.getDist2Node(prior) + next.getDist2Node(y);
        saving -= (x.getDist2Node(z)+z.getDist2Node(y)+C3.
getDist2Node(first)+last.getDist2Node(C4));

        double newLength1,newLength2;

        if(C3.getTourId()==first.getTourId()) {
            newLength1 = C3.getTour().getLength()-saving;
            newLength2 = newLength1;
        }
        //newLength2 x->prior->chain->next->y ==>x->z->y
        else {
            newLength1 = C3.getTour().getLength()+chain.
getLength()+C3.getDist2Node(first)+
          last.getDist2Node(C4)-C3.getDist2Node(C4);
            newLength2 = w.getLength()-prior.getDist2Node(first
)-chain.getLength()
         -last.getDist2Node(next)-x.getDist2Node(prior)-next.
getDist2Node(y)
         +x.getDist2Node(z)+z.getDist2Node(y);
        }
        if(saving >1e-10 && newLength1<=g.getDisConstraint() &&
 newLength2<=g.getDisConstraint()) {
            if(debug) System.out.println("[special chain seek
middle trip]removing one trip saving "+saving);
            x.getTrip().setLastLandFill(z);
            y.getTrip().setFirstLandFill(z);
            x.setNext(z); y.setPrior(z);
            C3.setNext(first);
            first.setPrior(C3);
            last.setNext(C4);
            if(C4 instanceof VRPNode) ((VRPNode)C4).setPrior(
last);
            x.getTrip().evaluate(); y.getTrip().evaluate();
            w.delTrip(t);
            w.evaluate();
            Trip s = C3.getTrip();
            s.addNode(chain.getNodes());
            s.rebuildTrip();
```

```
            s.getTour().evaluate();
            evaluate();
            improved = true;
        }
          }
      }
        }
    }
    if(improved) break;
      }
      //if(improved) dump();
      //System.out.println("leaving special_chain_seek"+chain);
      check_data_consistency("after special_chain_seek");
      return improved;
}

private boolean special_chain_seek_rtr(Chain chain,double
 record,double deviation) {
      return false;
}

public String toString() {
      StringBuffer sb = new StringBuffer();
      sb.append("the solution has "+ tours.size()+" tours.");
      sb.append("\nThe total traveling distance is " +length);
      return sb.toString();
}


public double twOpt() {
      double len1=0D,len2=0D;
      //len1 = global_twOpt();
      len2 = local_twOpt();
      //System.out.println("finish 2opt");
      return length;
}

private boolean check_length(String where) {
      Tour[] t = getTours();
      for(int i=0;i<t.length;i++) {
    if(t[i].check_length()==false) {
        System.out.println(where);
        return false;
    }
      }
```

```
    return true;
}

private boolean twOpt_RTR(double record,double deviation) {
    //System.out.println("dong 2opt RTR");
    boolean moved = false;
    setRTR();

    int count=0;
    while(count<g.size()) {
  Tour[] t = getTours();
  for(int i=0;i<t.length;i++) {
      VRPNode[] v = t[i].getNodes();
      for(int j=0;j<v.length;j++) {
    if(v[j].getRTR()==true) {
        count++;
        v[j].setRTR(false);
        if(twOpt_seek_rtr(v[j],record,deviation)) moved =
 true;
    }
      }
  }
    }


    /*
        int [] seq = rand.getRandPerm(g.size());
        for(int i=0;i<g.size();i++) {
        VRPNode v = g.getCustomer(seq[i]);
        if(twOpt_seek_rtr(v,record,deviation)) moved=true;

        }
        */
    safe_check("2OPT-RTR");
    //System.out.println("finishing 2opt RTR");
    return moved;
}

//C1->C2.....C4->C3
// NT: speciall case when C1 and C4 are both landfill
private boolean twOpt_seek_rtr(VRPNode C2,double record,
 double deviation) {
    boolean moved = false;
    //System.out.println("2opt-rtr "+ C2.getId());
    int c2 = C2.getId();
```

```
   Node C1 = C2.getPrior();
   //if(C2==null) System.out.println("C2 is null");
   //else System.out.println("C2="+C2.getId()+"C2.class="+C2
.getClass());
   int c1 = C1.getId();
   VRPNode max_C3=null;
   double max_saving = -1e10;
   if(true) {
 double d12 = C1.getDist2Node(C2);//g.distance(c1,c2);
 //System.out.println("try "+g.NBCount[c2]+" nodes");
 for(int m=0;m<g.NBCount[c2];m++) {
     int c3 = g.NBList[c2][m];
     VRPNode C3 = g.getCustomer(c3);
     //if(d23>=d12) break;
     if(C3.equals(C1)) continue;
     if(C3.inTheSameTour(C2)) {
   Node C4 = C3.getPrior();
   int c4 = C4.getId();
   double d43 = C4.getDist2Node(C3);
   if(C2.equals(C4)) continue;
   if(!(C1 instanceof LandFill && C4 instanceof LandFill)) {
       if(C3.inTheSameTrip(C2)) {
     if(twOpt_rtr_isAhead(C2,C3)) {
         // C1->C2...C4->C3 ==> C1->C4...C2->C3
         double d23 = C2.getDist2Node(C3);
         double d14 = C1.getDist2Node(C4);
         double saving = d12 + d43 - d23 - d14;
         if(saving>max_saving) {
       if(length-saving<=record+deviation) {
           if(C3.getTour().getLength()-saving<=g.
getDisConstraint()) {//distance constraint
         max_saving = saving;
         max_C3 = C3;
           }
       }
         }
     }
     else { // C4->C3...C1->C2==>C4->C1...C3->C2
         double d41 = C4.getDist2Node(C1);
         double d32 = C3.getDist2Node(C2);
         double saving = d12 + d43 - d41 - d32;
         if(saving>max_saving) {
       if(length-saving<=record+deviation) {
           if(C3.getTour().getLength()-saving<=g.
getDisConstraint()) {//distance constrainth
```

```
            max_saving = saving;
            max_C3 = C3;
              }
          }
            }
        }
          }
          else { // now C2 and C3 are in different trips
        if(twOpt_rtr_difftrip_isAhead(C2,C3)){
            //C1->C2...C4->C3=>C1->C4...C2->C3
            double d23 = C2.getDist2Node(C3);
            double d14 = C1.getDist2Node(C4);
            double saving = d12 + d43 - d23 - d14;
            if(saving>max_saving) {
          if(length-saving<=record+deviation) {
              if(twOpt_rtr_check_feasibility(C1,C2,C4,C3)) {
            max_saving = saving;
            max_C3 = C3;
              }
          }
            }
        }
        else {
            double d41 = C4.getDist2Node(C1);
            double d32 = C3.getDist2Node(C2);
            double saving = d12 + d43 - d41 - d32;
            if(saving>max_saving) {
          if(length-saving<=record+deviation) {
              if(twOpt_rtr_check_feasibility(C4,C3,C1,C2)){
            max_saving = saving;
            max_C3 = C3;
              }
          }
            }
        }
          }
      }
        }
    }
      }

      if(max_C3!=null) {
    Node max_C4 = max_C3.getPrior();
    //System.out.println("C1.class="+C1.getClass()+",C2.class
    ="+C2.getClass()+",C4.class="+max_C4.getClass()+",C3.class
```

```
=”+max_C3.getClass());
 //System.out.println(”doing 2opt-rtr(”+C1.getId()+”->”+C2.
getId()+”,”+max_C4.getId()+”->”+max_C3.getId()+”)”);
 if(max_C3.inTheSameTrip(C2)) {
     if(twOpt_rtr_isAhead(C2,max_C3)) {
   //System.out.println(”C2 is ahead of C3”);
   //System.out.println(C2.getTour());
   //System.out.println(”doing 2opt sametrip reverse”);
   twOpt_rtr_sametrip_reverse(C1,C2,(VRPNode)max_C4,max_C3);
     }
     else {
   //System.out.println(”C2 is behind C3”);
   //System.out.println(C2.getTour());
   twOpt_rtr_sametrip_reverse(max_C4,max_C3,(VRPNode)C1,C2);
     }
 }
 else {
     if(twOpt_rtr_difftrip_isAhead(C2,max_C3)){
   //System.out.println(”difftour C2 is ahead of C3”);
   //System.out.println(C2.getTour());
   twOpt_rtr_difftrip_reverse(C1,C2,max_C4,max_C3);
     }
     else{
   //System.out.println(”difftour C2 is behind C3”);
   //System.out.println(C2.getTour());
   twOpt_rtr_difftrip_reverse(max_C4,max_C3,C1,C2);
     }
 }
 C2.getTour().evaluate();
 evaluate();
 //dump();
 moved = true;
 //System.out.println(”finish 2opt-rtr ”+C1.getClass()+”,”+
C2.getClass()+”,”+max_C3.getClass());
   }
   return moved;
}

private boolean twOpt_rtr_isAhead(VRPNode n1,VRPNode n2) {
   VRPNode tag = n2;
   do {
 Node prior = tag.getPrior();
 if(prior instanceof LandFill) {
     return false;
 }
```

```
  else
      tag =(VRPNode) prior;
    }while(!tag.equals(n1));
    return true;
}

private boolean twOpt_rtr_difftrip_isAhead(VRPNode n1,
 VRPNode n2) {
    int tid1 = n1.getTripId();
    int tid2 = n2.getTripId();
    if(tid1==tid2) System.out.println("fatal error");
    if(n1.getTourId()!=n2.getTourId()) System.out.println("
 also fatal error");
    Tour t = n1.getTour();
    Trip[] trips = t.getTrips();
    for(int i=0;i<trips.length;i++) {
  if(trips[i].getId()==tid1) return true;
  if(trips[i].getId()==tid2) return false;
    }
    System.out.println("never be here");
    return false;
}

private void twOpt_rtr_sametrip_reverse(Node prior, VRPNode
 from, VRPNode to,Node next) {
    Trip t = from.getTrip();
    if(prior instanceof VRPNode) ((VRPNode)prior).setNext(to)
 ;
    if(next instanceof VRPNode)  ((VRPNode)next).setPrior(
 from);
    VRPNode itr = from;
    while(!itr.equals(to)) {
  VRPNode tmp = itr;
  itr=(VRPNode)itr.getNext();
  tmp.swap();
    }
    to.swap();
    to.setPrior(prior);
    from.setNext(next);
    t.rebuildTrip();
}

//prior->from->..lf1->...o->..->lf2->....->to->next
private void twOpt_rtr_difftrip_reverse(Node prior, VRPNode
 from, Node to, Node next) {
```

```
    if(debug) {
 System.out.println("doing 2opt-rtr-difftrip-reverse");
 System.out.println("prior="+prior.getId()+" class="+prior.
getClass()+",from="+from.getId()+" class="+from.getClass()+",
to="+to.getId()+" class="+to.getClass()+",next="+next.getId()
+" class="+next.getClass());
    }
    LandFill lf1=null,lf2=null;
    int tripId1=-1,tripId2=-1;
    Trip t1=null,t2=null;
    if(to instanceof VRPNode) {
 tripId1 = from.getTripId();
 tripId2 = ((VRPNode)to).getTripId();
 lf1 = from.getTrip().getLastLandFill();
 lf2 = ((VRPNode)to).getTrip().getFirstLandFill();
 t1 = from.getTrip();
 t2 = ((VRPNode)to).getTrip();
    }
    else {
 tripId1 = from.getTripId();
 tripId2 = ((VRPNode)next).getTripId();
 lf1 = from.getTrip().getLastLandFill();
 lf2 = ((VRPNode)next).getTrip().getFirstLandFill();
 t1 = from.getTrip();
 t2 = ((VRPNode)next).getTrip();
    }
    if(tripId1==tripId2) System.out.println("FATAL ERROR,2opt
-difftrip-reverse");

