ABSTRACT

Title of Dissertation:	DISCRETE OPTIMIZATION MODELS IN DATA VISUALIZATION
	Roselyn Mansa Abbiw-Jackson, Ph.D., 2004
Dissertation Directed By:	Professor B. Golden, RH Smith School of Business

Data visualization techniques have become important tools for analyzing large multidimensional data sets and providing insights with respect to scientific, economic, and engineering applications. Typically, these visualization applications are modeled and solved using nonlinear optimization techniques. In this dissertation, we propose a discretization of the data visualization problem that allows us to formulate it as a quadratic assignment problem. This formulation is computationally difficult to solve optimally using an exact approach. Consequently, we investigate the use of local search techniques, mathematical programming, and genetic algorithms for the data visualization problem. The space in which the data points are to be embedded can be discretized using an *n* x *n* lattice. Conducting a search on this *n* x *n* lattice is computationally ineffective. Consequently, we propose a divide-and-conquer approach that refines the lattice at each step. We show that this approach is much faster than conducting a search of the entire *n* x *n* lattice and, in general, it generates higher quality solutions. We envision two uses of our divide-and-conquer heuristics: (1) as stand-alone approaches for data visualization and (2) to provide good approximate starting solutions for a nonlinear algorithm.

DISCRETE OPTIMIZATION MODELS IN DATA VISUALIZATION

By

Roselyn Mansa Abbiw-Jackson

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
2004

Advisory Committee:
Professor Bruce Golden, Chair
Associate Professor Zhi-Long Chen
Associate Professor S. Raghavan
Professor Paul Smith
Professor Edward Wasil

Dedication

To my family.

Acknowledgements

While working on this dissertation, I have received a lot of help, guidance, and encouragement. I would like to express my appreciation and gratitude to everyone who directly or indirectly contributed to the completion of this dissertation.

I would like to take this opportunity to express my sincere gratitude and thanks to Professor Bruce Golden, Associate Professor S. Raghavan, and Professor Edward Wasil, for their helpful suggestions and guidance.

Finally, I would like to thank my family for their love and encouragement.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

The availability of superior instruments, computers and information technology has made possible the measuring of physical phenomena with higher precision and in a shorter time interval. This has changed not only the sources, nature, and volume of data available but also the numerical and graphical tools for data analysis. High volume or multidimensional data are frequently generated, captured, and stored in numerous operations in almost all spheres of human endeavor and particularly in science and business operations. For example, high-energy physics experiments generate data in the order of 1 – 10 MBs, about 10^8 – 10^9 times a year (Shoshani, 2003). Similar large and complex data operations and endeavors cover applications ranging from science, engineering, and medicine to commerce and finance (Mackinnon and Glick, 1999).

In today's business environment, transaction processing with the aid of computers and the use of information technologies such as barcode scanners generate huge volumes of data in operations ranging from retailing to banking to stock trading (Mackinnon and Glick, 1999). Many companies and organizations gather gigabytes or terabytes of business transactions, scientific data, web logs, satellite pictures, and text reports, which are large and complex (Morzy and Zakrzewicz, 2003). In essence, massive databases growing at unprecedented rates are indeed very common today.

Inherent in such data are important insights into the operations they represent. Businesses want to mine retail data to know how to acquire, retain, and increase the profitability and lifetime value of a customer (Cabena et al., 1997). Researchers are developing the tools to mine available data to discover knowledge that facilitate activities such as market research, fraud detection and prevention, the pricing of securities and derivatives, as well as the monitoring of the medical impacts of prescription drugs (Mackinnon and Glick, 1999). Consequently, data mining has become difficult to ignore and hence an area of intense research.

Data mining involves the extraction of hidden predictive information from large databases. It is a powerful new technology with great potential to help companies focus on the most important information in their data warehouses. Data mining tools help predict future trends and behaviors, allowing businesses to make proactive, knowledge-driven decisions. Data mining tools can answer business questions that traditionally were too time consuming to resolve. For instance, the type and number of all products in a customer's shopping basket can be recorded and examined, giving insight into the customer's behavior. This enables the shop to draw conclusions for the shop's presentation of its products (Morzy and Zakrzewicz, 2003).

Data mining is an interdisciplinary field and utilizes techniques and tools from fields such as machine learning, pattern recognition, statistics, database, and visualization, to address the issue of information extraction or knowledge discovery from complex databases (Cabena et al., 1997; Mackinnon and Glick, 1999). Before the advent of data mining, researchers focused on problems with data sizes that were at most a few hundred to a few thousand cases and had between one and a few dozen variables (Elder and Pregido, 1996). The field emerged when it was realized that traditional decision-support methodologies, which combine simple statistical techniques with executive information systems, could not handle large and complex data sets within the time limits and operational conditions imposed by today's business environment (Cabena et al., 1997). Enterprises must be able to recognize trends early in rapidly changing environments and implement their ideas as quickly as possible in order to survive and strengthen their own positions in an environment of increasing competition (Dorndorf and Pesch, 2003).

Today’s data sets are usually large and multidimensional, growing and changing with time; consequently, they are usually complex, dynamic, and difficult to visualize. Data visualization reveals the relationships and trends that are not evident from the raw multidimensional data sets by using mathematical techniques to reduce the number of dimensions while preserving the relevant inherent properties. Data visualization rests on the premise that a picture is worth a thousand words (Schiffman et al., 1981; Young, 1987). The practical value of data visualization is based on the fact that it is often easier and more informative to look at a picture of the data than to look at the data points themselves, particularly when the data set is large (Schiffman et al., 1981). Large and multidimensional data sets that require visualization are commonplace today and may be encountered in many disciplines ranging from the physical, biological, and behavioral sciences to product development, marketing, and advertising (Schiffman et al., 1981).

Popular techniques used to solve data visualization problems include multidimensional scaling (MDS) and Sammon maps (SM) (Borg and Groenen, 1997; Sammon, 1969; Schiffman et al., 1981; Young, 1987). These techniques solve the data visualization problem using nonlinear optimization techniques. A limitation of a nonlinear algorithm is the small number of vectors (data points) it can handle (Sammon, 1969). Even with today’s fast computers, nonlinear optimization techniques are usually slow and inefficient for large data sets. Discrete optimization techniques may provide an efficient solution to the data visualization problem.

The objectives of this dissertation are to:

(1) develop a discrete optimization formulation for the data visualization problem;

(2) develop an efficient divide-and-conquer approach to solve the discrete data visualization problem;

(3) investigate the use of local search, mathematical programming and genetic algorithms in providing accurate or efficient solutions;

(4) compare the divide-and-conquer discrete optimization heuristics with nonlinear optimization heuristics for the data visualization problem.

This dissertation is organized as follows.

In Chapter 1, we give an introduction and present the objectives of the dissertation.

In Chapter 2, we give an overview of the existing literature on data mining and data visualization. We present background information on quadratic assignment problems, local search heuristics, integer programming problems, and genetic algorithms.

In Chapter 3, we present the methodology that we use to formulate the data visualization problem as a quadratic assignment problem.

In Chapter 4, we investigate the use of four different local search heuristics, using different neighborhoods, to solve the discrete data visualization problem.

In Chapter 5, we investigate the use of an equivalent integer programming formulation to solve the quadratic assignment problem.

In Chapter 6, we investigate the use of a hybrid genetic algorithm heuristic to solve the data visualization problem. We use the results from our local search heuristic as the starting solutions for a genetic algorithm procedure.

In Chapter 7, we compare our local search heuristic to a nonlinear Sammon map procedure. We also investigate using the results from our local search heuristic as the starting solutions for the nonlinear procedure instead of using random starting solutions.

In Chapter 8, we compare our local search heuristic to a commercial nonlinear multidimensional procedure. We conduct experiments to see how well our heuristic performs on large data sets.

In Chapter 9, we give a summary of our results and present recommendations for future research.

Chapter 2: Literature review

Data mining techniques and methods have been developed to reveal relevant trends in high volume or multidimensional data to facilitate the making of scientific decisions on operational matters (Cabena et al., 1997; Borg and Groenen, 1997). In this chapter, we provide an overview of the literature on data visualization and multidimensional scaling techniques that may be relevant to the development of discrete models for solving the data visualization problem.

2.1 Data Visualization

Data mining activities include both directed and undirected approaches. Directed data mining focuses on one target variable. In undirected data mining, the goal is to understand the relationships amongst all of the variables. Data visualization is a key component of undirected data mining (Berry and Linoff, 2000).

Data visualization techniques are used to reveal relationships and trends that are not evident from raw, multidimensional data sets. They involve the use of mathematical techniques to reduce the number of dimensions while preserving the relevant inherent properties. The smaller number of dimensions can be easily evaluated by human observation (Sammon, 1969). The result renders complex data sets accessible to visual exploration and thus makes it easier to see structure not obvious from the raw data (Borg and Groenen, 1997). The results are presented in visual form, in two or three dimensions to facilitate human visualization (Sammon, 1969). The types of presentations range from scatter plots matrices and Chernoff faces to color encoded patterns and schemes for viewing hierarchical subsets (Mackinnon and Glick, 1999).

Applications are numerous and varied, especially in fields such as finance and marketing where an abundance of data exists (Deboeck and Kohonen, 1998; Berry and Linoff, 2004). Condon et al. (2002) used data visualization techniques to visualize data from a college selection problem and Condon et al. (2003) used data visualization to analyze the judgments of decision makers.

Popular techniques used to solve data visualization problems include multidimensional scaling (MDS) and Sammon maps (SM) (Borg and Groenen, 1997; Sammon, 1969; Young, 1987). Cox et al. (1997) provided examples of some of the visualization techniques available today. Traditionally, data visualization problems are solved using nonlinear optimization techniques (Sammon, 1969; Borg and Groenen, 1997). Sammon (1969) indicated that one of the limitations of a nonlinear algorithm is the small number of vectors (data points) it can handle. Even with today’s fast computers, nonlinear optimization techniques are usually slow and inefficient for large data sets.

2.2 Multidimensional Scaling

Today’s data sets are usually large and multidimensional, growing and changing, and hence dynamic (Mackinnon and Glick, 1999). Huber (1997) noted that massive databases become heterogeneous through opportunistic data collection (of various objects, by several observers, at multiple locations and times) and typically come from processes where data reduction cannot be built in. In such circumstances, traditional statistical techniques do not provide sufficient capacity in discovering knowledge inherent in the data. Friedman (1997) and Wegman (1995) discussed what is referred to as “the curse of dimensionality” that many algorithms used by statisticians suffer from.

Multidimensional scaling (MDS) is a set of mathematical techniques used to reduce multivariate or multidimentsional data to two or three dimensions to facillitate visualization (Kruskal and Wish, 1978). MDS refers to a class of data analysis techniques all of which portray the data's structure in a spatial fashion easily assimilated by the relatively untrained human eye (Kruskal and Wish, 1978; Young, 1987). The techniques are used to construct a geometric representation of the data, usually in a Euclidean space of fairly low dimensionality, while preserving some of the most prominent distance relationships in the original data set. Kruskal and Wish (1978) are among the pioneers in developing and using MDS techniques.

There are a number of different MDS techniques that may be distinguished based on the particular type of geometry into which one wants to map the data, the mapping function, the algorithm used to find an optimal data representation, and the treatment of statistical error in the models (Borg and Groenen, 1997).

MDS methods have been designed for all types of relational data matrices, including symmetric and asymmetric matrices, rectangular and square matrices, matrices with or without missing elements, equally and unequally replicated data matrices, two-way and multi-way matrices and other types of matrices (Torgerson, 1958; Young, 1987).

All MDS programs are iterative, that is, they all take the approach of trying over and over again to obtain the best possible solution (Schiffman et al., 1981). As is common in iterative processes, the quality of the solution is affected by the starting (initial) solution and the stopping criteria, so all MDS techniques involve some special way to get the iterations started and stopped. This is achieved by what is called the initialization routine and the termination routine.

2.3 Sammon Map

Sammon (1969) developed an algorithm for the analysis of multivariate data along with some experimental results. His algorithm is based on a point mapping of vectors from a high-dimensional space to a lower-dimensional space such that the inherent data structure is approximately preserved. Mainly mappings to two- and three-dimensional spaces are considered since the resulting data configuration can be easily evaluated by human observations in three or fewer dimensions (Sammon, 1969).