    //1. set up the prior and next pointer for nodes
following from and before to
    VRPNode [] n1 = t1.getNodes();
    VRPNode [] n2 = t2.getNodes();

    ArrayList a = new ArrayList();
    ArrayList b = new ArrayList();

    Node itr = from;
    while(itr instanceof VRPNode) {
 Node tmp = ((VRPNode)itr).getNext();
 ((VRPNode)itr).swap();
 a.add((VRPNode)itr);
 t1.delNode((VRPNode)itr);
 //partial_load1+=((VRPNode)itr).getDemand();
 itr = tmp;
```

```
    }
    itr = to;
    while(itr instanceof VRPNode) {
  Node tmp = ((VRPNode)itr).getPrior();
  ((VRPNode)itr).swap();
  b.add((VRPNode)itr);
  t2.delNode((VRPNode)itr);
  //partial_load2+=((VRPNode)itr).getDemand();
  itr = tmp;

    }
    if(prior instanceof VRPNode) ((VRPNode)prior).setNext(to)
;
    if(to instanceof VRPNode)  ((VRPNode)to).setPrior(prior);
    if(next instanceof VRPNode)  ((VRPNode)next).setPrior(
from);
    from.setNext(next);

    t1.addNode((VRPNode[])b.toArray(new VRPNode[0]));
    t1.rebuildTrip();
    t2.addNode((VRPNode[])a.toArray(new VRPNode[0]));
    t2.rebuildTrip();
    // change the landfill for these two trip
    t1.setLastLandFill(lf2);
    t2.setFirstLandFill(lf1);

    // if there are trips between t1 and t2 we need to
reverse them also
    Trip[] t = from.getTour().getTrips();
    int i1=-1,i2=-1;
    for(int i=0;i<t.length;i++){
  if(tripId1==t[i].getId()) i1 = i;
  if(tripId2==t[i].getId()) {
      i2 = i;
      break;
  }
    }
    if(i1<0 || i2<0) System.out.println("fatal error here");
    for(int i=i1+1;i<i2;i++) t[i].swap();
    for(int i=i1+1,j=i2-1;i<j;i++,j--) {
  Trip tmp = t[i];
  t[i] = t[j];
  t[j] = tmp;
    }
    from.getTour().evaluate();
```

```
        evaluate();
        //System.out.println("finish 2opt-difftrip-reverse");
    }

    private boolean twOpt_rtr_check_feasibility(Node prior,
     VRPNode from, Node to,Node next) {
        int partial_load1 = 0;
        int partial_load2 = 0;
        Node itr = from;
        while(itr instanceof VRPNode) {
      partial_load1+=((VRPNode)itr).getDemand();
      itr = ((VRPNode)itr).getNext();
        }
        itr = to;
        while(itr instanceof VRPNode) {
      partial_load2+=((VRPNode)itr).getDemand();
      itr = ((VRPNode)itr).getPrior();
        }

        //first check capacity constraint
        int demand1,demand2;
        demand1 = from.getTrip().getDemand();
        if(to instanceof VRPNode) demand2 = ((VRPNode)to).getTrip
     ().getDemand();
        else demand2 = ((VRPNode)next).getTrip().getDemand();

        if(demand1-partial_load1+partial_load2<=g.getCapacity()
     &&
           demand2-partial_load2+partial_load1<=g.getCapacity())
      return true;
        else return false;


    }

    public void check_data_consistency(String where){
        if(debug) System.out.println("checking data consistent at
     "+ where);
        Tour [] t = getTours();
        for(int i=0;i<t.length;i++) t[i].check_data_consistency()
     ;
        //dump();
    }

}
```

E.4 Tour.java

```
package edu.umd.math.lify.vrp;
import edu.umd.math.lify.util.*;
import java.util.*;
import edu.umd.math.lify.graph.*;

public class Tour implements Solution,Cloneable{
  private boolean debug = false;
  private VRPGraph g;
  private int numTrips;
  private ArrayList trips;
  private double length;
  private int demand;
  private int tourId;
  static int GLOBAL_ID=0;

  public Tour(Trip[] t) {
    trips = new ArrayList();
    for(int i=0;i<t.length;i++){
      addTrip(t[i]);
    }
    tourId = GLOBAL_ID++;
    numTrips = t.length;
    evaluate();

  }

  public void setGraph(VRPGraph graph) { g = graph; }

  public Object clone() {
  //  System.out.println("Tour.clone is called");
    Tour copy = null;
    try {
      copy = (Tour)super.clone();
    } catch(CloneNotSupportedException e) {System.out.println(e
   );}
    copy.trips = (ArrayList)trips.clone();
    Trip[] t = (Trip[])trips.toArray(new Trip[0]);
    Trip [] copyTrip = new Trip[t.length];
    for(int i=0;i<t.length;i++) copyTrip[i] =(Trip) t[i].clone
   ();
    copy.trips = new ArrayList();
    for(int i=0;i<t.length;i++) {
      copy.addTrip(copyTrip[i]);
```

```
    }
    copy.evaluate();
    return copy;
  }

  public double evaluate() {
    length=0D;
    demand = 0;
    for(int i=0;i<trips.size();i++){
      Trip t = (Trip)trips.get(i);
      //length+= t.getLength();
      length+= t.evaluate();
      demand += t.getDemand();
    }
    return length;
  }

  public void delTrip(Trip t) {
    remove(t);
    evaluate();
  }

  public void delTrip(Trip[] t) {
    for(int i=0;i<t.length;i++) delTrip(t[i]);
  }

  public VRPNode getNodeAfter(Trip t) {
    int i = trips.indexOf(t);
    return ((Trip)trips.get(i+1)).getFirstNode();
  }

  public VRPNode getNodeBefore(Trip t) {
    int i = trips.indexOf(t);
    return ((Trip)trips.get(i-1)).getLastNode();
  }

  public int getDemand() { return demand; }
  public int getId() { return tourId; }
  public Trip[] getTrips() {return (Trip[])trips.toArray(new
   Trip[0]); }
  public VRPNode [] getNodes() {
    ArrayList nodes = new ArrayList();
    for(int i=0;i<trips.size();i++) {
      VRPNode []v = ((Trip)trips.get(i)).getNodes();
      for(int j=0;j<v.length;j++) nodes.add(v[j]);
```

```
  }
  return (VRPNode[]) nodes.toArray(new VRPNode[0]);
}
//usually when we remove a trip,the tour containing it
// should be not used.
public void remove(Trip t) {
  if(!trips.contains(t)) {
    System.out.println("OOPS!FATAL ERROR in Tour.remove");
    System.exit(1);
  }
  trips.remove(trips.indexOf(t));
  numTrips--;
}

public void addTrip(Trip t) {
  trips.add(t);
  t.setTour(this);
  numTrips++;
  //now is inefficient implementation
  //evaluate();
  demand+=t.getDemand();
  length+=t.getLength();

}

public void addTrip(Trip[] t) {
  for(int i=0;i<t.length;i++) addTrip(t[i]);
}

public Trip getLastTrip() { return (Trip) trips.get(trips.
 size()-1); }
public boolean isEmpty() { return trips.size()==0; }
/**
 * size return the # of trips in the tour
 */
public int size() {
  return trips.size();
}

/**
 * return the # of customers in the tour
 */
public int getNumNodes() {
  int ret = 0;
  Trip [] t = getTrips();
```

```
  for(int i=0;i<t.length;i++) ret+=t[i].size();
  return ret;
}

public void makeNull() {
  trips.clear();
  demand = 0;
  length = 0D;
}

public String toString() {
  StringBuffer sb = new StringBuffer();
  sb.append("Tour("+tourId+")"+" total demand="+demand+",
 total length="+length+"\n");
  for(int i=0;i<trips.size();i++)
    sb.append(trips.get(i).toString()+"\n");

  //sb.append("("+demand+","+len+")");
  return sb.toString();
}

public double getLength() {
    /*
  double oldLen = length;
  evaluate();
  if(length!=oldLen) System.out.println("tour length is not
 consistent on Tour("+getId()+")!oldLen="+oldLen+",newLen="+
 length);
  */
    return length;
}

// precondition: other is a tour starting with a landfill
 instead of depot
// i.e. need to fix

public boolean check_length() {
  double oldLen = length;
  evaluate();
  if(length!=oldLen){
    System.out.println("tour length is not consistent on Tour
 ("+getId()+")!oldLen="+oldLen+",newLen="+length);
    return false;
  }
  else return true;
```

```
}

public void merge(Tour other,LandFill landfill) {
  contenateTour(other,landfill);
  Trip[] newTrip = other.getTrips();
  for(int i=0;i<newTrip.length;i++)
    addTrip(newTrip[i]);
  evaluate();
  other.makeNull();
}

private void contenateTour(Tour other,LandFill landfill) {
  VRPNode lastnode = getLastTrip().getLastNode();
  getLastTrip().setLastLandFill(landfill);
  lastnode.setNext(landfill);
  other.getFirstTrip().setFirstLandFill(landfill);
  other.getFirstTrip().getFirstNode().setPrior(landfill);
  getLastTrip().evaluate();
  other.getFirstTrip().evaluate();
}

public Trip getFirstTrip() {
  return (Trip) trips.get(0);
}

private double local_twopt() {
  //now we only do 2opt within each trip;
  Trip[] t = (Trip[])trips.toArray(new Trip[0]);
  for(int i=0;i<t.length;i++)
    t[i].twOpt();
  //evaluate();
  chooseOptLandFill();
  evaluate();
  return length;
}
private double global_twopt() {
  while(global_twopt_step()) ;
  return length;
}

private boolean global_twopt_step() {
  boolean improved = false;
  for(int i=0;i<trips.size();i++) {
    Trip t = (Trip)trips.get(i);
    g = t.getGraph();
```

```
    VRPNode [] n = t.getNodes();
    for(int j=0;j<n.length;j++) {
      improved = seek(n[j]);
      if(improved) break;
    }
    if(improved) break;
  }
  return improved;
}

private boolean seek(VRPNode C2) {
  if(debug) System.out.println("seeking "+C2.getId());
  boolean improved = false;
  int c2 = C2.getId();
  Node C1 = C2.getPrior();
  int c1 = C1.getId();
  double d12 = C1.getDist2Node(C2);//g.distance(c1,c2);
  for(int m=0;m<g.NBCount[c2];m++) {
    //System.out.println("m="+m+",c3="+g.NBList[c2][m]);
    int c3 = g.NBList[c2][m];
    VRPNode C3 = g.getCustomer(c3);
    double d23 = g.distance(c2,c3);
    //if(d23>=d12) break;
    if(C3.equals(C1)) continue;
    if(C3.inTheSameTour(C2)) {
      Node C4 = C3.getPrior();
      int c4 = C4.getId();
      if(C2.equals(C4)) continue;
      if(!(C1 instanceof LandFill && C4 instanceof LandFill))
  {
        if(C3.inTheSameTrip(C2)){
          improved=same_trip_seek(C1,C2,C3,C4);
          //if(improved)
        //    System.out.println("[2-opt same trip] saving "+
 saving);
        }
        else {
          improved = diff_trip_seek(C1,C2,C3,C4);
          //if(improved)
        //    System.out.println("[tour 2-opt diff trip]
 saving "+saving);
        }
      }
      if(improved){
        evaluate();
```

```
          break;
        }
      }
    }
    return improved;
}
/*
            double d34 = C3.getDist2Node(C4);//g.distance(c3,c4
 );
            double d14 = C1.getDist2Node(C4);//g.distance(c1,c4
 );
            //System.out.println("d12="+d12+",d34="+d34+",d23
 ="+d23+",d14="+d14);
            double saving = d12+d34-d14-d23;
            System.out.println("saving="+saving);
            if(saving>1e-6) {
            }

            if(improved) {
              evaluate();
              break;
            }
          }
        }
      }
    }
    System.out.println("finish seek"+C2.getId());
    return improved;
}
*/
// precodition : C1->C2, C4->C3
public boolean same_trip_seek(Node C1,VRPNode C2,VRPNode C3,
 Node C4) {
    boolean improved = false;
    double d12 = C1.getDist2Node(C2);
    double d43 = C4.getDist2Node(C3);
    int c1 = C1.getId();int c2=C2.getId();int c3 = C3.getId();
 int c4=C4.getId();
    if(same_trip_isAhead(C2,C3)){
      double d14 = C1.getDist2Node(C4);
      double d23 = C2.getDist2Node(C3);
      double saving = d12+d43-d14-d23;
      if(saving>1e-10) {
        if(same_trip_check_feasibility(C1,C2,C4,C3)) {
          //debug=true;
```

```
        if(debug) {
          System.out.println("C2("+c2+") is ahead of C3("+c3+
")");
          System.out.print("[same_trip] before reverse ");
          System.out.println("("+c1+","+c2+","+c4+","+c3+")")
;
          System.out.println(this);
        }
        same_trip_reverse(C1,C2,(VRPNode)C4,C3);
        if(debug) {
          System.out.println("[same_trip] after reverse");
          System.out.println(this);
        }
        improved = true;
        debug=false;
      }
    }
  }
  else {
    double d41 = C4.getDist2Node(C1);
    double d32 = C3.getDist2Node(C2);
    double saving = d12+d43 - d41-d32;
    if(saving>1e-10) {
      if(same_trip_check_feasibility(C4,C3,C1,C2)) {
        //debug=true;
        if(debug) {
          System.out.println("C2("+c2+") is behind of C3("+c3
+")");
          System.out.print("[same_trip] before reverse ");
          System.out.println("("+c4+","+c3+","+c1+","+c2+")")
;
          System.out.println(this);
        }
        same_trip_reverse(C4,C3,(VRPNode)C1,C2);
        if(debug) {
          System.out.println("[same_trip] after reverse ");
          System.out.println("("+c4+","+c3+","+c1+","+c2+")")
;
          System.out.println(this);
        }
        improved = true;
        debug=false;
      }
    }
  }
```

```
  return improved;
}

private boolean same_trip_isAhead(VRPNode n1,VRPNode n2) {
  VRPNode tag = n2;
  do {
    Node prior = tag.getPrior();
    if(prior instanceof LandFill) {
      return false;
    }
    tag =(VRPNode) prior;
  }while(!tag.equals(n1));
  return true;
}

// since we are in the same trip, the capacity constraint
 will be alwasy satisfied
// the only thing we need to consider is the max trip length
 if any
private boolean same_trip_check_feasibility(Node prior,
 VRPNode from, Node to, Node next) {
  return true;
}

private void same_trip_reverse(Node prior,VRPNode from,
 VRPNode to, Node next) {
  Trip t = from.getTrip();
  if(prior instanceof VRPNode) ((VRPNode)prior).setNext(to);
  if(next instanceof VRPNode)  ((VRPNode)next).setPrior(from)
 ;
  VRPNode itr = from;
  while(!itr.equals(to)) {
    VRPNode tmp = itr;
    itr=(VRPNode)itr.getNext();
    tmp.swap();
  }
  to.swap();
  /*
  System.out.println("after swap");
  for(int i=0;i<stop.size();i++) {
    VRPNode n = (VRPNode)stop.get(i);
    System.out.println(n.getPrior().getId());
    System.out.println(n.getNext().getId());
  }
  */
```

```
    to.setPrior(prior);
    from.setNext(next);
    t.rebuildTrip();
    evaluate();
    //System.out.println("after reverse:");
    //System.out.println(this);
  }

  //precodition: C1->C2, C4->C3
  public boolean diff_trip_seek(Node C1,VRPNode C2,VRPNode C3,
   Node C4) {
    boolean improved = false;
    double d12 = C1.getDist2Node(C2);
    double d43 = C4.getDist2Node(C3);
    int c1 = C1.getId();int c2=C2.getId();int c3 = C3.getId();
   int c4=C4.getId();
    if(diff_trip_isAhead(C2,C3)){
      // C1->C2....C4->C3==> C1->C4...C2->C3
      double d14 = C1.getDist2Node(C4);
      double d23 = C2.getDist2Node(C3);
      double saving = d12+d43-d14-d23;
      if(saving>1e-10) {
        if(diff_trip_check_feasibility(C1,C2,C4,C3)) {
          if(debug) {
            System.out.print("[diff_trip]before_reverse_");
            System.out.println("("+c1+","+c2+","+c4+","+c3+")")
   ;
            System.out.println(this);
          }
          diff_trip_reverse(C1,C2,C4,C3);
          if(debug) {
            System.out.println("[diff_trip]after_reverse");
            System.out.println(this);
          }
          improved = true;
        }
      }
    }
    else {
      // C4->C3...C1->C2 ==> C4->C1...C3->C2
      double d41 = C4.getDist2Node(C1);
      double d32 = C3.getDist2Node(C2);
      double saving = d12+d43 - d41-d32;
      if(saving>1e-10) {
        if(diff_trip_check_feasibility(C4,C3,C1,C2)) {
```

```
        if(debug) {
          System.out.print("[diff_trip]before reverse ");
          System.out.println("("+c4+","+c3+","+c1+","+c2+")")
 ;
        }
        diff_trip_reverse(C4,C3,C1,C2);
        if(debug) {
          System.out.println("[diff_trip]after reverse ");
          System.out.println(this);
        }
        improved = true;
      }
    }
  }
  return improved;
}

// assume the trips are properly ordered in the ArrayList
// return true if n1 is ahead of n2, false otherwise
public boolean diff_trip_isAhead(VRPNode n1,VRPNode n2) {
  Trip[] t = (Trip[])trips.toArray(new Trip[0]);
  int[] tid = new int[t.length];
  for(int i=0;i<tid.length;i++) tid[i] = t[i].getId();
  int id1 = n1.getTripId();
  int id2 = n2.getTripId();
  for(int i=0;i<tid.length;i++) {
    if(tid[i]==id1) return true;
    else if(tid[i]==id2) return false;
  }
  return false;
}

// precodition: the orientation is prior->from->...->to->next
// after this procedure, the tour becomes prior->to->..->from
 ->next
public boolean diff_trip_check_feasibility(Node prior,VRPNode
  from, Node to, Node next) {
  int partial_load1 = 0;
  int partial_load2 = 0;
  Node itr = from;
  while(itr instanceof VRPNode) {
    partial_load1+=((VRPNode)itr).getDemand();
    itr = ((VRPNode)itr).getNext();
  }
  itr = to;
```

```
  while(itr instanceof VRPNode) {
    partial_load2+=((VRPNode)itr).getDemand();
    itr = ((VRPNode)itr).getPrior();
  }

  //first check capacity constraint
  int demand1,demand2;
  demand1 = from.getTrip().getDemand();
  if(to instanceof VRPNode) demand2 = ((VRPNode)to).getTrip()
.getDemand();
  else demand2 = ((VRPNode)next).getTrip().getDemand();

  if(demand1-partial_load1+partial_load2<=g.getCapacity() &&
    demand2-partial_load2+partial_load1<=g.getCapacity())
    return true;
  else return false;

}
  /*
  //System.out.println("diff_trip_check_feasibility("+prior.
getId()+","+from.getId()+","+to.getId()+","+next.getId());
  if(prior==null || next==null) System.out.println("null here
");
  if(from==null || to==null) System.out.println("null here
0");
  System.out.flush();
  int tripId1 = prior.getTripId();
  int tripId2 = next.getTripId();
  int partial_load1 = 0;
  int partial_load2 = 0;
  Node itr = from;
  while(itr instanceof VRPNode) {
    partial_load1+=((VRPNode)itr).getDemand();
    itr = ((VRPNode)itr).getNext();
    if(itr==null) {
      System.out.println("null here 1");
      System.out.flush();
    }
  }
  itr = to;
  while(itr instanceof VRPNode) {
    partial_load2+=((VRPNode)itr).getDemand();
    itr = ((VRPNode)itr).getPrior();
    if(itr==null) {
      System.out.println("null here 2");
```

```
        System.out.flush();
      }
    }
    System.out.flush();
    if(from.getTrip()==null || to.getTrip()==null) System.out.
   println("null here 3");
    int demand1 = from.getTrip().getDemand();
    int demand2 = to.getTrip().getDemand();
    if(demand1-partial_load1+partial_load2<=g.getCapacity() &&
      demand2-partial_load2+partial_load1<=g.getCapacity())
      return true;
    else     return false;
    */


// we need to do the following when reverse a chain of nodes
// 1. setup the prior and next pointer for each customer
// 2. set the first and last landfill for each trip
// 3. set the trip/tripid for each node(this is done
   implicitly when adding node to trip)
// 4. update the length and demand for each trip and the tour
// 5. add/delete the nodes for each trip
// 6. change the trips(ArrayList) for proper ordering
private void diff_trip_reverse(Node prior,VRPNode from, Node
   to, Node next) {
    LandFill lf1=null,lf2=null;
    int tripId1=-1,tripId2=-1;
    Trip t1=null,t2=null;
    if(to instanceof VRPNode) {
      tripId1 = from.getTripId();
      tripId2 = ((VRPNode)to).getTripId();
      lf1 = from.getTrip().getLastLandFill();
      lf2 = ((VRPNode)to).getTrip().getFirstLandFill();
      t1 = from.getTrip();
      t2 = ((VRPNode)to).getTrip();
    }
    else {
      tripId1 = from.getTripId();
      tripId2 = ((VRPNode)next).getTripId();
      lf1 = from.getTrip().getLastLandFill();
      lf2 = ((VRPNode)next).getTrip().getFirstLandFill();
      t1 = from.getTrip();
      t2 = ((VRPNode)next).getTrip();
    }
    if(tripId1==tripId2) System.out.println("FATAL_ERROR,2opt-
```

```
difftrip-reverse");

 //1. set up the prior and next pointer for nodes following
from and before to
 VRPNode [] n1 = t1.getNodes();
 VRPNode [] n2 = t2.getNodes();

 ArrayList a = new ArrayList();
 ArrayList b = new ArrayList();

 Node itr = from;
 while(itr instanceof VRPNode) {
   Node tmp = ((VRPNode)itr).getNext();
   ((VRPNode)itr).swap();
   a.add((VRPNode)itr);
   t1.delNode((VRPNode)itr);
   //partial_load1+=((VRPNode)itr).getDemand();
   itr = tmp;
 }
 itr = to;
 while(itr instanceof VRPNode) {
   Node tmp = ((VRPNode)itr).getPrior();
   ((VRPNode)itr).swap();
   b.add((VRPNode)itr);
   t2.delNode((VRPNode)itr);
   //partial_load2+=((VRPNode)itr).getDemand();
   itr = tmp;