Sammon (1969) randomly selected the initial configuration for the vectors in the new dimension. The inter-point distances for the new dimension were computed and then used to define an error, which represented how well the present configuration of the points fit the original configuration. The next step in the algorithm was to change the new configuration so as to decrease the error. A steepest descent procedure was used to search for a minimum error.

Sammon's nonlinear mapping algorithm was applied on several data sets in order to test and evaluate the utility of the program in detecting and identifying structure in data (Sammon, 1969). Sammon reported results for several artificially generated data sets. For demonstration purposes, it was useful to work with artificially generated data so that the results could be compared with known data structures. Sammon (1969) noted that, for the utility of any data analysis technique, it is more convincing when the technique is applied to real data as opposed to artificially generated data, presuming of course that the analysis results are correct. Therefore, he ran his algorithm on many real data sets and achieved highly satisfactory results.

MDS and Sammon maps traditionally solve the data visualization problem using nonlinear optimization techniques (Sammon, 1969; Borg and Groenen, 1997). A limitation of a nonlinear algorithm is the small number of vectors (data points) it can handle (Sammon, 1969). Even with today's fast computers, nonlinear optimization techniques are usually slow and inefficient for large data sets.

Discrete optimization techniques may provide a possible alternative for the data visualization problem. Essentially, data visualization problems are assignment problems, so that it may help to model the data visualization problem as a quadratic assignment problem (QAP).

2.4 Quadratic Assignment Problem

Koopmans and Beckman (1957) were the first to introduce the QAP. They used it in an economic context to model a plant location problem. The problem involved the assignment of a set of *n* facilities that have certain amounts of flow between them to a set of *n* predetermined locations at certain distances apart in such way that the sum of product of flows and their respective distances is minimized. Therefore, the QAP is usually described in a facility-location context (Cela, 1998). The terms facility and location are used even if there is no occurrence of a facility-location problem.

The QAP has been studied extensively by mathematicians, computer scientists, operations researchers, and practitioners (Cela, 1998). Among the applications of the QAP are placement problems, scheduling, manufacturing, VLSI design, statistical data analysis, and parallel and distributed computing. Currently, the QAP has been applied to large variety of applications and areas such as scheduling, wiring problems in electronics, parallel and distributed computing, statistical analysis, design of control panels and typewriter keyboards, chemistry, archeology, balancing of turbine runners, and computer manufacturing (Burkard, 1984; Finke et al. 1987).

The QAP has enjoyed great interest mostly because of its wide applicability, but partly because of its insurmountable resistance to efficient solution techniques (Bazaraa and Sherali, 1980). This lack of success is attributed to its combinatorial nature, that is, the exponential growth of feasible binary solutions as a function of the parameter *n* of the problem (Adams and Johnson, 1994). The QAP is widely regarded as one of the most difficult combinatorial optimization problems (Ahuja et al., 2000). It belongs to a class of problems referred to as NP-hard, where NP stands for nondeterministic polynomial (Garey and Johnson, 1979). For such problems, no known algorithms are able to generate the best answer in an amount of time that grows only as a polynomial function of the number of elements in the problem. Actually, not only is the QAP NP- hard and NP-hard to approximate, it is also practically intractable as it is generally considered very difficult to solve (to optimality) QAP instances of size larger than 20 within reasonable time limits (Cela, 1998).

We note that remarkable progress in data structures and algorithmic developments, as well as major advances in computing technology, have enabled a tremendous increase in the size of NP-hard problems that can be solved. Robust solvers are now available that solve large-scale linear programming problems and various classes of integer programming problems (Hentenryck, 1999). For example, large-scale instances of combinatorial problems such as the traveling salesman problem (TSP) with thousands of cities can now be solved optimally in practice (Cela, 1998). However, the QAP still remains a challenge. It has been particularly difficult to use exact methods to solve even relatively small instances ($n \approx 20$) of the QAP, (Fleurent and Ferland, 1994).

Cela (1998) gives a general overview of the most studied aspects of the QAP and outlines a number of promising research directions. She states that a QAP of size n can in theory be solved by:

1. Enumerating the $n!$ feasible binary solutions.
2. Computing the objective function value at each point.
3. Selecting a solution with the minimum value.

When $n \geq 15$, enumeration is computationally intractable even for special implicit procedures designed to effectively eliminate nonoptimal solutions from consideration (Adams and Johnson, 1994). Generally problems of size larger than 20 cannot be solved to optimality in a reasonable amount of time, while problems of size 15 to 20 are considered to be difficult to solve, (Cela, 1998; Li et al., 1994).

Exact algorithms for the QAP include approaches based on dynamic programming (Christofides and Benavent, 1989), cutting planes (Bazaraa and Sherali, 1980) and branch-and-bound (Lawler, 1963). Among these only branch-and-bound algorithms are guaranteed to obtain the optimal solution but they generally are unable to solve problems of size larger than 20 (Ahuja et al., 2000).

Recently, theoretical results obtained on the combinatorial structure of the QAP polytope have raised new hopes that cutting planes might be successfully used to solve reasonably sized QAPs (Cela, 1998). Clearly, the design of efficient branch-and-cut methods is conditioned by the identification of new valid and possibly facet defining inequalities for the QAP polytope and the development of the corresponding separation algorithms. Thus, quite a lot of effort may be required before the current size limits of solvable QAPs can be significantly improved (Cela, 1998).

2.4.1 Linearizations of QAP

When dealing with QAPs, it appears that the quadratic form in its objective function destroys the hope of finding efficient solution methods (Cela, 1998). A well-known method to cope with the problematic quadratic form is the linearization of the QAP (Kaufman and Broeckx, 1978). Linearization transforms the quadratic QAP objective function into an equivalent linear function by introducing a number of new variables and constraints (Bazaraa and Sherali, 1980). Numerous QAP linearizations have been proposed (Bazaraa and Sherali, 1980; Frieze and Yadegar, 1983; Kaufman and Broeckx, 1978). Most QAP linearizations are mixed integer linear problems with a large number of variables and equations (Cela, 1998).

A lot of effort has been put into obtaining compact linearizations, i.e., linearizations with relatively few variables and constraints (Cela, 1998). The size of the linearization matters particularly in cases where pure enumeration procedures are used to solve the problem at hand. The linearization of Kaufman and Broeckx is perhaps the smallest QAP linearization in terms of the number of variables and constants. However, for large *n*, even this linearization has a large number of variables and constraints. Under these conditions, even powerful tools to cope with linear integer programs such as Benders' decomposition or cutting planes do not help a lot (Cela, 1998). These considerations do not matter when the overall problem can be embedded into a continuum, such as when linear programming and assignment problem-type relaxations are used (Padberg and Rijal, 1996).

The linearization of Frieze and Yadegar (1983) has turned out to be the foundation of many other linearizations. Some of the best existing bounding procedures for the QAP have been obtained by building on the linearization of Frieze and Yadegar.

2.4.2 QAP Heuristics

Since many applications of QAP give rise to problems of size far greater than 20, there is a need for good heuristics that can solve large size QAPs (Ahuja et al., 2000). Various polynomial time heuristics have been proposed to find good solutions for larger QAP instances, which arise for many applications. These heuristics do not always obtain the optimal solution, but they do produce good approximate solutions in a reasonable amount of time. Although many heuristics for the QAP have been proposed, no dominant algorithm has emerged (Tate and Smith, 1995). These heuristics include construction methods, local improvement methods, tabu search algorithms, simulated annealing approaches, and genetic algorithms (Ahuja, et al., 2000; Tate and Smith, 1995).

Construction methods basically start with an empty solution and recursively assign facilities to locations according to some criteria until all facilities have been assigned. Construction methods are considered to be the simplest heuristic approaches to the QAP and this simplicity is often associated with a poor quality of resulting results (Cela, 1998).

Local improvement methods are classical approaches to solving difficult combinatorial optimization problems (Cela, 1998). They belong to the larger class of local search methods. Local search methods start with an initial feasible solution and iteratively try to improve the current solution by replacing it with a better feasible solution from its neighborhood. Local improvement methods have the advantage of providing good solutions in a short amount of time. However, they have the drawback of stopping at the first local minimum (Fleurent and Ferland, 1994). Local search methods are still the method of choice for NP-hard problems as they provide a robust approach for obtaining high-quality solutions to problems of a realistic size in a reasonable amount of computing time (Aarts and Lenstra, 1997).

Tabu search is a heuristic method that improves on local search by allowing solutions to escape from a poor local minimum (Glover and Laguna, 1997; Skorin-Kapov, 1990). This method allows climbing moves when no improving neighbor exists. A data structure keeps track of the history of the search in order to prevent cycling.

Simulated annealing approaches are another group of heuristic methods that try to overcome the issue of poor local solutions found in hard combinatorial optimization problems. This approach is based on an analogy between combinatorial optimization problems and statistical mechanics (Wilhelm and Ward, 1987). Feasible solutions of the problem correspond to the states of a physical system, while the objective function value for a feasible solution corresponds to the energy of the state of the physical system (Cela, 1998). Simulated annealing is a therefore a general approach that can be applied to any combinatorial optimization problem as soon as a neighborhood structure has been introduced on the problem's set of feasible solutions (Burkard and Rendl, 1984). A current solution is updated when certain specified conditions are met until a stopping ruling is met. Even though tabu search and simulating annealing produce better results than local search methods, they require much more computational effort (Fleurent and Ferland, 1996).

Genetic algorithms (GA's) are a class of optimization procedures in which populations of individual solutions evolve in a manner inspired by evolution and natural selection (Fleurent and Ferland, 1996). There is a growing interest in solving problems based on principles of evolution and hereditary. Such systems maintain a population of potential solutions, they have some selection process based on fitness of individuals, and they use genetic operators (Michalewicz, 1996). Genetic algorithms are not overly costly in terms of computational effort (Tate and Smith, 1995). Unlike many other search algorithms, the lack of explicit memories makes GAs very fast (Rawlins, 1991).

2.5 Local Search

Many combinatorial optimization problems including the QAP are NP-hard and it is generally believed that NP-hard problems cannot be solved to optimality within polynomially bounded computation times (Aarts and Lenstra, 1997). In combinatorial optimization, optimal solutions to NP-hard problems are sought, a task which can be prohibitively difficult even for problem instances of relatively small size (Cela, 1998). In cases when an optimal solution to the problem is not found, we often find a local optimal solution that is good in some sense. For example, instead of looking for the best solution among all feasible ones, we may seek a solution that is the best out of a subset of feasible solutions. Finding such a solution could intuitively be easier than finding the optimal one (Cela, 1998). Formalizing this kind of compromise leads to what are commonly called local search approaches (Fleurent and Ferland, 1994).

Local search (LS) has grown from a simple heuristic idea into a mature field of research in combinatorial optimization and this area of discrete mathematics is of great practical use and is attracting ever-increasing attention (Aarts and Lenstra, 1997). A general conclusion is that LS algorithms can find good solutions for many problems of interest in low-order polynomial running times (Aarts and Lenstra, 1997).

Most LS algorithms have the advantage of being generally applicable and flexible (Aarts and Lenstra, 1997). Roughly speaking, LS starts with an initial solution and then continually tries to find better solutions by searching neighborhoods. It only requires a specification of solutions, an objective function, a neighborhood function, and an efficient method for exploring a neighborhood, all of which can be easily obtained for most problems.

A basic version of LS is iterative improvement (Aarts and Lenstra, 1997). Iterative improvement starts with an initial solution and searches its neighborhood for a better solution. If such a solution is found, it replaces the current solution, and the search continues. Otherwise, the algorithm returns the current solution as the local optimal solution. Iterative improvement can mean either first improvement or best improvement. The first improvement method searches the neighborhood until a better solution has been found and replaces the current solution with it. Best improvement replaces the current solution with the best solution in its neighborhood.

Finding efficient neighborhood functions that lead to high-quality local optima can be viewed as one of the challenges of LS algorithms. No general rules are available for defining good neighborhood structures and appropriate methods for searching through them have to be considered separately. The literature presents many examples of neighborhood functions and even for the same problem there are often many different possibilities available (Aarts and Lenstra, 1997).