 }
 if(prior instanceof VRPNode) ((VRPNode)prior).setNext(to);
 if(to instanceof VRPNode)  ((VRPNode)to).setPrior(prior);
 if(next instanceof VRPNode)  ((VRPNode)next).setPrior(from)
;
 from.setNext(next);

 t1.setLastLandFill(lf2);
 t2.setFirstLandFill(lf1);
 t1.addNode((VRPNode[])b.toArray(new VRPNode[0]));
 t1.rebuildTrip();
 t2.addNode((VRPNode[])a.toArray(new VRPNode[0]));
 t2.rebuildTrip();
 // change the landfill for these two trip

 // if there are trips between t1 and t2 we need to reverse
them also
```

```
    Trip[] t = from.getTour().getTrips();
    int i1=-1,i2=-1;
    for(int i=0;i<t.length;i++){
      if(tripId1==t[i].getId()) i1 = i;
      if(tripId2==t[i].getId()) {
        i2 = i;
        break;
      }
    }
    if(i1<0 || i2<0) System.out.println("fatal error here");
    for(int i=i1+1;i<i2;i++) t[i].swap();
    for(int i=i1+1,j=i2-1;i<j;i++,j--) {
      Trip tmp = t[i];
      t[i] = t[j];
      t[j] = tmp;
    }
    evaluate();
    //System.out.println("finish 2opt-difftrip-reverse");
  }
  /*
    int tripId1 = prior.getTripId();
    int tripId2 = next.getTripId();
    int partial_load1 = 0;
    int partial_load2 = 0;
    LandFill lf1 = from.getTrip().getLastLandFill();
    LandFill lf2 = to.getTrip().getFirstLandFill();
    //1. set up the prior and next pointer for nodes following
   from and before to
    Trip t1 = prior.getTrip();
    Trip t2 = next.getTrip();
    VRPNode [] n1 = t1.getNodes();
    VRPNode [] n2 = t2.getNodes();

    ArrayList a = new ArrayList();
    ArrayList b = new ArrayList();

    Node itr = from;
    while(itr instanceof VRPNode) {
      Node tmp = ((VRPNode)itr).getNext();
      ((VRPNode)itr).swap();
      a.add((VRPNode)itr);
      t1.delNode((VRPNode)itr);
      //partial_load1+=((VRPNode)itr).getDemand();
      itr = tmp;
    }
```

```
itr = to;
while(itr instanceof VRPNode) {
  Node tmp = ((VRPNode)itr).getPrior();
  ((VRPNode)itr).swap();
  b.add((VRPNode)itr);
  t2.delNode((VRPNode)itr);
  //partial_load2+=((VRPNode)itr).getDemand();
  itr = tmp;

}
prior.setNext(to);
to.setPrior(prior);
next.setPrior(from);
from.setNext(next);

t1.addNode((VRPNode[])b.toArray(new VRPNode[0]));
//something could be wrong!
//int asize = a.size();
//VRPNode [] rev = new VRPNode[a.size()];
//for(int i=asize-1;i>=0;i--) rev[asize-1-i] = (VRPNode)a.
get(i);
t2.addNode((VRPNode[])a.toArray(new VRPNode[0]));
// change the landfill for these two trip
t1.setLastLandFill(lf2);
t2.setFirstLandFill(lf1);
t1.rebuildTrip();
t2.rebuildTrip();

// if there are trips between t1 and t2 we need to reverse
them also
Trip[] t = (Trip[])trips.toArray(new Trip[0]);
int i1=-1,i2=-1;
for(int i=0;i<t.length;i++){
  if(tripId1==t[i].getId()) i1 = i;
  if(tripId2==t[i].getId()) {
    i2 = i;
    break;
  }
}
if(i1<0 || i2<0) System.out.println("fatal error here");
for(int i=i1+1;i<i2;i++) t[i].swap();
for(int i=i1+1,j=i2-1;i<j;i++,j--) {
  Trip tmp = t[i];
  t[i] = t[j];
  t[j] = tmp;
```

```
    }
    evaluate();

  }
*/
  public void testTwOpt() {
    twOpt();
  }

  /*
  public boolean twOpt_RTR(double record, double deviation) {
    boolean moved = false;
    for(int i=0;i<trips.size();i++) {
      Trip t = (Trip)trips.get(i);
      g = t.getGraph();
      VRPNode [] n = t.getNodes();
      for(int j=0;j<n.length;j++) {
        if(seek_rtr(n[j],record,deviation)) moved=true;
      }
    }
    return moved;
  }
  */
/*
  private boolean seek_rtr(VRPNode C1, double record, double
   deviation) {
    boolean moved = false;
    //System.out.println("seeking "+C1.getId());
    int c1 = C1.getId();
    Node C2 = C1.getNext();
    int c2 = C2.getId();
    VRPNode max_C3=null;
    double max_saving = -1e10;
    if(C2 instanceof VRPNode) {
      double d12 = g.distance(c1,c2);
      for(int m=0;m<g.NBCount[c2];m++) {
        int c3 = g.NBList[c2][m];
        VRPNode C3 = g.getCustomer(c3);
        double d23 = g.distance(c2,c3);
        //if(d23>=d12) break;
        if(c3==c1) continue;
        if(C3.inTheSameTour(C1)) {
        Node C4 = C3.getPrior();
        int c4 = C4.getId();
        if(c4==c2) continue;
```

```
        if(C4 instanceof VRPNode) {
        double d34 = g.distance(c3,c4);
        double d14 = g.distance(c1,c4);
//System.out.println("d12="+d12+",d34="+d34+",d23="+d23+",d14
    ="+d14);
double saving = d12+d34-d14-d23;
if(saving>max_saving) {
if(C3.inTheSameTrip(C1)){
if(
//moved=same_trip_seek_rtr(C1,(VRPNode)C2,C3,(VRPNode)C4,record
    ,deviation);
//if(improved)
//System.out.println("[2-opt same trip] saving "+saving);
}
else {
moved = diff_trip_seek_rtr(C1,(VRPNode)C2,C3,(VRPNode)C4,record
    ,deviation);
//if(improved)
//System.out.println("[tour 2-opt diff trip] saving "+saving);
}
}

if(improved) {
evaluate();
break;
}
        }
        }
      }
    }
    return improved;
  }
  */
private boolean local_twopt = false;

public double twOpt() {
    if(local_twopt) return local_twopt();
    else return global_twopt();
}
// one point move heuristc for landfill routing problem
// here we consider moving one customer from on place and
// insert it into the other place of the tour
//public double OPM() {}

public void chooseOptLandFill() {
```

```
    Trip[] t = (Trip[])trips.toArray(new Trip[0]);
    for(int i=0;i<t.length-1;i++) {
  Trip t1 = t[i];
  Trip t2 = t[i+1];
  VRPNode t1last = t1.getLastNode();
  VRPNode t2first = t2.getFirstNode();
  VRPGraph g = t1.getGraph();
  int optLF=-1;double min = 1e10;
  int oldLF = t1.getLastLandFill().getId();
  for(int k=0;k<g.getLandFillSize();k++) {
      double d = g.getLandFillById(k).getDist2Cust(t1last)+
    g.getLandFillById(k).getDist2Cust(t2first);
      if(d<min) {
    min = d;
    optLF=k;
      }
  }
  if(oldLF!=optLF) System.out.println("tour "+getId()+" find a
   better landfill");
  LandFill lf = g.getLandFillById(optLF);
  t1last.setNext(lf);
  t2first.setPrior(lf);
  t1.setLastLandFill(lf);
  t2.setFirstLandFill(lf);
  t1.evaluate();
  t2.evaluate();
    }
    evaluate();
}

public double getRemainDistance(Trip t) {
    Trip[] tp = getTrips();
    double ret = 0D;
    for(int k=tp.length-1;k>=0;k--)
  if(tp[k].getId()==t.getId()) break;
  else ret += tp[k].getRemainDistance();
  return ret;
}

public double getBeforeDistance(Trip t) {
    Trip[] tp = getTrips();
    double ret = 0D;
    for(int i=0;i<tp.length;i++) {
  if(tp[i].getId()==t.getId())
      break;
```

```
  else ret+=tp[i].getBeforeDistance();
    }
    return ret;
}

public Trip[] getTripsAfter(Trip t) {
    ArrayList a = new ArrayList();
    int i = trips.indexOf(t);
    for(int k=i+1;k<trips.size();k++)
  a.add(trips.get(k));
    if(a.size()==0) return null;
    else return (Trip[])a.toArray(new Trip[0]);
}

public void check_data_consistency(){
    Trip[] t = getTrips();
    for(int i=0;i<t.length;i++) t[i].check_data_consistency();
}


}
```

E.5 Trip.java

```
package edu.umd.math.lify.vrp;
import edu.umd.math.lify.graph.*;
import edu.umd.math.lify.util.*;
import java.util.*;

public class Trip implements Solution,Cloneable{
    /**
     * A trip is a sequence of nodes starting with a landfill
     * and ending with a landfill(could be the same) and in
   between
     * some nodes to visit
     */

    private VRPGraph g;
    private int numStop;
    private LandFill start;
    private LandFill end;

    //    boolean dirty;
    /**
     * all the customers in the trip
```

```
   */
  private ArrayList stop;
  /**
   * the trip length
   */
  private double length;
  /**
   * the total demand
   */
  private int demand;
  private int tid;
  private Tour tour;
  private static int GLOBAL_TID=0;

  public VRPGraph getGraph() { return g;}
  public void setGraph(VRPGraph graph) { g = graph; }

  public Object clone() {
//System.out.println("Trip.clone() is called");
Trip copy = null;
try {
    copy = (Trip) super.clone();
} catch(CloneNotSupportedException e) {System.out.println(e)
 ;}
//copy.stop = new ArrayList();
copy.stop = (ArrayList)stop.clone();
//for(int i=0;i<stop.size();i++) copy.stop.add(stop.get(i));
if(copy.length!=length) System.out.println("bad clone in trip
 ");
return copy;
  }


  public void fixNode() {
VRPNode[] n = getNodes();
clearNode();
for(int i=0;i<n.length;i++){
    VRPNode tmp = g.getCustomer(n[i].getId());
    stop.add(tmp);
}
if(stop.size() != n.length) System.out.println("fixNode
 failed");
  }

  public int getId() { return tid; }
```

```
  public Trip(LandFill s, LandFill t, VRPNode[] v,VRPGraph g)
  {
start = s;
end = t;
tid = GLOBAL_TID++;
stop = new ArrayList();
numStop = v.length;
for(int i=0;i<numStop;i++) {
    addNode(v[i]);
}
this.g = g;
evaluate();
  }

  public VRPNode[] getNodes() { return (VRPNode[])stop.
 toArray(new VRPNode[0]); }
  /**
   * return # of customers in the trip
   */
  public int size() { return stop.size(); }


  public void addNode(VRPNode v) {
stop.add(v);
demand+=v.getDemand();
//numStop++;
//if(tour!=null) v.setTourId(tour.getId());
v.setTripId(tid);
v.setTrip(this);
  }
  //ugly hack for global 2opt
  public void addNodeBefore(VRPNode []v) {
VRPNode [] exist = getNodes();
clearNode();
addNode(v);
addNode(exist);
  }

  public void addNode(VRPNode []v) {
for(int i=0;i<v.length;i++) addNode(v[i]);
  }

  public void delNode(VRPNode v) {
stop.remove(v);
demand-=v.getDemand();
```

```
v.setTrip(null);
v.setTripId(-100);
  }

  public void delNode(VRPNode []v){
for(int i=0;i<v.length;i++) delNode(v[i]);
  }

  public void clearNode() {
stop.clear();
demand=0;
length = 0;
  }

  //public void setDirty(boolean state) { dirty = state; }
  //public boolean isDirty() { return dirty; }

  public double evaluate() {
VRPNode [] nodes = (VRPNode []) stop.toArray(new VRPNode[0]);
length = 0D;
demand = 0;
if(size()>0) {
    numStop = stop.size();
    //System.out.println("evaluate()");
    length += start.getDist2Cust(nodes[0]);
    //System.out.println(start+"-->"+nodes[0]);
    //System.out.println("length="+length);
    //System.out.println(nodes[numStop-1]+"-->"+end);
    length += nodes[numStop-1].getDist2LandFill(end);
    //System.out.println("length="+length);
    for(int i=0;i<numStop-1;i++) {
  length+=nodes[i].getServiceTime();
  length+=g.distance(nodes[i],nodes[i+1]);
  demand+=((VRPNode)nodes[i]).getDemand();
    }
    demand+=((VRPNode)nodes[numStop-1]).getDemand();
    length+=((VRPNode)nodes[numStop-1]).getServiceTime();
}
//setDirty(false);
check_data_consistency();
return length;
  }

  public void rebuildTrip() {
```

```
VRPNode [] v = getNodes();
Node head=null;
for(int i=0;i<v.length;i++)
    if(v[i].getPrior().isLandFill()){
  head = v[i];
  break;
    }
clearNode();
while(head instanceof VRPNode) {
    addNode((VRPNode)head);
    head = ((VRPNode)head).getNext();
}
evaluate();
  }

  public double getLength() {
//if(isDirty())
//    evaluate();
//double oldLen = length;
//evaluate();
//if(Math.abs(oldLen-length)>1e-10) System.out.println("trip
 length is not consistent!record="+oldLen+",computed="+length
 );
return length;
  }

  //public boolean contains(VRPNode n) { return stop.contains
 (n); }

  public int getDemand() { return demand; }
  /**
   * tid is the id of the tour this trip belongs to
   */
  //public void setTourId(int id) { tid = id; }
  //public int getId() { return tid; }
  public void setTour(Tour t) {
tour = t;
//for(int i=0;i<stop.size();i++) ((VRPNode)stop.get(i)).
 setTourId(t.getId());
  }

  public Tour getTour() { return tour; }
  public int getTourId() { return tour.getId(); }

  public String toString() {
```

```
StringBuffer sb = new StringBuffer();
sb.append("Trip("+tid+")["+start.getId()+"]");
for(int i=0;i<stop.size();i++)
    sb.append(((Node)stop.get(i)).getId()+"-");
sb.append("["+end.getId()+"]");
sb.append("("+demand+","+length+")");
return sb.toString();
  }

  public boolean isConsistent() {
double oldLen  = length;
return Math.abs(evaluate()-oldLen)<1e-10;
  }

  // precodition: other is the first trip of a tour
  // postcodition: other is merged with the this trip and the
  rest trip following other should
  //                also be added to the tour this belongs to
  public void merge(Trip other) {
//demand+= other.getDemand();
VRPNode last = getLastNode();
VRPNode first = other.getFirstNode();
last.setNext(first);
first.setPrior(last);
VRPNode[] newNodes =(VRPNode []) other.stop.toArray(new
 VRPNode[0]);
//System.out.println("[merge] got "+newNodes.length+" nodes")
 ;
//first we add all the nodes in this trip in the our stops
for(int i=0;i<newNodes.length;i++) {
    addNode(newNodes[i]);
    numStop++;
}
end = other.end;
//other.numStop=0;
//    other.makeNull();
//setDirty(true);
evaluate();
Trip[] t = other.getTour().getTrips();
for(int i=0;i<t.length;i++) {
    if(t[i]!=other) t[i].setTour(this.getTour());
}
getTour().evaluate();
Tour other_father = other.getTour();
other.makeNull();
```

```
Trip [] remaining = other_father.getTrips();

for(int i=0;i<remaining.length;i++)
    getTour().addTrip(remaining[i]);
other_father.makeNull();
//we still need to set the tourId of other trips that is in
 the same tour as other

  }

  private void makeNull() {
getTour().remove(this);
stop = null;
  }

  public VRPNode getLastNode() {
return (VRPNode) stop.get(stop.size()-1);
  }
  public VRPNode getFirstNode() {
return (VRPNode) stop.get(0);
  }

  public LandFill getLastLandFill() {
if(!end.isDepot()) {
    return end;
}
else {
    VRPNode last = (VRPNode) stop.get(stop.size()-1);
    return last.getLastdrop();
}
  }

  public LandFill getFirstLandFill() { return start;}
  public void setLastLandFill(LandFill l) { end = l; }
  public void setFirstLandFill(LandFill l) {start = l; }
  //need to check the following code
  //not so 100% sure
  public int hashCode() { return tid; }
  public boolean equals( Object other) {
if(other instanceof Trip) {
    if(tid==((Trip)other).getId()) return true;
    else return false;
}
else return false;
  }
```

```
  public double twOpt() {
while(twOptStep()) ;
return length;
  }

  private boolean twOptStep() {
boolean improved = false;
VRPNode[] nodes = (VRPNode[])stop.toArray(new VRPNode[0]);
for(int i=0;i<nodes.length;i++) {
    if(seek(nodes[i])) {
  improved=true;
  break;
    }
}
return improved;
  }

  // for 2-opt in a trip
  // we have some constraint on the orientation of the trip.
  // a trip must start from the start landfill and end in the
  end landfill
  // so we should take extra care when reversing a chain of
 customers
  private boolean seek(VRPNode n) {
boolean improved = false;
VRPNode C1 = n;
Node C2 = n.getNext();
int c2 = C2.getId();
int c1 = C1.getId();
if(C2 instanceof VRPNode) { // here if C2 is the last
 customer, it's next node is a landfill
    double d12 = g.distance(c1,c2);
    // search through the variable-length neighbor list
    for(int m=0;m<g.NBCount[c2];m++) {
  int c3 = g.NBList[c2][m];
  VRPNode C3 = g.getCustomer(c3);
  double d23 = g.distance(c2,c3);
  //if(d23>=d12) break;
  if(C3.equals(C1)) continue;
  //if(C3.getTripId()==C1.getTripId()){
  if(C3.inTheSameTrip(C1)) {
      Node C4 = C3.getPrior();
      if(C4.equals(C2)) continue;
      int c4 = C4.getId();
```

```
      if(C4 instanceof VRPNode) {
    double d34 = g.distance(c3,c4);
    double d14 = g.distance(c1,c4);
    double saving = d12+d34-d14-d23;
    if(saving>1e-10) {
        //System.out.println("c2="+c1+",c2="+c2+",c3="+c3+",
 c4="+c4);
        //System.out.println("[local 2opt same trip]saving "+
 saving);
        System.out.flush();
        if(isAhead(C1,C3)){
      //System.out.println("("+c1+","+c2+","+c4+","+c3+")");
      reverse(C1,(VRPNode)C2,(VRPNode)C4,C3);
        }
        else {
      //System.out.println("("+c4+","+c3+","+c1+","+c2+")");
      reverse((VRPNode)C4,C3,C1,(VRPNode)C2);
        }
        length-=saving;
        getTour().evaluate();
        //evaluate();
        improved=true;
        check_feasibility();
        break;
    }
      }
  }
  }
    }
    return improved;
}

private void check_feasibility() {
    if( Math.abs(length-evaluate())>1e-8)
  System.out.println("BAD!!!");
}

public void check_data_consistency() {
    VRPNode [] n = getNodes();
    for(int i=0;i<n.length-1;i++) {
  if(n[i].getNext()!=n[i+1]) {
      System.out.println("link is not consistent on node("+n[
 i].getId()+") trip("+getId()+")");
      System.out.println("node("+n[i+1]+").class="+n[i+1].
 getClass()+",n[i].next.id="+n[i].getNext().getId()+"class="+
```

```
 n[i].getNext().getClass());
  }
    }
    for(int i=1;i<n.length;i++) {
  if(n[i].getPrior()!=n[i-1]) {
      System.out.println("link is not consistent on node("+n[
 i].getId()+") trip("+getId()+")");
      System.out.println("node("+n[i-1]+").class="+n[i-1].
 getClass()+",n[i].prior.id="+n[i].getPrior().getId()+"class=
 "+n[i].getPrior().getClass());
  }
    }
    if(n[0].getPrior()!=start) System.out.println("first
 landfill is not consistent");
    if(n[n.length-1].getNext()!=end){
  System.out.println("last landfill is not consistent on node
 "+n[n.length-1].getId());
  System.out.println("recorded="+end.getId()+"acutual="+n[n.
 length-1].getNext());
    }
}

// before prior->from->...->to->next
// after  prior->to->...->from->next
private void reverse(VRPNode prior,VRPNode from,VRPNode to,
 VRPNode next) {
    //System.out.println("before reverse:");
    //System.out.println(this);
    prior.setNext(to);
    next.setPrior(from);
    VRPNode itr = from;
    while(!itr.equals(next)) {
  VRPNode tmp = itr;
  itr=(VRPNode)itr.getNext();
  tmp.swap();
    }
    /*
        System.out.println("after swap");
        for(int i=0;i<stop.size();i++) {
        VRPNode n = (VRPNode)stop.get(i);
        System.out.println(n.getPrior().getId());
        System.out.println(n.getNext().getId());
        }
        */
    to.setPrior(prior);
```

```
    from.setNext(next);
    VRPNode[] nodes = (VRPNode[])stop.toArray(new VRPNode[0])
 ;
    stop.clear();
    Node it = nodes[0];
    while(!(it instanceof LandFill)) {
  stop.add(it);
  it = ((VRPNode)it).getNext();
    }
    //System.out.println("after reverse:");
    //System.out.println(this);
}

//return true if n1 is ahead of n2
private boolean isAhead(VRPNode n1,VRPNode n2) {
    VRPNode tag = n2;
    do {
  Node prior = tag.getPrior();
  if(prior instanceof LandFill) {
      return false;
  }
  tag =(VRPNode) prior;
    }while(tag!=n1);
    return true;
}

public void testAhead() {
    int n = stop.size();
    Random r = new Random();
    int i,j;
    do {
  i = r.nextInt(n);
  j = r.nextInt(n);
    } while(i==j);
    boolean result = isAhead((VRPNode)stop.get(j),(VRPNode)
 stop.get(i));
    if(i>j) System.out.println(result+" should be true");
    else System.out.println(result+" should be false");
}

// to change the orientatio of a Trip, we need to do the
 following step
// 1. switch start and end landfill
// 2. swap each node
public void swap() { // change the orientation of the trip
```