Another important decision for LS is the choice of a feasible starting solution (Cela, 1998). Poor local optimal solutions found in several cases are attributed to bad starting solutions. A classical corrective approach to this shortcoming has been to run the LS procedure a number of times with different starting solutions and to keep the best solution found as the final solution (Aarts and Lenstra, 1997). This approach allows better solutions to be obtained in practice (Fleurent and Ferland, 1994).

2.6 Genetic Algorithm

There are many hard optimization problems, like the QAP, that arise frequently in engineering, economics, management, mathematics, and the social sciences and for which no reasonably fast algorithms have been developed (Michalewicz, 1996; Schwefel, 1981). With hard optimization problems, it is often only possible to find an efficient algorithm whose solution is approximately optimal. Consequently, there is much interest in approximation algorithms that can find near-optimal solutions with reasonable running times (Aarts and Lenstra, 1997).

Probabilistic algorithms can be used to solve some hard optimization problems. These algorithms do not guarantee the optimal solution, but by randomly choosing sufficiently many solutions the probability of error may be made as small as we like (Michalewicz, 1996). Genetic algorithms (GAs) are parallel, randomized-search optimization heuristics that are based on the biological process of natural selection (Tate and Smith, 1995). GAs are based on evolutionary strategies found in nature, that is, survival of the fittest (Michalewicz, 1996). GAs belong to the class of probabilistic algorithms, but they are different from other random algorithms as they combine elements of directed search and stochastic search and are therefore more robust than existing directed search methods, (Michalewicz, 1996). GAs often find the needle in the haystack, even though they use random search strategies (Mackinnon and Glick, 1999).

Genetic algorithms (GA) were first presented by Holland in the early 1970s (Holland, 1975). They have since become an important tool in machine learning and function optimization (Rawlins, 1991). A GA has a control structure that adapts to the problem being solved (Rawlins, 1991). GAs translate the environment and dynamics of a combinatorial optimization problem in terms of a coding structure and a stochastic battle for fitness amongst rival candidates (Mackinnon and Glick, 1999). GAs have found applications in many operations research problems. They have been applied to problems in scheduling, as well as in finance and insurance. For example, the European portfolio management for banks, OMEGA, utilizes GAs (Mackinnon and Glick, 1999).

Tate and Smith investigate the use of GAs to solve QAPs (Tate and Smith, 1995). They present a GA approach to QAPs that uses the problem specific structure. They show that the GA finds solutions that are competitive with those of previously known heuristics without undue computational overhead. They argue that GAs provide a particularly robust method for solving the QAP and its more complex extensions.

Pure GAs have shortcomings for combinatorial optimization problems. While the pure GA approach yields good solutions for small problems, it cannot really compete with other heuristics, such as simulated annealing, for larger problems (Fleurent and Ferland, 1996; Tate and Smith, 1995). Even though the best-known solutions for the most difficult large QAP problems could not be obtained with GAs, the results indicate that the GA approach works and can provide good solutions by means different from those used by simulated annealing and tabu search, (Tate and Smith, 1995). This suggests the use of GAs to complement and improve existing procedures for combinatorial optimization problems (Fleurent and Ferland, 1994). Therefore, it is common to hybridize GAs with heuristic techniques that already perform well for specific problems (Davis, 1991; Fleurent and Ferland, 1994).

A hybrid procedure that combines genetic operators with existing heuristics is proposed by Fleurent and Ferland (1994) to solve the QAP. They find that genetic operators improve the performance of both local search and tabu search.

A greedy randomized adaptive search procedure (GRASP) for the QAP is given by Li et al. (1994). GRASP is an iterative process consisting of two phases, a construction phase and a local search phase. The best overall solution is kept as the result. Li et al. (1994) discussed aspects of a GRASP implementation for solving the QAP. Their algorithm was tested on a board range of problems and produced good-quality solutions in a reasonable amount of computation time.

Ahuja et al. (2000) give a greedy genetic algorithm for solving the QAP. They investigate the use of several possible enhancements to GAs. The overall performance of the GA for the QAP is found to improve by using greedy methods. Fairly effective heuristic algorithms can be obtained by striking the right balance between greedy methods that improve the quality of solutions and methods that promote diversity (Ahuja et al., 2000).

A GA manipulates a population of solutions using probabilistic, genetic-like operators like pairwise string recombination, called crossover, and string mutation to produce new populations with the intent of generating solutions with better objective function values (Rawlins, 1991). The members of the population act as a primitive memory for the GA and the genetic operators are so chosen that manipulating the population often leads the GA away from unpromising areas of search and towards promising ones, without the GA having to explicitly remember its trail through the search space (Rawlins, 1991).

GAs perform multi-directional searches by maintaining a population of potential solutions and repeatedly performing a cycle of operations until some termination condition is satisfied. A GA for a particular problem must have the following components.

(a) A selection mechanism that selects individual solutions from the population, usually giving preference to those with better objective function values.

(b) A reproduction or crossover mechanism that generates new feasible solutions by combining features from many known solutions.

(c) A mutation mechanism that generates new feasible solutions by randomly perturbing a single known solution.

(d) An evaluation mechanism that plays the role of the environment and evaluates each solution in the population and gives some measure of its fitness.

(e) A culling or replacement mechanism that removes some solutions from the population.

(f) Values for parameters that the GA uses, e.g., population size and probabilities of applying genetic operators.

A GA has a genetic representation for potential solutions to the problem. An encoding scheme maps feasible solutions to strings. Traditionally binary representations were used for GAs. These binary representations, however, have some drawbacks when applied to multidimensional, high-precision numerical problems (Michalewicz, 1996). The effectiveness of a crossover operator depends greatly on the encoding scheme used. The encoding should be such that the crossover operator preserves high performing arrangements of strings within solutions (Ahuja et al., 2000). A GA must include a way to create an initial population of potential solutions. The performance of a GA is often sensitive to the quality of the initial population.

Reproducing subsets are selected from the current population. There is a great amount of flexibility in the choice of how to select individuals in a population. For example, parents may be selected according to their absolute fitness, their rank in the current population, or some other criteria (Tate and Smith, 1995). Selection mechanisms should allow for better solutions to reproduce more often (Fleurent and Ferland, 1996).

In addition, there are many different possible crossover and mutation schemes for a given problem. Mutation operators are unary transformations that create new individuals by a small change in a single individual. Crossover operators are higher order transformations that create new individuals by combining parts from several (two or more) individuals (Michalewicz, 1996). In most GA implementations, mutation only takes place on a newly formed offspring, but mutation and reproduction can also be completely independent (Tate and Smith, 1995).

Crossover generates new solutions using various reproductive strategies. Reproductive sets are usually of size two and the members are chosen probabilistically with probabilities weighted by the solution values. In general crossover, the *i*th symbol of an offspring is the *i*th symbol of one of the members of the reproductive set. The crossover operator should be capable of producing a new feasible solution by combining good characteristics of both parents (Ahuja, Orlin, and Tiwari, 2000). The offspring should be considerable different from both parents.

Mutation arbitrarily alters parts of a selected solution by a random change with a probability equal to the mutation rate. The idea behind the mutation operator is the introduction of some extra variability into the population. Mutation is probabilistic and is usually independent of the value of the solution. Mutation should increase the diversity in the population by introducing random variations in members of the population (Ahuja, Orlin, and Tiwari, 2000).

Replacement or culling replaces some or all of the original population with the new solutions. In classical GAs, the complete population is usually replaced at each generation whereas in steady-state models, only a few individuals of the population are replaced at each generation (Davis, 1991). Steady-state models exhibit very strong elitism and are therefore better suited for hybrid schemes (Fleurent and Ferland, 1996). Also, steady-state models have been found to be faster than generational reproduction (Ahuja, Orlin, and Tiwari, 2000). In most GAs the population size remains constant.

After a number of generations, when no further improvement is observed, the program has converged and it is hoped that the best individual represents a near-optimal solution. Often the algorithm is stopped after a fixed number of iterations depending on speed and resource criteria (Michalewicz, 1996).

Chapter 3: Methodology

3.1 Theoretical Development

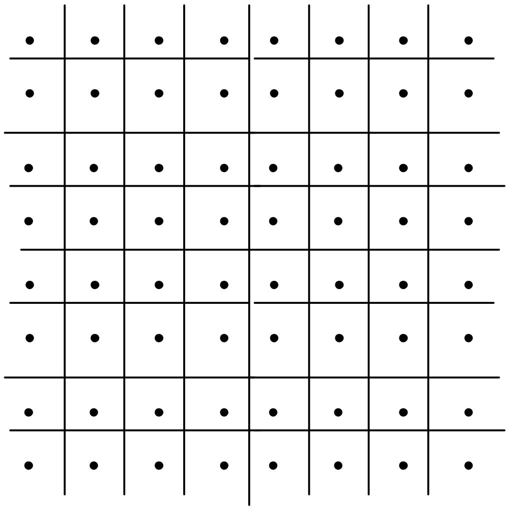


Figure 3.1 Lattice structure.

Given a set M of m points and their coordinates in r-space, the data visualization problem locates these points in q-space, $q < r$ (usually $q = 2$ or 3) such that a relevant measure of distance is preserved.

In order to apply discrete optimization techniques, we approximate the continuous q-space by a lattice N in q-space (see Figure 3.1) in which each cell has a center point. This results in the problem of assigning the m points to n lattice (center) points. We can make n as large or as small as the particular data visualization problem requires. A problem with points spread out will require a larger grid than one with points clustered together. The larger the grid, the more accurate the final result. To ensure that the grid (in q-space) scales to a given data set, we find the greatest distance between the pairs of points in M. Let this distance be a. Let the greatest distance in the chosen grid be b. Then we multiply all the original distances between points by *b*/*a*, so that our grid scales to the given problem.

An assignment problem assigns *r* indivisible entities, called facilities, to *r* mutually exclusive locations at a minimum cost. It is assumed that each facility must be assigned to exactly one location. Kaufman and Broeckx (1978) give the following mathematical formulation of the assignment problem:

$$\text{Minimize} \sum_{i=1}^{r}\sum_{j=1}^{r} c_{ij}x_{ij} \tag{3.1}$$

subject to

$$\sum_{j=1}^{r} x_{ij} = 1, \forall i = 1,\ldots,l \tag{3.2}$$

$$\sum_{i=1}^{r} x_{ij} = 1, \forall j = 1,\ldots,l \tag{3.3}$$

$$x_{ij} \in (0,1) \tag{3.4}$$

where x_{ij} equals 1 if facility *i* is assigned to location *j* and c_{ij} is the cost of assigning facility *i* to location *j*. The constraints in (3.2) model the fact that each facility must be assigned to exactly one location, while the constraints in (3.3) ensure that each location is assigned exactly one facility. The objective function is linear and assumes that the location of one facility does not affect the cost of locating other facilities.

3.2 Quadratic Assignment Problem (QAP)

In some assignment problems, the location of one facility affects the cost of another. This is the case in problems that involve the assignment of pairs of facilities. Thus, the assumption that the benefit of assigning a facility to some location does not depend on the locations of other facilities does not adequately address the reality of some assignment problems. That is, linear assignment problems cannot handle the complexities of the location decisions associated with all assignment problems. For example, the benefits of improvements to one location that extend to adjacent locations or the detrimental effects such as noise, vibration, and air or water pollution stemming from surrounding activities are not addressed by linear assignment problems (Koopmans, 1957). When the cost is affected by simultaneously making two assignments, the objective function can be formulated as a quadratic function of the assignment variables x_{ik} as follows (Kaufman and Broeckx, 1978):

$$\text{Minimize} \sum_{i=1}^{r}\sum_{j=1}^{r}\sum_{k=1}^{r}\sum_{l=1}^{r} c_{ijkl} x_{ik} x_{jl}$$

where c_{ijkl} is the cost of assigning facilities i and j to locations k and l, respectively. This problem is commonly known as the Quadratic Assignment Problem (QAP).

The problem of assigning m points to n lattice points cannot be treated as a linear assignment problem because a linear assignment problem assumes that the cost of assignment of one point to a lattice point does not depend upon the assignment of the other points. However, this is not the case for data visualization problems. We now formulate the data visualization problem, i.e., the problem of assigning m points to the n lattice points, as a quadratic assignment problem.