```
      LandFill tmp = start;
      start = end;
      end = tmp;
      VRPNode[] nodes = (VRPNode[])stop.toArray(new VRPNode[0])
   ;
      for(int i=0;i<nodes.length;i++)
    nodes[i].swap();
      stop.clear();
      VRPNode itr = nodes[nodes.length-1];
      for(int i=nodes.length-1;i>=0;i--)
    stop.add(nodes[i]);
  }


  public double getRemainDistance() {
      return length;
  }

  public double getBeforeDistance() {
      if( start.isDepot() ) {
    VRPNode v = getFirstNode();
    return length-start.getDist2Cust(v)+v.getLastdropDist();
      }
      else return length;
  }

    }
```

E.6 Saving.java

```
package edu.umd.math.lify.vrp;
import edu.umd.math.lify.graph.*;
//import edu.umd.math.lify.util.*;

public class Saving implements Comparable {
  private Node from,to;
  private double saving;
  private LandFill landfill;

  // the first type saving with directly connecting two nodes
  public Saving(Node f,Node t,double s) {
    this(f,t,null,s);
  }
  // the second type saving which connecting two nodes via a
    landfill
```

```
  public Saving(Node f, Node t, LandFill lf,double s) {
    from = f;
    to = t;
    saving =s;
    landfill = lf;
  }

  public int compareTo(Object other) {
    if(saving<((Saving)other).saving) return -1;
    else if(saving>((Saving)other).saving) return 1;
    else return 0;
  }


  public double getSaving() { return saving; }

  public String toString() {
    //if(from instanceof VRPNode && to instanceof VRPNode) {
    //  return
  //  return from.toString()+to.toString()+"saving="+saving;
    return "saving="+saving+"\t"+from.getId()+"-->"+to.getId();
  }

  public Node getFirst() { return from; }
  public Node getLast() { return to; }
  public LandFill getLandFill() { return landfill; }

  public boolean isFirstType() {
    return landfill==null;
  }

  public void makeNull() {
    //NT: need to check the syntax of heap
    // if the root of the heap is min, then makeNull() will
    // move the saving to the bottom of the heap
    saving = 100D;
  }
}
```

E.7 CWHeuristic.java

```
package edu.umd.math.lify.vrp;
import edu.umd.math.lify.util.*;
import edu.umd.math.lify.graph.*;
import java.util.*;
```

```
public class CWHeuristic implements Heuristic {
  private VRPGraph g;
//  private Solution sol;
  public Saving saving[][];
  public Saving saving2[][][];
  public Tour [] tour;
  private Depot depot;
  private double lamda;

  public CWHeuristic(VRPGraph vrp) {
    g = vrp;
//    sol = null;
    saving = null;
    depot = g.getDepot();
  }

  public void setParameter(double l) { lamda=l; }
/*
  public BinaryHeap getSavingList() {
    int lfSize = g.getLandFillSize();
    int n = g.size();

    saving = new Saving[n][n];
    saving2 = new Saving[n][n][lfSize];
    tour = new Tour[n];
    // first calculate savings between two customers
    // i is the first customer in one trip
    // j is the last customer in the other trip
    // s_ij = d_i + d^j - d_{ij} in this case
    BinaryHeap savingList = new BinaryHeap(n*(n-1)*(lfSize+1));
    //first type of saving is by connecting 2 nodes directly
    for(int i=0;i<n;i++) {
      VRPNode from = g.getCustomer(i);
      for(int j=0;j<n;j++) {
        if(j==i) continue;
        if(from.shareLandFillWith(g.getCustomer(j))){
          double s = depot.getDist2Cust(i) +
          g.getCustomer(j).getLastdropDist() - g.distance(i,j);
          if(s<=0) System.err.println("SAVING NEGTIVE!"+i+","+j
  );
          saving[i][j] = new Saving(from,g.getCustomer(j),-1*s)
  ;
          try {
            savingList.insert(saving[i][j]);
```

```
        } catch(Overflow of) {}
      }
    }
    for(int j=0;j<n;j++) {
      if(j==i) continue;
      VRPNode last = g.getCustomer(j);
      for(int k=0;k<lfSize;k++){
      LandFill lf = g.getLandFillById(k);
      if(from.hasCandLandFill(lf) && last.hasCandLandFill(lf)
 ) {
        double s = depot.getDist2Cust(i)+last.
 getDist2LandFill(depot)
          -lf.getDist2Cust(i)-lf.getDist2Cust(j);
        if(s<=0){}// System.err.println("possible Negtive 2nd
  saving!");
        else
          {
            saving2[i][j][k] = new Saving(from,last,lf,-1*s);
            try {
              savingList.insert(saving2[i][j][k]);
            } catch(Overflow of){}
          }
        }
      }
    }
  }
  //System.out.println("the maximun saving is "+savingList.
 findMin());
  System.out.println("the  savingList size is " + savingList.
 size());
  return savingList;
}
*/
public Saving[] getSavingList() {
  int lfSize = g.getLandFillSize();
  int n = g.size();
  ArrayList savingList = new ArrayList(n*(n-1)*(lfSize+1));
  saving = new Saving[n][n];
  saving2 = new Saving[n][n][lfSize];
  tour = new Tour[n];
  // first calculate savings between two customers
  // i is the first customer in one trip
  // j is the last customer in the other trip
  // s_ij = d_i + d^j - d_{ij} in this case
  //first type of saving is by connecting 2 nodes directly
```

```
 for(int i=0;i<n;i++) {
   VRPNode from = g.getCustomer(i);
   for(int j=0;j<n;j++) {
     if(j==i) continue;
     //if(MyMath.gridDistance(from,g.getCustomer(j))<=1) {
     if(from.shareLandFillWith(g.getCustomer(j))){
       double s = depot.getDist2Cust(i) +
       g.getCustomer(j).getLastdropDist() -
       lamda*g.distance(i,j);
       if(s<=0){}// System.err.println("SAVING NEGTIVE!"+i
+","+j);
       else {
         saving[i][j] = new Saving(from,g.getCustomer(j),-1*
s);
         savingList.add(saving[i][j]);
       }
     }

   }
   for(int j=0;j<n;j++) {
     if(j==i) continue;
     VRPNode last = g.getCustomer(j);
     //if(MyMath.gridDistance(from,last)<=1) {
       for(int k=0;k<lfSize;k++){
         LandFill lf = g.getLandFillById(k);
         //if(MyMath.gridDistance(from,lf)<=1 && MyMath.
gridDistance(lf,last)<=1){
         if(from.hasCandLandFill(lf) && last.hasCandLandFill
(lf)) {
           double s = depot.getDist2Cust(i)+last.
getDist2LandFill(depot)
             -lamda*(lf.getDist2Cust(i)+lf.getDist2Cust(j));
           if(s<=0){}// System.err.println("possible Negtive
 2nd saving!");
           else
             {
             saving2[i][j][k] = new Saving(from,last,lf,-1*s
);
             savingList.add(saving2[i][j][k]);
             }
           }
         }
   }
 }
 Saving [] sortedSaving = (Saving [])savingList.toArray(new
```

```
 Saving[0]);
  Arrays.sort(sortedSaving);
  System.out.println("the savingList size is   "+savingList.
 size());
  return sortedSaving;
}

public Solution apply() {
/**
  * for the Clark and Wright's saving
  * first we compute all the possible savings
  * and sort them in the decreasing order.
  * then pick the saving from top to bottom.
  */
  int n = g.size();
  int lfSize = g.getLandFillSize();
  //BinaryHeap savingList = getSavingList();
  Saving [] savingList = getSavingList();
  create_self_trip();
  int count=0;
  //while(!savingList.isEmpty()) {
  for(int w=0;w<savingList.length;w++) {
    //Saving max = (Saving)savingList.deleteMin();
    Saving max = savingList[w];
    //System.out.println("saving list size is "+ savingList.
 size());
    if(max.getSaving()>=0) {
      //System.out.println("OOPS, positive saving");
      continue;
    }
    Node f = max.getFirst();
    Node l = max.getLast();

    if(max.isFirstType()) {
      VRPNode last = (VRPNode) l;
      VRPNode first = (VRPNode) f;
      //we know this is type I saving, i.e. both of them are
 customers
      if(first.isLandFill()) {
        System.err.println("FATAL ERROR");
        System.exit(1);
      }
      if(!first.isFirstNode()||!last.isLastNode()
        ||first.getTrip().getTour().getId()==last.getTrip().
 getTour().getId()) continue;
```

```
            // D-first---D
        // D---last-D
        // after combination D---last-first---D
        int total_demand = last.getTrip().getDemand()+first.
getTrip().getDemand();
        double total_dist = last.getTrip().getTour().getLength
()+first.getTrip().getTour().getLength()+max.getSaving();
        if(total_demand<=g.getCapacity() && total_dist<=g.
getDisConstraint()) {
          //let's merge them. This action will only
          //System.out.println("finding 1st type saving!\n"+max
);
          //last.setNext(first);
          //first.setPrior(last);
          last.getTrip().merge(first.getTrip());
          //System.out.println("merging trip "+last.getTrip().
getId());
          //if(!last.getTrip().isConsistent()) {
          //  System.out.println("merging failed on "+last.
getTrip().getId());
          //}
          //first is no longer the first node in a trip
          // because last is in front of it
          int k = first.getId();
          for(int j=0;j<n;j++){
            if(j==k) continue;
            if(saving[k][j]!=null)
              saving[k][j].makeNull();
          }
          //last is no longer the last node in a trip
          // so the kth column should be null
          k = last.getId();
          for(int j=0;j<n;j++) {
            if(j==k) continue;
            if(saving[j][k]!=null)
              saving[j][k].makeNull();
          }
          count++;
          //if(count%((int)Math.sqrt(n))==0)
          //savingList.rebuild_heap();
        }
      }
      else {//now we know it's the 2nd type of saving
        // first check if first is the first and last is the
last and they are not in
```

```
      // the same tour
      VRPNode last = (VRPNode) l;
      VRPNode first = (VRPNode) f;
      LandFill landfill = (LandFill) max.getLandFill();
      if(landfill==null) {
        System.out.println("OOPS!LANDFILL in saving is null")
;
      }
      Tour ft = first.getTrip().getTour();
      Tour lt = last.getTrip().getTour();
      if(!first.isFirstNode()||!last.isLastNode()
        ||ft.getId()==lt.getId()) continue;
            // D-first----D
        //we need to merge the two tour
      if(ft.getLength()+lt.getLength()+max.getSaving()<=g.
getDisConstraint()) {
        //System.out.println("find 2nd type saving!\n"+max);
        int i = first.getId();
        for(int j=0;j<n;j++){
          if(j==i) continue;
          for(int k=0;k<lfSize;k++) {
            if(saving2[i][j][k]!=null){
              saving2[i][j][k].makeNull();
            }
          }
        }
        i = last.getId();
        for(int j=0;j<n;j++) {
          if(j==i) continue;
          for(int k=0;k<lfSize;k++) {
            if(saving2[j][i][k]!=null) {
              saving2[j][i][k].makeNull();
            }
          }
        }
        lt.merge(ft,landfill);
        count++;
        //if(count%((int)Math.sqrt(n))==0)
        //savingList.debug(true);
        //savingList.rebuild_heap();
      }
    }

  }
  // up to here we have all the savings ready
```

```
    // next we take saving from the list
    //g.dump();
    //System.out.println("call rebuild_heap "+ count + " times
    ");
    double result = 0D;
    ArrayList solution = new ArrayList();
    for(int i=0;i<tour.length;i++){
      if(!tour[i].isEmpty()){
        //System.out.println(tour[i]);
        solution.add(tour[i]);
        result+=tour[i].getLength();
      }
          }
    //System.out.println("the total distance traveled is " +
    result);
    saving = null;
    saving2 = null;
      return new LRSolution((Tour [])solution.toArray(new Tour
    [0]),g);
    }

    public void create_self_trip() {
      for(int i=0;i<g.size();i++){
      VRPNode n = g.getCustomer(i);
      n.setPrior(depot);
      n.setNext(depot);
      //VRPNode [] cust = new VRPNode[1];
      //cust[0] = n;
      Trip t = new Trip((LandFill)depot,(LandFill)depot,new
    VRPNode[]{n},g);
      tour[i] = new Tour(new Trip[]{t});
    }
    //System.out.println("**************** dumping tour
    ************************");
    //for(int i=0;i<g.size();i++)
      //System.out.println(tour[0]);
    }
}
```

E.8 Depot.java

```
package edu.umd.math.lify.vrp;
import edu.umd.math.lify.graph.*;
import edu.umd.math.lify.util.*;
/**
```

```
 * Depot is a speical landfill
 * It's always the first and last landfill in a tour
 * It contains all the customer.
 * @see LandFill
 */

public class Depot extends LandFill
{
  //private int id;
  protected LandFill [] landfill;
  //protected VRPNode [] customer;
  protected double [] dist2lf;
  //protected double [] dist2cust;
  public Depot(double x,double y,int id) {
    super(x,y,id);
  }

  /**
    * add all the other landfill
    */
  public void addLandFill(LandFill[] lf) {
    landfill = lf;
    dist2lf = new double[lf.length];
    for(int i=0;i<lf.length;i++) {
      dist2lf[i] = MyMath.sphericalDistance(this,lf[i]);
    }
  }

  /**
   * add all the customers
   */
  public void addCustomer(VRPNode []cus,double[] d) {
    //customer = cus;
    //  dist2cust = new double[cus.length];
    try {
      for(int i=0;i<cus.length;i++) {
        register(cus[i]);
        setDist2Cust(cus[i],d[i]);
      }
    } catch(Exception e){}
  }

  public double getDist2LandFill(int index) { return dist2lf[
   index]; }
  //public double getDist2Cust(int i) { return dist2cust[i]; }
```

```
  public String toString() {
    return "[DP]"+super.toString();
  }
  public boolean isDepot() { return true; }
  public double getDist2Node(Node n) {
    if(n instanceof LandFill) return getDist2LandFill(n.getId()
   );
    else return super.getDist2Node(n);
  }
}
```

E.9 TourPlot.java

```
package edu.umd.math.lify.vrp;
import java.io.*;
import edu.umd.math.lify.graph.*;

public class TourPlot {
  private String data;
  PrintWriter pw;

  public TourPlot(String output){
    data = output;
    try {
      FileWriter fw = new FileWriter(data,false);
      pw = new PrintWriter(new BufferedWriter(fw));
    } catch(FileNotFoundException e) {}
    catch(IOException ie){}
  }

  public void print(Node n) {
    pw.println(n.x+"\t"+n.y);

  }
  public void print(Trip t) {
    LandFill first = t.getFirstLandFill();
    LandFill last = t.getLastLandFill();
    VRPNode[] nodes = t.getNodes();
    print((Node)first);
    for(int i=0;i<nodes.length;i++)
      print((Node)nodes[i]);
    if(last instanceof Depot) {
      Node lastdrop = ((VRPNode)nodes[nodes.length-1]).
   getLastdrop();
      print(lastdrop);
```

```
    }
    print((Node)last);
  }

  public void print(Tour t){
    Trip[] trips = t.getTrips();
    Depot depot = trips[0].getGraph().getDepot();
    for(int i=0;i<trips.length;i++)
      print(trips[i]);
    print((Node)depot);
    print("\n\n");//required by gnuplot ot separate data
  }

  public void print(String s) { pw.print(s); }

  public void print(LRSolution sol) {
    Tour[] t = sol.getTours();
    for(int i=0;i<t.length;i++)
      print(t[i]);
  }

  public void close() {
    pw.close();
  }
}
```

E.10 BinaryHeap.java

```
package edu.umd.math.lify.util;
import edu.umd.math.lify.vrp.*;
// BinaryHeap class
//
// CONSTRUCTION: with optional capacity (that defaults to
100)
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// Comparable deleteMin( )--> Return and remove smallest
item
// Comparable findMin( )  --> Return smallest item
// boolean isEmpty( )     --> Return true if empty; else
false
// boolean isFull( )      --> Return true if full; else
false
// void makeEmpty( )      --> Remove all items
```

```
// ******************ERRORS********************************
// Throws Overflow if capacity exceeded

/**
 * Implements a binary heap.
 * Note that all "matching" is based on the compareTo
method.
 * @author Mark Allen Weiss
 */
public class BinaryHeap
{
    /**
     * Construct the binary heap.
     */
    public BinaryHeap( )
    {
        this( DEFAULT_CAPACITY );
    }

    /**
     * Construct the binary heap.
     * @param capacity the capacity of the binary heap.
     */
    public BinaryHeap( int capacity )
    {
        currentSize = 0;
        array = new Comparable[ capacity + 1 ];
    }

    /**
     * Insert into the priority queue, maintaining heap
order.
     * Duplicates are allowed.
     * @param x the item to insert.
     * @exception Overflow if container is full.
     */
    public void insert( Comparable x ) throws Overflow
    {
        if( isFull( ) )
            throw new Overflow( );

            // Percolate up
        int hole = ++currentSize;
        for( ; hole > 1 && x.compareTo( array[ hole / 2 ] )
< 0; hole /= 2 )
```

```
            array[ hole ] = array[ hole / 2 ];
        array[ hole ] = x;
    }

    /**
     * Find the smallest item in the priority queue.
     * @return the smallest item, or null, if empty.
     */
    public Comparable findMin( )
    {
        if( isEmpty( ) )
            return null;
        return array[ 1 ];
    }

    /**
     * Remove the smallest item from the priority queue.
     * @return the smallest item, or null, if empty.
     */
    public Comparable deleteMin( )
    {
        if( isEmpty( ) )
            return null;

        Comparable minItem = findMin( );
        array[ 1 ] = array[ currentSize-- ];
        heapify( 1 );

        return minItem;
    }

    /**
     * Establish heap order property from an arbitrary
     * arrangement of items. Runs in linear time.
     */
    public void buildHeap( )
    {
        for( int i = currentSize / 2; i > 0; i-- )
            heapify( i );
//      for(int i=0;i<currentSize;i++) {

    }

    /**
     * Test if the priority queue is logically empty.
```

```
         * @return true if empty, false otherwise.
         */
        public boolean isEmpty( )
        {
            return currentSize == 0;
        }

    public void debug(boolean d) { dbg = true; }
        /**
         * Test if the priority queue is logically full.
         * @return true if full, false otherwise.
         */
        public boolean isFull( )
        {
            return currentSize == array.length - 1;
        }

    public boolean allPositive() {
      for(int i=1;i<=currentSize;i++)
        if(((Saving)array[i]).getSaving()>=0) return false;
      return true;
    }

        /**
         * Make the priority queue logically empty.
         */
        public void makeEmpty( )
        {
            currentSize = 0;
        }

        private static final int DEFAULT_CAPACITY = 100;

        protected int currentSize;      // Number of elements
     in heap
        protected Comparable [ ] array; // The heap array
    private boolean dbg = false;
        /**
         * Internal method to percolate down in the heap.
         * @param hole the index at which the percolate begins.
         */
        private void heapify( int hole )
        {
/* 1*/          int child;
/* 2*/          Comparable tmp = array[ hole ];
```

```
/* 3*/       for( ; hole * 2 <= currentSize; hole = child )
             {
/* 4*/           child = hole * 2;
/* 5*/           if( child != currentSize &&
/* 6*/                   array[ child + 1 ].compareTo( array[
   child ] ) < 0 )
/* 7*/               child++;
/* 8*/           if( array[ child ].compareTo( tmp ) < 0 )
/* 9*/               array[ hole ] = array[ child ];
                 else
/*10*/               break;
             }
/*11*/       array[ hole ] = tmp;
         }


  public void fragmentation() {
    //Comparable zero = new Saving(null,null,-1e-10);
    int forward = 1;
    int backward = currentSize;
    int reduce_count=0;
    //if(dbg) System.out.println("before fragmentation saving
   size is "+size());
    outloop:while(forward<backward) {
      while(((Saving)array[forward]).getSaving()<0) {
        //System.out.println("forward="+forward);
        forward++;
        if(forward>=backward) break outloop;
      }
      while(((Saving)array[backward]).getSaving()>=0){
        //System.out.println("backword="+backward);
        backward--;
        if(backward<=forward) break outloop;
      }
      //if(dbg) System.out.println("forward="+forward+",
   backward="+backward);
      //if(((Saving)array[forward]).getSaving()<0) System.out.
   println("fragmentation not working");
      //if(((Saving)array[backward]).getSaving()>=0) System.out
   .println("fragmentation not working also");
      if(forward<backward) {
        Comparable tmp = array[forward];
        array[forward] = array[backward];
        array[backward] = tmp;
```

```
        forward++;backward--;
        //reduce_count++;
        //System.out.println("we make "+reduce_count+" null
 savings");
      }
   }
   currentSize=backward;
   //if(dbg) System.out.println("we made "+ reduce_count+"
 null savings after fragmentation the list size is "+ size())
 ;
}

public void rebuild_heap() {

   if(dbg) { System.out.println("dumping savinglist");
     dump();
   }
   fragmentation();
   if(dbg) {
     System.out.println("after fragmented,dumping saving list"
 );
     dump();
   }
   /*
   for(int i=1;i<=currentSize;i++) {
      if(((Saving)array[i]).getSaving()>=0) {
        System.out.println("savinglist is not fragmented at "+i
 +","+array[i]);
      }
   }
   */
   buildHeap();
}

public int size() { return currentSize; }

public void dump() {
   System.out.println("********dumping heap********");
   for(int i=1;i<=currentSize;i++)
     System.out.println("["+i+"]"+array[i]);
   System.out.println("*****************************");
}
          // Test program
   }
```

E.11 HeapNode.java

```
/*
 * @(#)HeapNode.java   1.0 96/04/14 Walter Korman
 */

package edu.umd.math.lify.util;

/**
 * HeapNode class, stores data and children for the Heap.
 * Offers construction, printing, and insertion/removal
   facilities.
 * This class is private, and only meant for use by the Heap
 * class.
 *
 * @version 1.0, 04/14/96
 * @author  Walter Korman
 * @see Heap
 */
class HeapNode {
    HeapNode  left, right;  /* children */
    HeapItem  item;         /* data */

    /**
     * Constructor, initializes member data.
     */
    HeapNode() {
  left = null;
  right = null;
  item = null;
    }

    /**
     * Constructor, allows setting data.
     * @param HeapItem item The data item
     */
    HeapNode(HeapItem item) {
  left = right = null;
  this.item = item;
    }

    /**
     * Constructor, allows setting children and data.
     * @param HeapNode l Left child
     * @param HeapNode r Right child
```

```
     * @param HeapItem item The data item
     */
    HeapNode(HeapNode l, HeapNode r, HeapItem item) {
  left = l;
  right = r;
  this.item = item;
    }