3.3 QAP Formulation of Data Visualization Problem

To formulate the data visualization problem as a QAP, let $od(i, j)$ for $i, j \in M$ be the original distance between two points i and j in r-space and $nd(k, l)$ for $k, l \in N$ be the distance between lattice points k and l in q-space. Let x_{ik} be a binary variable that is equal to one if point $i \in M$ is assigned to lattice point $k \in N$ and 0 otherwise. The data visualization problem as described above can then be formulated as follows:

$$\text{Minimize} \sum_{i \in M} \sum_{\substack{j \in M \\ j > i}} \sum_{k \in N} \sum_{l \in N} [od(i, j) - nd(k, l)]^2 x_{ik} x_{jl} \qquad (3.5)$$

subject to

$$\sum_{k \in N} x_{ik} = 1, \forall i \in M \qquad (3.6)$$

$$x_{ik} \in (0,1). \qquad (3.7)$$

The data visualization problem in (3.5) – (3.7) is known as the multidimensional scaling model with raw stress (Borg and Groenen, 1997) or the least squares scaling model (Young, 1987). Any other problem where the objective is a pairwise function of the points to be assigned in the data visualization problem can be modeled as a QAP. Other choices, which can be considered, include Sammon mapping (Sammon, 1969), classical scaling (Borg and Groenen, 1997), and all objective functions for nonmetric scaling (Borg and Groenen, 1997). The QAP formulation has *mn* variables and *m* assignment constraints.

The constraints in (3.6) are simple assignment constraints. They require that each point in $i \in M$ must be assigned to a lattice point $k \in N$ indicating that a data visualization problem can be formulated and solved as a QAP. Note that only one set of constraints is needed here since there is no restriction on the number of points assigned to a lattice point. After any two points $i, j \in M$ have been assigned to points $k,l \in N$, respectively, the distance between them, denoted by *nd*(*k*, *l*), can be calculated and their contribution to the objective function can be determined. The objective function in (3.5) penalizes deviations from *od*(*i, j*) quadratically.

3.4 Test of QAP Formulation

We implemented our QAP formulation in ILOG OPL Studio 3.5.1 (Hentenryck, 1999) and ran our experiments using Windows 2000 with an 800 MHz Pentium III processor and 512 MB of RAM. For our heuristics, we used values of $n = 4^p$, where p = 1, 2, 3, and 4.

First, we considered a lattice of size four, that is $n = 4$, and10 different sets of M, each with $m = 10$. We observed that arbitrary orientation resulted in four different solutions that were equal (see Figure 3.2). We also observed that symmetry exists between the second and third quadrants (see Figure 3.3). Successfully eliminating this symmetry will reduce the size of the solution space that the heuristic needs to handle and thereby improve the efficiency of our heuristic.

3.5 Elimination of Arbitrary Orientation and Symmetry

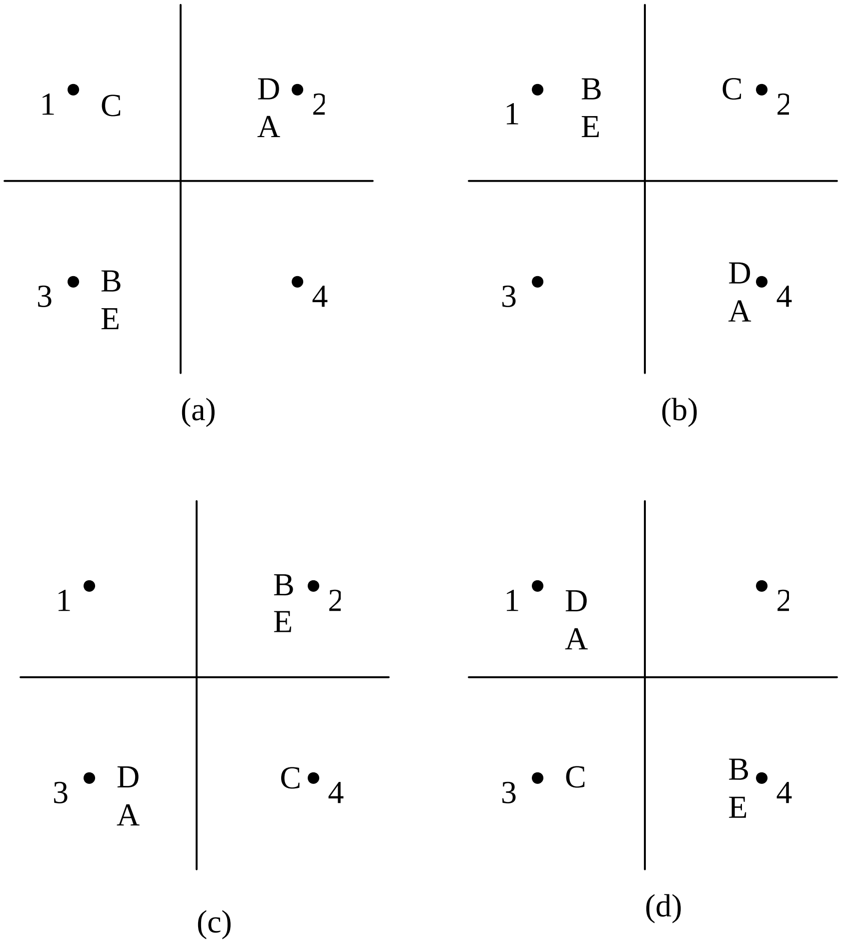
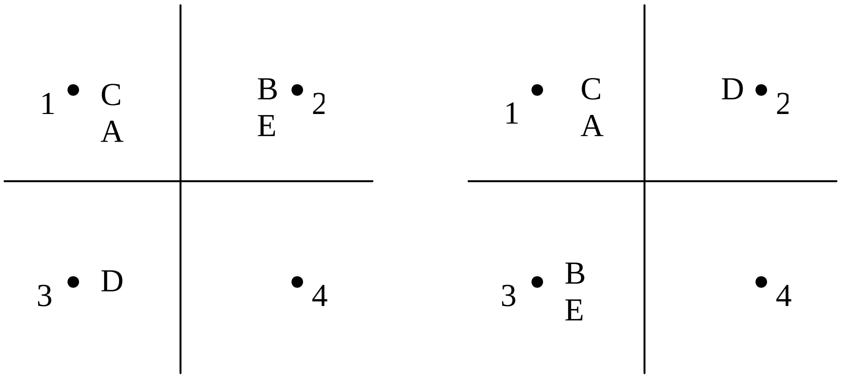


Figure 3.2 Four equivalent solutions resulting from arbitrary orientation.

In order to eliminate the arbitrary orientation, we fix the first point in *M* to the first quadrant, that is, $x_{11} = 1$ (for convenience, we use x_{11} instead of x_{A1}). For example, in Figure 3.2, we consider only the solution in Figure 3.2(d). To eliminate symmetry between the second and the third quadrant, we assign the next lowest numbered point that



(a) (b)

Figure 3.3 Two equivalent solutions resulting from symmetry.

can be assigned to quadrants two or three to quadrant two. So, point two cannot be assigned to quadrant three and all other points can only be assigned to quadrant three if a lower numbered point has already been assigned to quadrant two. That is, $x_{23} = 0;\ x_{i3} \le \sum_{\substack{j\in M \\ j<i}} x_{j2}, i \in M, i \ge 3$ (we point out that we use A = 1, B = 2, C = 3, D = 4, and E = 5, in our formulation). In Figure 3.3, we consider only solution 3.3(a).

	QAP		QAPSE	
Problem	Objective value Function	Running time (seconds)	Objective value function	Running time (seconds)
1	1785.58	17.21	1785.58	2.33
2	2253.06	22.42	2253.06	6.89
3	2038.34	15.45	2038.34	6.04
4	1848.60	23.60	1848.60	5.32
5	1279.13	13.89	1279.13	3.23
6	1841.48	14.72	1841.48	3.27
7	1848.53	47.78	1848.53	16.71
8	2185.95	12.96	2185.95	2.08
9	2494.98	58.48	2494.98	8.90
10	1183.79	13.50	1183.79	2.78
Average		23.99		5.76

Table 3.1. Results for 10-point problems for QAP and QAPSE.

Our QAP formulation with symmetry elimination constraints (QAPSE) is given below.

$$\text{Minimize} \sum_{i\in M}\sum_{\substack{j\in M\\ j>i}}\sum_{k\in N}\sum_{l\in N}[od(i,j)-nd(k,l)]^2 x_{ik}x_{jl} \tag{3.8}$$

subject to

$$\sum_{k\in N} x_{ik} = 1, \forall i \in M \tag{3.9}$$

$$x_{11} = 1 \tag{3.10}$$

$$x_{23} = 0 \tag{3.11}$$

$$x_{i3} \le \sum_{\substack{j\in M\\ j>i}} x_{j2}, \forall i \in M, i \ge 3 \tag{3.12}$$

$$x_{ik} \in (0,1) \tag{3.13}$$

In Table 3.1, we give the results for the QAP and QAPSE formulations for the sets of *M* of size 10. The average running times for QAP and QAPSE are 23.99 seconds and 5.76 seconds, respectively. The results indicate that the formulations give the same optimal solution and QAPSE is much faster than QAP. Thus, the symmetry elimination constraints greatly reduce the running time of QAP.

Despite recent progress, it is still not possible to solve the QAP formulation exactly for the problem sizes of interest here. Recall that only QAPs of size less than 20 can be solved to optimality (Cela, 1998). Therefore, we investigate the use of equivalent integer problems, local search techniques, and genetic algorithms to solve our QAP formulation for the data visualization problem.

Chapter 4: Local Search

There are a number of feasible solutions to a QAP. Finding the best solution is difficult and time consuming. As discussed in the previous chapter, discretizing a data visualization problem results in a large QAP problem that is very difficult to solve. One approach to overcoming this difficulty is to seek a solution that is the best solution in a subset of feasible solutions instead of looking for the best solution among all feasible solutions. Such a solution is usually referred to as a local optimum and it can be obtained by using local search techniques. In this chapter, we develop a local search technique to solve our QAP.

Our local search procedure starts with an initial feasible solution that is generated randomly. In this case, a solution is a string of numbers of length M where the ith number in the string represents the lattice point to which point $i \in M$ is assigned. We change the assignments of points in M in a solution to better assignments one point at a time, keeping all other points fixed. We use a best-improvement algorithm in which the current solution is replaced by the best solution in its neighborhood (the neighborhood structure is explained below). The process goes though all the points in M and is repeated until a stopping condition is satisfied.

More specifically, the contribution to the objective function of a point $i \in M$ is calculated for all possible assignments keeping all other points in M fixed at their current assignments. That is, we calculate, $\sum_{j \in M} \sum_{l \in N} [od(i, j) - nd(k, l)]^2 x_{jl}$, for all possible assignments of i to a point $k \in N'$, holding fixed the assignment of the other points j to lattice points $l \in N$, where N' is the neighborhood of the assignment of i. We assign point i to the lattice point that gives the smallest contribution to the objective function.

Our basic local search procedure is given as follows.

1. Start with a set S of t randomly generated feasible solutions.
2. For each solution $s \in S$, consider the points in M in random order.
   - (i) For a point $i \in M$ under consideration, keep the assignments of all points in M except i fixed as they are in solution s.
   - (ii) Calculate the contribution of point i to the objective function value for different lattice assignments k, where k belongs in the neighborhood of the assignment of point i in solution s. Identify the lattice assignment k^* that minimizes the value of the objective function in the neighborhood.
   - (iii) Point i is assigned to lattice point k^*.
   - (iv) Repeat steps (i) – (iii), following the order selected for points $i \in M$ until there are $|M|$ consecutive iterations with no improvement in the objective function value.
3. Calculate the objective function value for each $s \in S$.
4. The best solution is the one with the smallest objective function value.

We investigated four different discrete local search algorithms. These algorithms are described in Table 4.1. We implemented our discrete local search algorithms in the C++ programming language. We used Microsoft Visual C++ 6.0 and ran our experiments using Windows 2000 with an 800 MHz Pentium III processor and 512 MB RAM.

Algorithm	Description
LS	Local search heuristic
DAC	Divide-and-conquer local search heuristic
DACQ	Divide-and-conquer local search heuristic with quadrant restrictions
DACN	Divide-and-conquer local search heuristic with neighbor restrictions

Table 4.1 Local search algorithms.

4.1 Local Search Techniques

We tested our algorithms on artificially generated data sets. We applied our algorithm to several data sets with 50, 100, or 150 points. We set t = 100. The data sets were randomly generated from a lattice set of 256 points in two dimensions. For each problem size, 10 different problems were generated. To evaluate different versions of our local search heuristic, we used problems generated from a two-dimensional lattice, as it is easy to compare the quality of the computational results when the original set of points is in two dimensions. In this case, the optimal objective function value is known and equal to zero.

4.1.1 Local Search Heuristic

Initially, we selected the neighborhood of a point $i \in M$, assigned to $k \in N$, to be all points in N. That is, every lattice point $k \in N$ is considered as a possible choice for assigning $i \in M$.

In Table 4.2, we show the results for the experiments for this local search heuristic (LS). The frequency column gives the number of solutions (out of 100) that converged to the best solution. In nine of the 10 problems of size 50, LS finds the global

	50-point problems			100-point problems			150-point problems		
Problem	Best solution	Freq	Running time (sec)	Best solution	Freq	Running time (secs)	Best solution	Freq	Running time (secs)
1	0.00	1	87.07	0.00	10	360.30	0.00	7	851.19
2	0.00	2	64.68	0.00	8	414.34	0.00	5	805.37
3	0.00	2	83.32	0.00	8	462.46	0.00	6	913.42
4	62.30	3	73.52	0.00	7	325.42	0.00	9	739.81
5	0.00	1	98.76	0.00	1	519.55	0.00	13	810.72
6	0.00	5	80.09	0.00	12	306.68	0.00	21	702.37
7	0.00	8	74.47	0.00	3	449.00	0.00	23	665.79
8	0.00	4	82.36	0.00	1	429.53	0.00	2	814.69
9	0.00	1	83.91	0.00	4	359.89	0.00	16	889.03
10	0.00	1	100.86	0.00	10	452.81	0.00	8	786.27
Average			82.90			408.00			797.87

Table 4.2 Results for problem sets originally in two dimensions for LS.

optimum. For the 100-point and 150-point problems, LS finds the global optimum in all 20 problems. The average running times are 82.90 seconds, 408.00 seconds, and 797.87 seconds for the 50-point, 100-point, and 150-point problems, respectively.

LS allows points to be assigned to all points in the lattice structure. In this case we consider 256 lattice points. As the size of *M* increased, the running time for LS increased substantially.

4.1.2 Divide-and-Conquer Local Search Heuristic

To solve the problem of large running times with LS, we propose a divide-and-conquer heuristic to reduce the size of the problem that is solved at each stage. Our divide-and-conquer heuristic has five steps.

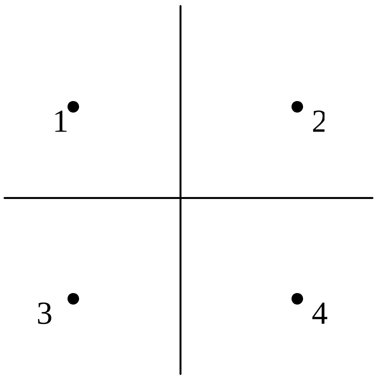


Figure 4.1 Lattice of four points.

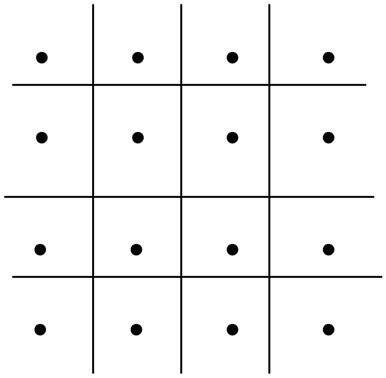


Figure 4.2 Lattice with 16 points after the four initial points have been subdivided into four additional points each.

1. Start with a lattice of four points (see Figure 4.1).

2. Perform local search (as described above) on points in *M* using these four points; that is, points in *M* can be assigned to only these four lattice points.

When local search terminates, the solutions that have been generated have points divided into four quadrants.

3. Divide each quadrant into four points (see Figure 4.2).

4. Randomly assign points from each quadrant from the previous assignment to one of the four new points. These solutions are the starting solutions for local search and local search is performed using the new lattice structure; that is, points can be assigned to any of the points in the current lattice structure in local search.

5. Continue dividing each point into four points and repeat the previous step until a stopping rule is met. We stop at 256 points, unless otherwise specified.

For the divide-and-conquer heuristic, the initial feasible solutions are generated randomly taking into consideration the symmetry elimination constraints. That is, for the initial step of our local search procedure where there are only four lattice points, the symmetry constraints are taken into account and the first point is always assigned to the first lattice point.

We apply our divide-and-conquer local search heuristic (DAC) to the same problem sets we used to test LS. In Table 4.3, we show the results for the experiments for DAC. For all 30 problems, DAC finds the global optimum. In two of the ten 100-point problems and six of the ten 150-point problems, all solutions generated by DAC were optimal (that is, the frequencies were 100%). The average running times for the 50-point problems, the 100-point problems, and the 150-point problems are 41.24 seconds, 96.91 seconds, and 163.14 seconds, respectively.

In seven of 10 problems for the 50-point problems, DAC finds the optimal solution more times than LS. For the 100-point problems, DAC finds the optimal solution more times in nine of the 10 problems. In all 150-point problems, DAC finds the optimal solution with a higher frequency than LS. Also, in the few problems for which LS has a

	50-point problems			100-point problems			150-point problems		
Problem	Best solution	Freq	Running time (secs)	Best solution	Freq	Running time (secs)	Best solution	Freq	Running time (secs)
1	0.00	69	30.03	0.00	58	136.34	0.00	100	144.81
2	0.00	3	46.57	0.00	92	92.97	0.00	90	192.34
3	0.00	1	39.49	0.00	77	85.80	0.00	100	145.40
4	0.00	52	31.74	0.00	2	102.56	0.00	63	167.16
5	0.00	43	51.04	0.00	91	77.57	0.00	100	149.24
6	0.00	50	34.91	0.00	100	75.59	0.00	100	145.32
7	0.00	5	47.14	0.00	46	124.60	0.00	57	206.28
8	0.00	8	57.23	0.00	10	85.41	0.00	100	160.94
9	0.00	3	44.69	0.00	68	112.42	0.00	90	170.22
10	0.00	1	29.61	0.00	100	75.85	0.00	100	149.65
Average			41.24			96.91			163.14

Table 4.3 Results for problem sets originally in two dimensions for DAC.

higher frequency, the difference in the frequency of optimal solutions generated was relatively small. For example, consider problem 4 in the 100-point problems. LS found the optimal solution seven times while DAC finds the optimal solution two times.

Our experiments indicate that DAC generates better results with much smaller running times than LS. Furthermore, DAC finds the optimal solution with a greater frequency than LS. Thus, the probability that DAC finds the optimal solution is likely to be greater than for than LS, especially if the procedure involves fewer starting solutions. However, for a 150-point problem with a 256-point lattice, DAC has an average running time of 163 seconds. For larger data sets with more lattice points, the running time may become very large. This is because DAC allows points to be assigned to any of the lattice points in the current lattice structure. So, as *n* increases, more lattice points have to be considered and this will increase the running time of DAC.

4.1.3 Divide-and-Conquer Local Search Heuristics with Quadrant and Neighbor Restrictions

Next, we suggest two refinements to DAC that reduce running time. We propose a divide-and-conquer local search heuristic with quadrant restrictions (DACQ) and a divide-and-conquer local search heuristic with neighbor restrictions (DACN). In these local search algorithms, there are neighborhood restrictions on the lattice points to which points can be assigned. Both algorithms follow the same steps used in DAC. However, the points in *M* are not assigned to all of the lattice points. The neighborhood is restricted as follows.

For DACQ, we restrict a point to the quadrant in which it is assigned initially. When a lattice point is divided into four new points, we consider only movements of currently assigned points to one of the four new points. Therefore, at each stage of the algorithm, only four lattice points are considered for local search.

For DACN, we restrict a point so that it can be assigned only to a neighbor of the lattice point to which it is currently assigned. A solution *s'* belongs to the neighborhood of a solution *s* if for any point $i \in M$ with an assignment of $k \in N$ in *s*, the assignment of *i* in *s'* is *l*, where *l* is any lattice point in *N* that is next to *k*, either horizontally, vertically, or diagonally (see Figure 4.3). Thus, at each step, a maximum of nine lattice points can be considered for local search. Note that *i* can stay where it is currently assigned.

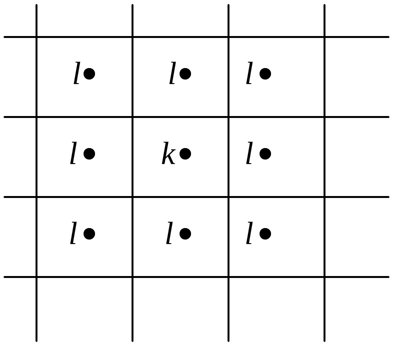


Figure 4.3 Neighborhood of a lattice point.

In Tables 4.4 and 4.5, we show the results for DACQ and DACN for the same problems used in the previous experiments. DACQ generates very poor results. It never finds the global optimum to any of the 50-point, 100-point, or 150-point problems. On the other hand, DACN produces very good results. In nine of 10 problems of size 50 and nine of the 10 problems of size 100, DACN finds the global optimum. DACN finds the global optimum in all 10 problems of size 150.

The average running times for DACQ are 1.22 seconds, 5.01 seconds, and 11.26 seconds for the 50-point problems, the 100-point problems, and the 150-point problems, respectively. For DACN, the average running times for the 50-point problems, the 100-point problems, and the 150-point problems are 4.07 seconds, 13.24 seconds, and 27.95 seconds, respectively. Both heuristics have much lower running times than DAC.

Considering both solution quality and running time, DACN appears to be the best heuristic, when compared to LS, DAC, and DACQ. It gives high-quality solutions in a reasonable amount of time. Increasing the size of *n* does not increase the size of the

	50-point problems			100-point problems			150-point problems		
Problem	Best solution	Freq	Running time (secs)	Best solution	Freq	Running time (secs)	Best solution	Freq	Running time (secs)
1	1911.54	5	1.20	4154.63	8	4.59	7760.85	17	10.80
2	2062.03	1	1.29	4765.05	6	5.00	12795.7	4	12.67
3	2873.79	1	1.16	7140.47	9	5.05	9190.68	21	10.19
4	4914.27	18	1.20	10587.6	5	4.65	16183.4	10	12.08
5	4323.12	5	1.21	9021.32	32	4.75	13719.6	27	10.61
6	1291.25	14	1.12	4081.64	1	4.74	3768.71	24	9.55
7	2579.19	2	1.21	10244.5	6	5.51	12331.5	9	11.56
8	2008.61	2	1.20	4036.79	8	5.19	9836.29	1	12.59
9	2306.85	15	1.26	6457.58	10	5.42	6439.19	3	11.60
10	1934.54	3	1.37	6400.54	8	5.18	8715.07	17	10.91
Average			1.22			5.01			11.26

Table 4.4 Results for problem sets originally in two dimensions for DACQ.

	50-point problems			100-point problems			150-point problems		
Problem	Best solution	Freq	Running time (secs)	Best solution	Freq	Running time (secs)	Best solution	Freq	Running time (secs)
1	0.00	70	3.88	0.00	50	13.72	0.00	91	28.00
2	0.00	1	4.25	0.00	91	14.83	0.00	84	32.79
3	131.77	29	4.12	0.00	72	12.98	0.00	99	27.67
4	0.00	52	3.85	631.23	100	11.79	0.00	55	26.54
5	0.00	43	4.49	0.00	91	12.28	0.00	98	26.04
6	0.00	47	3.86	0.00	95	12.71	0.00	93	25.90
7	0.00	5	4.04	0.00	29	13.90	0.00	65	26.80
8	0.00	4	4.50	0.00	13	13.39	0.00	98	28.86
9	0.00	36	4.02	0.00	79	13.57	0.00	88	29.83
10	0.00	2	3.72	0.00	98	13.19	0.00	97	27.04
Average			4.07			13.24			27.95

Table 4.5 Results for problem sets originally in two dimensions for DACN.

neighborhood, as in DAC, since a maximum of nine lattice points is considered at each stage of the algorithm. This results in a more gradual increase in the running time of DACN as a function of *n*.