    /**
     * Print this node and its children in pre-order fashion.
     */
    public void print() {
  if(left != null)
      left.print();

  if(item != null)
      item.print();

  if(right != null)
      right.print();
    }
}
```

E.12 HeapItem.java

```
/*
 * @(#)HeapItem.java   1.0 96/04/14 Walter Korman
 */

package edu.umd.math.lify.util;

/**
 * HeapItem interface.  Must be implemented by any class which
   will
 * be stored in the Heap class.  The print() method isn't
   really
 * necessary, but can be useful for debugging/perusal of your
 * data structures.
 *
 * @version 1.0, 04/14/96
 * @author  Walter Korman
 * @see Heap
 */
public interface HeapItem {
```

```
    /**
     * Returns whether this item's value is greater than the
    passed item.
     * @param HeapItem item The item to compare this item to.
     */
    public boolean greaterThan(HeapItem item);

    /**
     * Prints this item.
     */
    public void print();
}
```

E.13 MyMath.java

```
package edu.umd.math.lify.util;
import edu.umd.math.lify.graph.*;

public class MyMath {
    final static double DOUBLE_INF = 1e10;
    final static int    INT_MAX = 2<<31-1;
    /**
     *constant <code>EARTH_RADIUS</code> radius of earth in
    kilometers
     *
     */
    public static final double EARTH_RADIUS=6378.1;

    /**
     * <code>SphericalDistance</code> Compute the spherical
    distance between two points.
     *
     * @param a1 longitude of the first point(in radius)
     * @param b1 lantitude of the first point(in radius)
     * @param a2 longitude of the second point(in radius)
     * @param b2 lantitude of the second point(in radius)
     * @return the spherical distance between (a1,b1) and (a2,
    b2)
     */
    public static double sphericalDistance(double a1,double b1,
    double a2,double b2){
  return EARTH_RADIUS*Math.acos(Math.cos(a1)*Math.cos(a2)*Math.
    cos(b1-b2)+Math.sin(a1)*Math.sin(a2));
    //  return Math.sqrt(sqr(a1-a2)+sqr(b1-b2));
  }
```

```
    public static double sphericalDistance(Node n1,Node n2) {
      return sphericalDistance(n1.x,n1.y,n2.x,n2.y);
    //  return euclideanDistance(n1.x,n1.y,n2.x,n2.y);
    }

    public static int gridDistance(Node n1,Node n2) {
      return Math.max(Math.abs(n1.getGridX()-n2.getGridX()),
   Math.abs(n1.getGridY()-n2.getGridY()));
//    return 0;
  }

    public static double euclideanDistance(double a1,double b1,
   double
    a2,double b2) { return Math.sqrt(sqr(a1-a2)+sqr(b1-b2)); }
    static double sqr(double x) { return x*x; }
}
```

E.14 Overflow.java

```
package edu.umd.math.lify.util;
public class Overflow extends Exception {
  public Overflow() {
    super();
  }
}
```

E.15 Solution.java

```
package edu.umd.math.lify.util;

public interface Solution {
  /**
   * the objective value of the solotion
   */
  public double evaluate();
  public double getLength();
}
```

References

[1] S.K. Amponsah and S. Salhi. The investigation of a class of capacitated arc routing problems: The collection of garbage in developing countries. *Waste Management*, 24:711–721, 2004.

[2] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. On the solution of traveling salesman problems. *Documenta Mathematica Journal der Deutschen Mathematiker-Vereinigung*, 1998. Available at `http://www.tsp.gatech.edu/methods/papers/index.html`.

[3] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. Finding tours in the TSP. Technical Report 99885, Universitat Bonn, Forschungsinstitut fur Diskrete Mathematik, 1999. Available at `http://www.tsp.gatech.edu/methods/papers/index.html`.

[4] Beltrami and Bodin. Networks and vehicle routing for municipal waste collection. *Networks*, 4:65–94, 1974.

[5] J. Bentner, G. Bauer, G. Obermair, I. Morgenstern, and J. Schneider. Optimization of the time-dependent traveling salesman problem with Monte Carlo methods. *Physical Review E*, 64(036701), 2001.

[6] L. Bodin, B. Golden, A. Assad, and M. Ball. Routing and scheduling of vehicles and crews: The state of the art. *Computers & Operations Research*, 10:63–211, 1983.

[7] J. Brandao. A tabu search algorithm for the open vehicle routing problem. *European Journal of Operational Research*, 157:552–564, 2004.

[8] M. Braun and J.M. Buhmann. The noisy Euclidean traveling salesman problem and learning. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14*, pages 251–258. Cambridge, MA:MIT Press, 2002.

[9] I-Ming Chao. *Algorithms and Solutions to Multi-level Vehicle Routing Problems*. PhD thesis, University of Maryland, College Park, Maryland, 1993.

[10] N. Christofides, A. Mingozzi, and P. Toth. The vehicle routing problem. In N. Christofides, A. Mingozzi, P. Toth, and C. Sandi, editors, *Combinatorial Optimization*, pages 315–338. Chichester, UK: John Wiley & Sons, 1979.

[11] G. Clarke and J. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12:568–81, 1964.

[12] J-F. Cordeau, M. Gendreau, G. Laporte, J-Y. Potvin, and F. Semet. A guide to vehicle routing heuristics. *Journal of the Operational Research Society*, 53:512–22, 2002.

[13] S. Coy, B. Golden, G. Runger, and E. Wasil. See the forest before the trees: Fine-tuned learning and its application to the traveling salesman problem. *IEEE Transactions on Systems, Man, and Cybernetics*, 28(4):454–64, 1998.

[14] R.A. Finkel and J.L. Bentley. Quad trees, a data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.

[15] M. Fisher. Optimal solution of vehicle routing problems using minimum k-trees. *Operations Research*, 42:626–642, 1994.

[16] Z. Fu, R. Eglese, and L. Li. A new tabu search heuristic for the open vehicle routing problem. *Journal of the Operational Research Society*, 56:267–274, 2005.

[17] M. Gendreau, G. Laporte, C. Musaraganyi, and E. Taillard. A tabu search heuristic for the heterogenous fleet vehicle routing problem. *Computers & Operations Research*, 26:1153–1173, 1999.

[18] M. Gendreau, G. Laporte, and J-Y. Potvin. Metaheuristics for the capacitated VRP. In P. Toth and D. Vigo, editors, *The Vehicle Routing Problem*, pages 129–54. Philadelphia, PA: SIAM, 2002.

[19] B. Golden, A. Assad, L. Levy, and F. Gheysens. The fleet size and mix vehicle routing problem. *Computers & Operations Research*, 11:49–66, 1984.

[20] B. Golden, T. Magnanti, and H. Nguyen. Implementing vehicle routing algorithms. *Networks*, 7:113–148, 1977.

[21] B. Golden, E. Wasil, J. Kelly, and I-M. Chao. The impact of metaheuristics on solving the vehicle routing problem: Algorithms, problem sets, and computational results. In T. Crainic and G. Laporte, editors, *Fleet Management and Logistics*, pages 33–56. Boston, MA: Kluwer, 1998.

[22] P. Jaillet. A priori solution of a traveling salesman problem in which a random subset of the customers are visited. *Operations Research*, 36(6):929–936, 1998.

[23] D. Johnson and L. McGeoch. The traveling salesman problem: A case study in local optimization. In E. Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. London: Wiley, 1997.

[24] M. Junger, G. Reinelt, and G. Rinaldi. The traveling salesman problem. In M. Ball, T. Magnanti, C. Monma, and G. Nemhauser, editors, *Network Models*, pages 225–330. The Netherlands: North Holland, 1995.

[25] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated anealing. *Science*, 220:617–80, 1983.

[26] A.M. Law and W.D. Kelton. *Simulation modeling and analysis*. McGraw-Hill, New York, 1999.

[27] L. Levy. Private communication, 2005.

[28] F. Li, B. Golden, and E. Wasil. Very large-scale vehicle routing: New test problems, algorithms, and results. *Computers & Operations Research*, 32:1165–79, 2005.

[29] C. Malandraki and M.S. Daskin. Time dependent vehicle routing problems: Formulation, properties, and heuristic algorithms. *Transportation Science*, 26:185–200, 1992.

[30] C. Malandraki and R.B. Dial. A restricted dynamic programming heuristic algorithm for the time dependent traveling salesman problem. *European Journal of Operational Research*, 90:45–55, 1996.

[31] J. Pepper, B. Golden, and E. Wasil. Solving the traveling salesman problem with annealing-based heuristics: A computational study. *IEEE Transactions on Systems, Man, and Cybernetics*, 32A(1):72–77, 2002.

[32] D. Pisinger and S. Ropke. A general heuristic for vehicle routing problems. Working paper, University of Copenhagen, Copenhagen, Denmark, 2005.

[33] G. Reinelt. TSPLIB. Available at `http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95`, 2001.

[34] S. Sahoo, S. Kim, B-I. Kim, B. Kraas, and A. Popov. Routing optimization for Waste Management. *Interfaces*, 35(1):24–36, 2005.

[35] D. Sariklis and S. Powell. A heuristic method for the open vehicle routing problem. *Journal of the Operational Research Society*, 51:564–573, 2000.

[36] J. Schneider. The time-dependent traveling salesman problem. *Physica A*, 314:151–155, 2002.

[37] L. Schrage. Formulation and structure of more complex/realistic routing and scheduling problems. *Networks*, 11:229–232, 1981.

[38] E. Taillard. A heuristic column generation method for the heterogenous fleet VRP. *RAIRO operations research*, 1999.

[39] C. Tarantilis. Threshold accepting-based heuristic for solving large scale vehicle routing problems. Working paper, National Technical University of Athens, Athens, Greece, 2003.

[40] C. Tarantilis, D. Diakoulaki, and C. Kiranoudis. Combination of geographical information system and efficient routing algorithms for real life

distribution operations. *European Journal of Operational Research*, 152:437–453, 2004.

[41] C. Tarantilis, G. Ioannou, C. Kiranoudis, and G. Prastacos. A threshold accepting approach to the open vehicle routing problem. *RAIRO*, 38:345–360, 2004.

[42] C. Tarantilis, G. Ioannou, C. Kiranoudis, and G. Prastacos. Solving the open vehicle routeing problem via a single parameter metaheuristic algorithm. *Journal of the Operational Research Society*, 56:588–596, 2005.

[43] C. Tarantilis and C. Kiranoudis. Boneroute: An adaptive memory-based method for effective fleet management. *Annals of Operations Research*, 115:227–41, 2002.

[44] C. Tarantilis, C. Kiranoudis, and V. Vassiliadis. A list based threshold accepting metaheuristic for the heterogeneous fixed fleet vehicle routing problem. *Journal of the Operational Research Society*, 54:65–71, 2003.

[45] C. Tarantilis, C. Kiranoudis, and V. Vassiliadis. A threshold accepting metaheuristic for the heterogeneous fixed fleet vehicle routing problem. *European Journal of Operational Research*, 152:148–158, 2004.

[46] P. Toth and D. Vigo, editors. *The Vehicle Routing Problem*. SIAM, Philadelphia, 2002.

[47] P. Toth and D. Vigo. The granular tabu search and its application to the vehicle routing problem. *INFORMS Journal on Computing*, 15(4):333–46, 2003.

[48] J. Xu and J. Kelly. A network flow-based tabu search heuristic for the vehicle routing problem. *Transportation Science*, 30:379–93, 1996.