4.2 Results and Analysis for DACN

In this section, we apply DACN to several data sets with 50, 100, and 150 points with nonzero global optimal objective function values. The data sets were randomly generated from lattice sets in three, four, and five dimensions (e.g., for three dimensions, points were generated from a 16 × 16 ×16 lattice; for four dimensions, points were generated from a 16 × 16 ×16×16 lattice, and so on). For each combination of dimension and size, 10 different problems were generated. The nine problem sets (problem sets 1 to 9) are described in Table 4.6.

In all our experiments, we use $q = 2$. In cases where $r = q$, it is easy to compare the quality of the computational results. In this case, the optimal objective function value is known and equal to zero. However, for problems where $q < r$, the optimal value of the objective function is unknown and greater than zero. No local criterion exists for deciding how good a local optimal solution is as compared to a global one (Cela, 1998). Actually, from a complexity point of view, Cela (1998) states that even deciding whether a given local solution is a global optimal is an NP-hard problem.

Problem Set	Dimensions	Number of Points
1	3	50
2	3	100
3	3	150
4	4	50
5	4	100
6	4	150
7	5	50
8	5	100
9	5	150

Table 4.6 Characteristics of problem sets.

	50-point problems			100-point problems			150-point problems		
Problem	Best solution	Freq	Running time (secs)	Best solution	Freq	Running time (secs)	Best solution	Freq	Running time (secs)
1	20982.0	2	3.93	101459	1	16.59	247553	1	36.80
2	16251.0	1	4.78	105203	3	17.13	265991	1	38.39
3	25871.5	2	4.23	126796	1	15.52	296865	2	33.57
4	22649.6	1	4.38	97143.6	1	12.96	246756	1	33.68
5	27914.8	1	4.49	133668	1	15.19	306127	1	37.46
6	22609.5	1	4.19	101323	17	15.45	244771	3	32.96
7	14946.6	1	4.12	82331.6	5	14.61	263992	6	35.41
8	16103.2	1	4.18	84547.2	7	15.93	231010	1	44.75
9	21778.1	2	4.51	115651	2	17.68	299859	1	37.49
10	26752.2	5	4.50	90984.9	1	16.19	261190	1	35.95
Average			4.33			15.73			36.65

Table 4.7 Results for problem sets 1, 2, and 3 for DACN. These problem sets are originally in three dimensions.

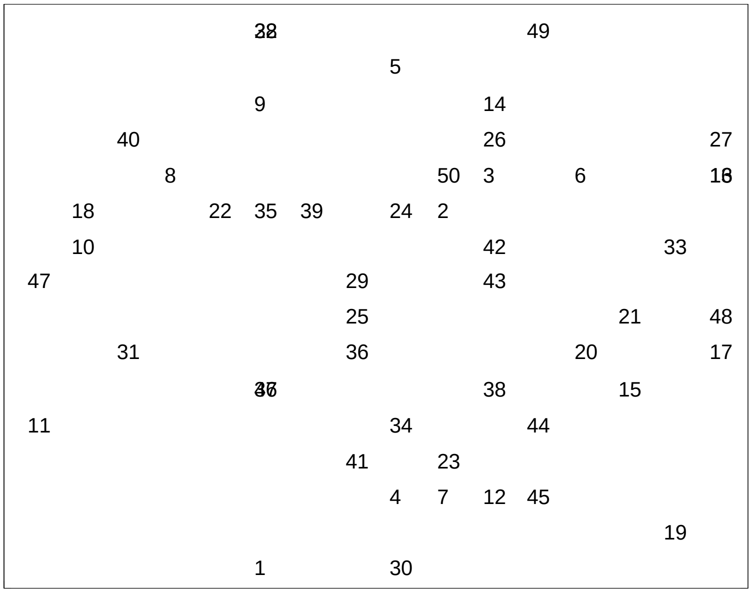


Figure 4.4 Plot for problem 2 from Problem Set 1.

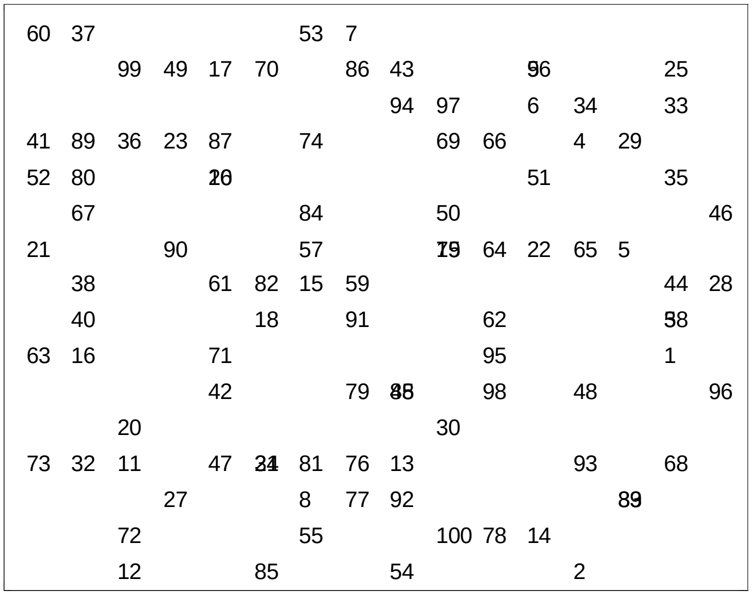


Figure 4.5 Plot for problem 9 from Problem Set 2.

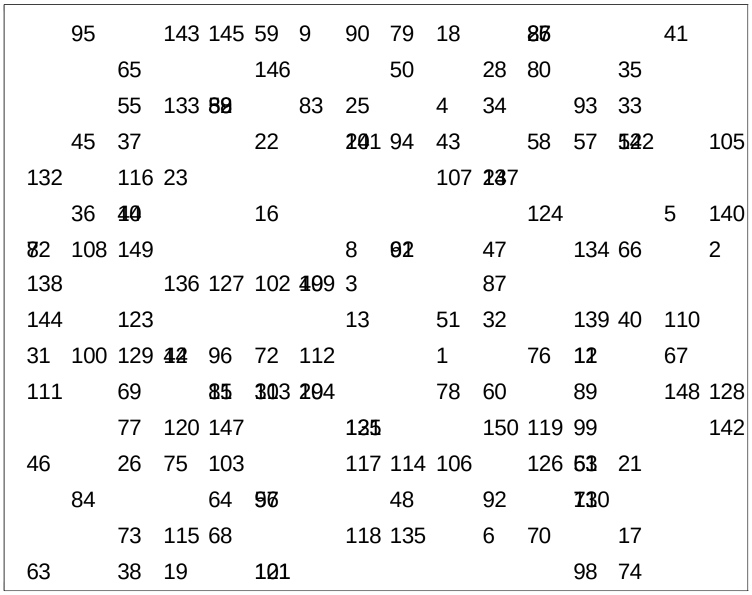


Figure 4.6 Plot for problem 4 from Problem Set 3.

In Table 4.7, we show the results for problem sets 1, 2, and 3. We do not know the global optimal value for these problems and so we cannot compare the results we obtained. We observed though that the frequencies for the best solutions are very low. The frequencies are all less than 10, expect for problem six of problem set 2, which has a frequency of 17. The average running times are 4.33 seconds, 15.73 seconds, and 36.65 seconds, for problem sets 1, 2, and 3, respectively. In Figure 4.4 we show a plot of the final result obtained by DACN for problem two of problem set 1. This plot and all other plots in this thesis are produced using Matlab 7.0 (Sigmon and Davis, 2002). In Figures 4.5 and 4.6, we show the plots for problem nine of problem set 2 and problem four of problem set 3, respectively.

	50-point problems			100-point problems			150-point problems		
Problem	Best solution	Freq	Running time (secs)	Best solution	Freq	Running time (secs)	Best solution	Freq	Running time (secs)
1	44880.1	1	4.82	222313	1	18.76	524053	1	48.29
2	41767.4	1	5.34	212677	2	18.04	555620	1	44.65
3	50730.6	2	5.08	245370	1	18.88	572832	1	44.33
4	44550.6	1	5.05	213892	1	17.63	505089	1	42.81
5	48698.9	1	4.99	227596	1	18.36	584245	1	40.61
6	51246.5	1	4.96	226109	1	19.36	510668	1	51.30
7	32953.6	4	4.86	168891	4	18.11	483936	1	41.59
8	45511.4	2	4.81	179716	1	20.20	475717	1	39.45
9	50341.1	1	5.10	241309	1	19.19	572519	1	47.86
10	53649.7	5	4.87	229803	3	18.41	556993	2	41.98
Average			4.99			18.69			44.29

Table 4.8 Results for problem sets 4, 5, and 6 for DACN. These problem sets are originally in four dimensions.

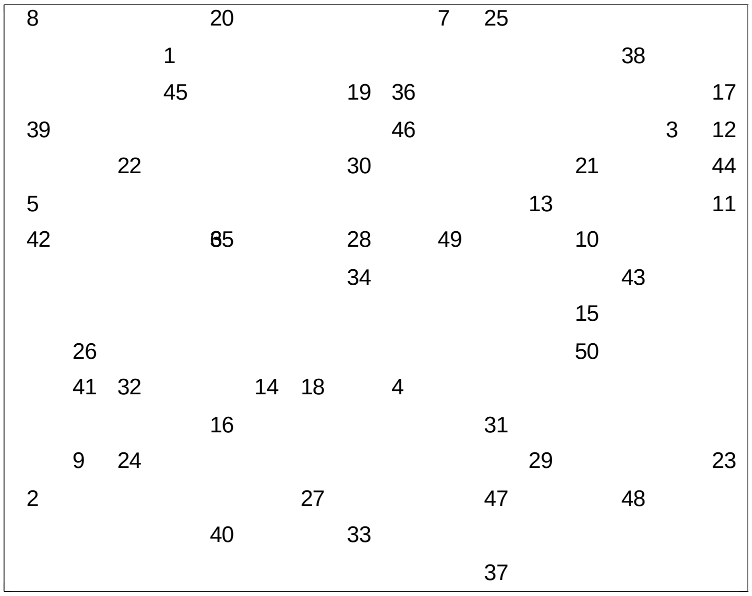


Figure 4.7 Plot for problem 6 from Problem Set 4.

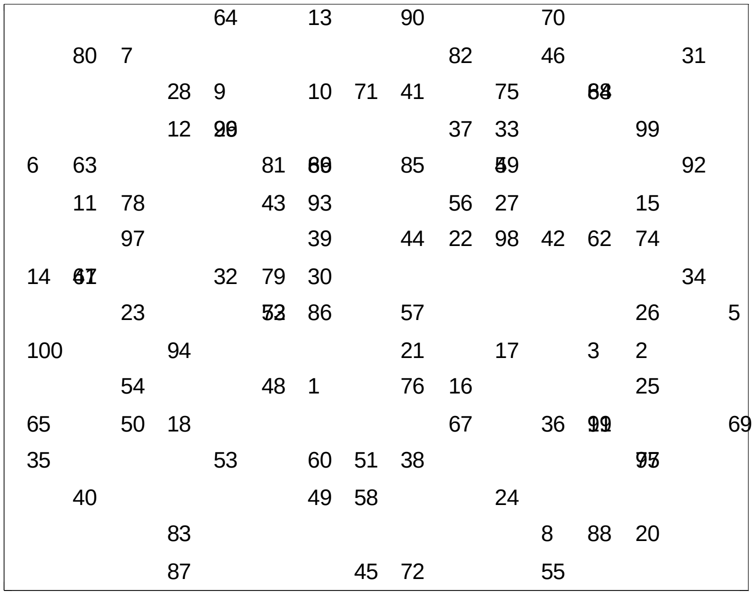


Figure 4.8 Plot for problem 7 from Problem Set 5.

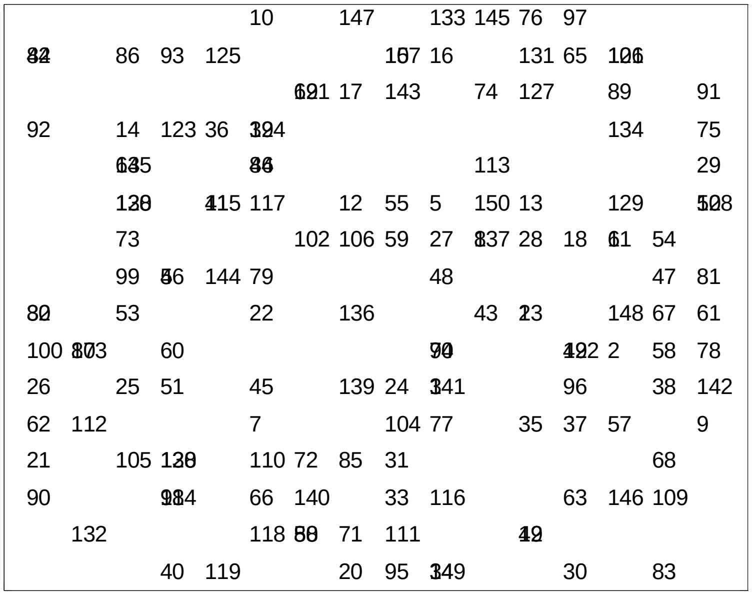


Figure 4.9 Plot for problem 5 from Problem Set 6.

In Table 4.8, we show the results for the data sets originally in four dimensions, that is, problem sets 4, 5, and 6. The average running times for the 50-point problems, the 100-point problems, and the 150-point problems, originally in four dimensions, are 4.99 seconds, 18.69 seconds, and 44.29 seconds, respectively. In Figure 4.7, we show the plot of the final result obtained by DACN for problem six of problem set 4. We show the plot for problem seven of problem set five in Figure 4.8 and the plot for problem five of problem set six in Figure 4.9.

In Table 4.9, we show the results for problem sets 7, 8, and 9, (the data sets originally in five dimensions). For the 50-point problems, 100-point problems, and 150-point problems, originally in five dimensions, the average running times are 4.66 seconds, 17.41 seconds, and 40.27 seconds, respectively. In Figures 4.10, 4.11, and 4.12, we show the plots of the final results produced by DACN for problem 1 of problem set 7, problem eight of problem set 8, and problem 10 of problem set 9, respectively.

	50-point problems			100-point problems			150-point problems		
Problem	Best solution	Freq	Running time (secs)	Best solution	Freq	Running time (secs)	Best solution	Freq	Running time (secs)
1	75247.8	1	4.57	325175	1	17.81	802908	1	38.22
2	64814.3	1	5.24	333587	1	18.68	841420	1	39.86
3	78566.4	1	4.45	348777	1	17.31	860219	1	41.67
4	75835.4	1	4.74	356028	1	17.25	812147	1	40.28
5	72817.3	1	4.39	329985	1	15.57	813606	2	40.68
6	81840.3	1	4.64	360491	1	18.31	856291	1	40.79
7	64342.7	1	4.46	297337	1	16.99	747779	1	37.02
8	71669.1	1	4.49	305608	1	17.46	753434	1	43.03
9	85219.9	1	4.76	368939	1	16.49	877125	1	39.62
10	78102.9	2	4.82	319882	1	18.24	784065	1	41.53
Average			4.66			17.41			40.27

Table 4.9 Results for problem sets 7, 8, and 9 for DACN. These problem sets are originally in five dimensions.

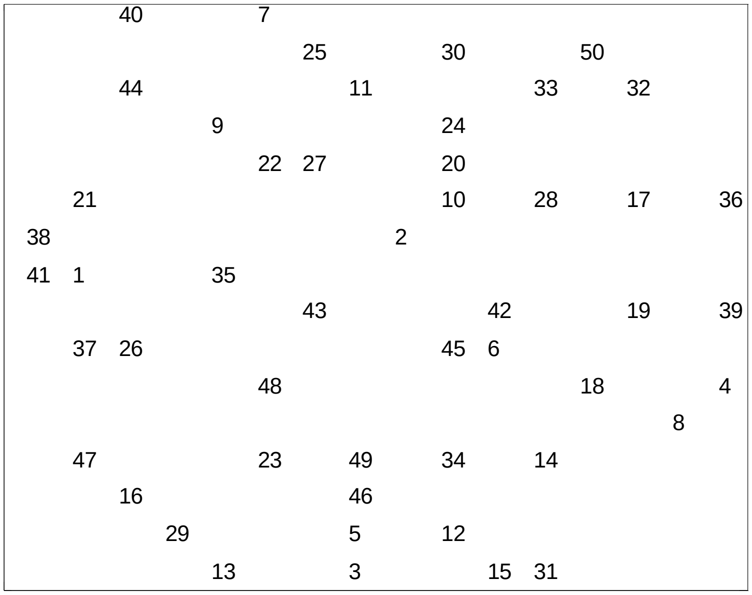


Figure 4.10 Plot for problem 1 from Problem Set 7.

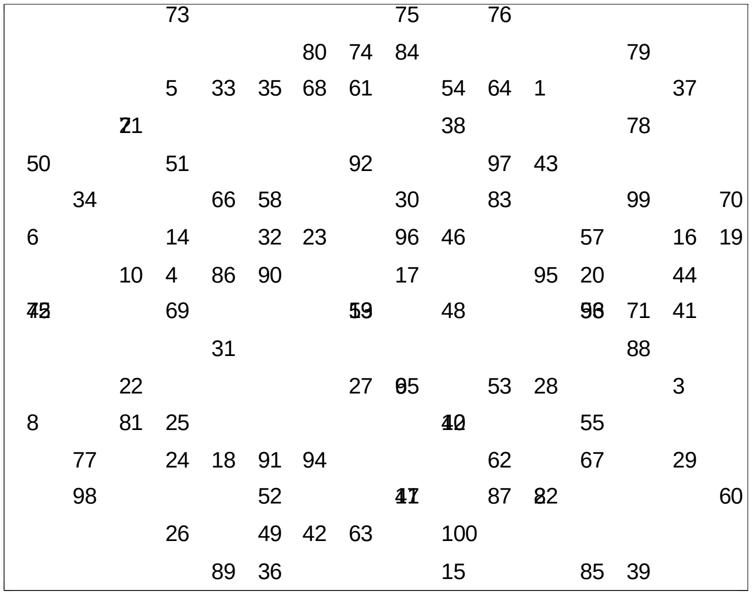


Figure 4.11 Plot for problem 8 from Problem Set 8.

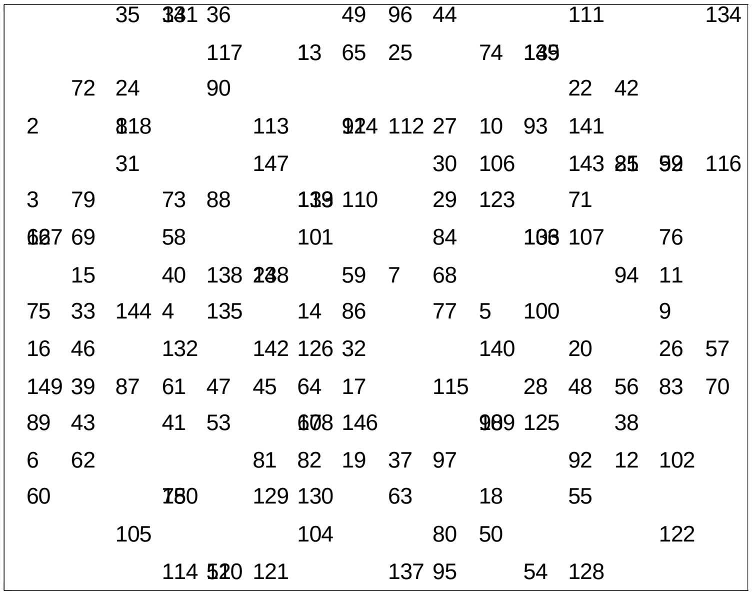


Figure 4.12 Plot for problem 10 from Problem Set 9.

4.3 Conclusions

DACN provides an approximate solution to the data visualization problem in a small amount of computing time. For the problem sets originally in two-dimensions, DACN produces the global optimum in 28 of the 30 problems. For the other problem sets, the global optimal solutions are unknown. In the remaining chapters, we will use other algorithms on these problem sets and then compare these results to the solutions generated by DACN.

Chapter 5: Mathematical Programming

5.1 Integer Problem Formulation

A QAP may be formulated as an equivalent integer program (IP). In general, the equivalent IP for a QAP is much larger than the QAP. However, an IP does not involve any complicated quadratic coefficients, which seems to destroy any hope of finding efficient solution methods. Also, high-quality commercial solvers are available to solve IP, and so the larger IP can be solved faster than the smaller QAP.

Let y_{ij}^{kl} be a binary variable that is equal to one when $i \in M$ is assigned to $k \in N$ and $j \in M$ is assigned to $l \in N$, and zero otherwise. The QAP formulation of DVP can be expressed as the following linear integer program (IP):

$$\text{Minimize} \sum_{i \in M} \sum_{\substack{j \in M \\ j > i}} \sum_{k \in N} \sum_{l \in N} [od(i, j) - nd(k,l)]^2 y_{ij}^{kl} \tag{5.1}$$

subject to

$$\sum_{k \in N} x_{ik} = 1, \forall i \in M \tag{5.2}$$

$$\sum_{k \in N} \sum_{l \in N} y_{ij}^{kl} = 1, \forall\, i, j \in M,\ j > i \tag{5.3}$$

$$x_{ik} = \sum_{l \in N} y_{ij}^{kl}, \forall i, j \in M,\ j > i \tag{5.4}$$

$$x_{ik} = \sum_{l \in N} y_{ji}^{lk}, \forall i, j \in M,\ j < i \tag{5.5}$$

$$x_{ik} \in (0,1) \tag{5.6}$$

$$y_{ij}^{kl} \in (0,1). \tag{5.7}$$

This IP has $\frac{m(m-1)n^2}{2}$ *y* variables and *mn x* variables. It also has *m* assignment constraints for the *x* variables, $\frac{m(m-1)}{2}$ assignment constraints for the *y* variables, and mn^2 other constraints. Thus, there are a total of $mn\left(1+\frac{n(m-1)}{2}\right)$ variables and $m\left(1+n^2+\frac{m-1}{2}\right)$ constraints in the IP formulation. The IP is much larger than QAP which has *mn* variables and *m* assignment constraints.

The constraints in (5.2) are for the *x* variables and are the same as in the QAP. These are simple assignment constraints that require each point in *M* to be assigned to a point in *N*. The constraints in (5.3) are for the *y* variables and are also assignment constraints. These constraints require that a pair of points in *M* can only be assigned to a pair of points in *N*. The constraints in (5.4) and (5.5) ensure that if a point $i \in M$ is not assigned to a lattice point $k \in N$ ($x_{ik} = 0$), then, for all *y* variables that include *i*, point *i* cannot be assigned to point $k \in N$ $\left(\text{that is, } y_{i.}^{k.} = 0\right)$.

Next, we establish that the QAP and IP formulations are equivalent.

Theorem

The QAP formulation given in (3.5) – (3.7), and the IP formulation given in (5.1) – (5.7) are equivalent.

Proof

The proof has two parts. In the first part, we show that any solution to the IP is also a solution to the QAP and vice versa. In the second part, we show that the two formulations have the same objective function value.

Let (*x*,*y*) be any feasible solution to the IP formulation. Then (*x*,*y*) satisfies all the constraints (5.2) – (5.5) for the IP formulation. In particular, in (5.2), we have $\sum_{k \in N} x_{ik} = 1, \forall i \in M$, which is the same as (3.6) in the QAP formulation. Thus, if (*x*,*y*) is a feasible solution to the IP, then *x* is a feasible solution to the QAP formulation.

Conversely, let *x* be a feasible solution to the QAP formulation. Let $y_{ij}^{kl} = x_{ik} x_{jl}$, $\forall$ *i*, *j* $\in$ *M*, *j* > *i*, and $\forall$ *k*, *l* $\in$ *N*. Then *x* satisfies

$$\sum_{k \in N} x_{ik} = 1, \forall i \in M \tag{5.8}$$

as it is a solution to the QAP formulation.

From (5.8)

$$\sum_{l \in N} x_{jl} = 1, \forall j \in M.$$

Hence,

$$\begin{aligned}
&\sum_{k \in N} x_{ik} \sum_{l \in N} x_{jl} = 1, \forall i, j \in M \\
&\Rightarrow \sum_{k \in N} \sum_{l \in N} x_{ik} x_{jl} = 1, \forall i, j \in M \\
&\Rightarrow \sum_{k \in N} \sum_{l \in N} y_{ij}^{kl} = 1, \forall i, j \in M \\
&\Rightarrow \sum_{k \in N} \sum_{l \in N} y_{ij}^{kl} = 1, \forall i, j \in M, j > i.
\end{aligned}$$

Also,

$$x_{ik}x_{jl} = y_{ij}^{kl}, \forall i, j \in M, k, l \in N$$
$$\Rightarrow \sum_{l\in N} x_{ik}x_{jl} = \sum_{l\in N} y_{ij}^{kl}, \forall i, j \in M, k \in N$$
$$\Rightarrow x_{ik}\sum_{l\in N} x_{lj} = \sum_{l\in N} y_{ij}^{kl}, \forall i, j \in M, k \in N$$
$$\Rightarrow x_{ik} = \sum_{l\in N} y_{ij}^{kl}, \forall i, j \in M, k \in N$$
$$\Rightarrow x_{ik} = \sum_{l\in N} y_{ij}^{kl}, \forall i, j \in M, i < j, k \in N$$
$$\text{and } x_{ik} = \sum_{l\in N} y_{ji}^{lk}, \forall i, j \in M, i > j, k \in N.$$

Thus, if *x* is a feasible solution to the QAP formulation, then (*x*,*y*) is a feasible solution to the IP formulation.

Next, we show that the objective function values are equal. Let (*x*,*y*) be any feasible solution to the IP formulation. By definition, $y_{ij}^{kl} = 1,$ implies that point *i* is assigned to lattice point *k*, so x_{ik} = 1. Also, point *j* is assigned to lattice point *l*, so x_{jl} = 1. Therefore, if $y_{ij}^{kl} = 1,$ then x_{ik} x_{jl} = 1 On the other hand, $y_{ij}^{kl} = 0$ means the pair of points (*i*, *j*) are not assigned to the pair of lattice points (*k*, *l*). Therefore, at least one of the following holds: x_{ik} = 0 or x_{jl} = 0 when $y_{ij}^{kl} = 0,$ so we have x_{ik} x_{jl} = 0. Therefore, $y_{ij}^{kl} = x_{ik}x_{jl}, \forall i, j \in M, j > i, \forall k, l \in N.$ Hence, the objective function values are equal for the two formulations. This completes the proof that the two formulations are equivalent.

In the IP formulation, the *y* variables are integer variables (see (5.7)). However, the *y* variables will be integer, if the *x* variables are integer (Cela, 1998). Therefore, the *y* variables can be relaxed to real variables. (5.7) then becomes $0 < y_{ij}^{kl} < 1.$

Problem	Running time for 4 lattice points (seconds)	Running time for 16 points (seconds)	Running time for 64 points (seconds)
1	0.50	2.56	295.53
2	0.41	16.79	489.85
3	0.10	12.67	183.92
4	0.56	1.76	563.64
5	0.73	19.34	256.13
6	0.52	23.96	217.87
7	0.87	38.50	5088.29
8	0.59	17.24	1101.91
9	0.72	58.12	7577.17
10	0.50	18.16	195.96
Average	0.55	20.91	1597.03

Table 5.1 Running times for the IP for problem sets with *m* = 10 and *n* = 4, 16, and 64.

5.2 Preliminary Computational Results

We implemented our IP formulation in ILOG OPL Studio 3.5.1 (Hentenryck, 1999) and ran our experiments using Windows 2000 with an 800 MHz Pentium III processor and 512 MB of RAM. We tested our formulation on 10 problem sets each of size 10. We used values of *n* equal 4, 16, and 64.

In Table 5.1, we show the running times for our IP formulation on these problem sets. The average running times for *n* = 4, 16, and 64 are 0.55 seconds, 20.91 seconds, and 1597.03 seconds, respectively. It appears that even for problems with a small value of *m*, the running time gets very large as *n* increases. Therefore, as *m* and *n* increase, the running time of our IP formulation increases substantially.

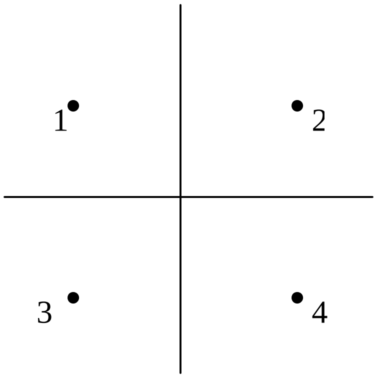


Figure 5.1 Lattice of four points.

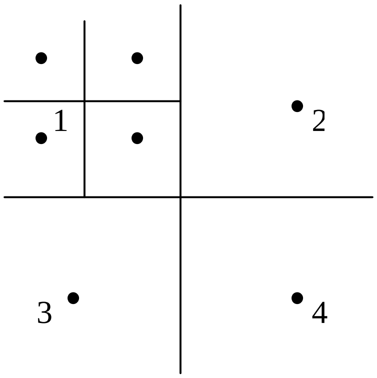


Figure 5.2 Lattice with four initial points and point one subdivided into four additional points.

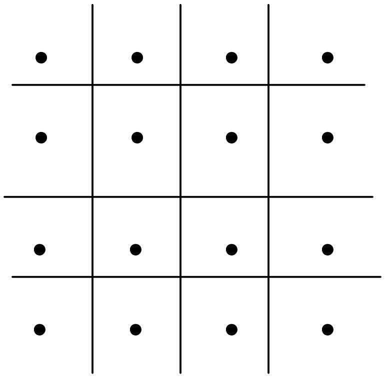


Figure 5.3 Lattice with 16 points after the four initial points have been subdivided into four additional points each.

Next, we developed a divide-and-conquer heuristic that solves a set of smaller problems at each stage instead of one large problem. This gives a more manageable problem to solve at each stage. There are five steps to our divide-and-conquer heuristic.

1. Start with a lattice of four points and solve the IP to assign points in *M* to these four lattice points (see Figure 5.1). Points are now divided into four quadrants.

2. Divide quadrant one into four points (see Figure 5.2).

3. Assign points in quadrant one from the assignment in step 1 to the four new points while keeping assignments to the other quadrants fixed.

4. Repeat Steps 2 and 3 for the other three quadrants (see Figure 5.3).

5. Continue dividing each point into four points and repeat until a stopping rule is met. We stop at 256 points. At each stage, only four lattice points are considered.

Problem	Running time for 5 points (seconds)	Running time for 10 points (seconds)	Running time for 15 points (seconds)	Running time for 20 points (seconds)
1	0.03	0.50	5.97	13.12
2	0.02	0.41	0.33	8.65
3	0.03	0.10	5.96	31.47
4	0.05	0.56	3.84	81.82
5	0.03	0.73	4.67	48.88
6	0.02	0.52	4.77	22.72
7	0.03	0.87	9.69	15.65
8	0.03	0.59	0.30	25.32
9	0.03	0.72	4.57	59.71
10	0.04	0.50	5.97	83.68
Average	0.03	0.55	4.61	39.10

Table 5.2 Running times for the IP for problem sets with
m = 5, 10, 15, and 20, and *n* = 4.

We also tested our IP formulation for different sizes of *m* with *n* = 4. We use *m* = 5, 10, 15, and 20. In Table 5.2, we show the running times for these problems. The average running times for *m* = 5, 10, 15, and 20 are 0.03 seconds, 0.55 seconds, 4.61 seconds, and 39.10 seconds, respectively. The running time increases substantially as *m* increases.

Next, we developed an algorithm to divide *M* into several smaller sets and assign the points in these sets, one after another, instead of assigning all the points in *M* at the same time. If all the points in *M* are assigned at once, the solution obtained will most probably be more accurate than when we assign the points in smaller sets. However, we use smaller sets to reduce the running time. We want these smaller sets to be as large as possible in order to obtain more accurate solutions. Preliminary experiments indicated that with respect to size and running time *m* = 10 gives reasonable results. Based on these results, we group points into sets of 10 and then assign one set at a time. After the first set has been located, these assignments are kept fixed while the next set of 10 is assigned. We continue until all points have been located. Each set of 10 is selected randomly.

Our IP algorithm for *n* = 4 is given by the following.

1. Start with 10 points and solve the IP to fix their locations.

2. Keeping the assignments of these located points fixed, select 10 more points from the remaining points and solve IP to fix them.

3. Repeat Step 2 until all points are assigned to one of the four lattice points.

5.3 Integer Programming Heuristics

We applied our IP heuristic to several data sets with 50, 100, and 150 points. These are the same data sets generated from a lattice of 256 points in two dimensions used in our local search experiments. We used problems generated from a two-dimensional lattice, because it is easy to compare the quality of the computational results when the original set of points is in two dimensions. In this case, the optimal objective function value is known and equal to zero. We investigated five IP heuristics and these are described in Table 5.3.

Algorithm	Description
IP	Integer program heuristic
IR	Integer program heuristic - Step 1 repeated
IRN	Integer program heuristic - Step 1 repeated - Points allowed to move to neighboring lattice points
IRNS	Integer program heuristic - Step 1 repeated - Points allowed to move to neighboring lattice points - Maximum of 20 points considered in reassigning points after Step 2 - After Step 4 points reassigned five points at a time
IMP	Integer program heuristic - Uses final results from DACN as starting solution - Reassigns points randomly, five points at a time

Table 5.3 Integer programming heuristics.

5.3.1 IP Heuristic

In Table 5.4, we show the results for the experiments for the IP heuristic (IP) for our 50-point, 100-point, and 150-point problem sets. In all 30 problems, IP does not find the global optimum. The quality of the results obtained is very poor. The objective function values are very large and are not close to the global objective function value of zero. The average running times for the 50-point, 100-point, and 150-point problem sets are 40.08 seconds, 60.47 seconds, and 82.50 seconds, respectively. These running times are much higher than those for our local search experiments. We consider some improvements to our IP heuristic that may reduce running time and find an objective function value closer to zero.

	50-point problems		100-point problems		150-point problems	
Problems	Best solution obtained	Running time (seconds)	Best solution obtained	Running time (seconds)	Best solution obtained	Running time (seconds)
1	1782.55	38.76	3943.15	60.57	10543.39	81.11
2	1065.33	43.17	4787.36	61.79	9474.23	84.18
3	1582.72	38.58	7555.24	59.69	15425.09	81.19
4	1432.41	40.67	4930.43	59.27	11236.00	81.79
5	1380.65	40.05	4023.72	57.40	8970.13	83.28
6	3486.62	40.86	9526.07	61.84	18000.82	82.60
7	1186.39	35.77	5731.94	61.25	12520.92	81.71
8	2167.78	43.89	7458.12	62.61	14464.60	83.20
9	1331.17	40.13	5125.06	59.12	9418.85	82.55
10	896.44	38.88	4609.96	61.20	10782.77	83.36
Average		40.08		60.47		82.50

Table 5.4 Results for problem sets originally in two dimensions for IP.

We tried different solver options available in ILOG OPL but none yielded substantially reduced running times. Since the method used here is an approximation approach, we experimented with solving an LP relaxation for Steps 2, 3, and 4, instead of solving the IP. Also, we considered solving the problem to within a fixed percentage of the optimal objective function value. These attempts did not achieve any noticeable reduction in the running time.

5.3.2 IR Heuristic

	50-point problems		100-point problems		150-point problems	
Problems	Best solution obtained	Running time (seconds)	Best solution obtained	Running time (seconds)	Best solution obtained	Running time (seconds)
1	1506.81	38.21	3826.07	70.41	4132.63	132.57
2	1065.33	43.32	6073.60	69.28	9213.26	93.50
3	1332.33	40.11	4587.99	71.14	6586.88	105.14
4	1076.72	41.05	2884.42	66.51	9886.35	97.80
5	1216.70	45.10	1496.15	65.31	2863.98	92.93
6	1021.26	41.41	2782.36	71.01	2255.72	105.39
7	1035.80	38.62	5505.76	69.26	9981.09	94.84
8	1946.48	42.16	6387.51	72.64	8029.16	118.57
9	1048.95	43.85	3401.99	65.80	9900.15	94.37
10	639.72	41.36	2166.36	73.12	3530.58	103.08
Average		41.52		69.45		103.82

Table 5.5 Results for problem sets originally in two dimensions for IR.

We observed that the fewer the number of points assigned incorrectly in Step 1, the better the final solution. Therefore, to improve the objective function values, we propose an IP heuristic with Step 1 repeated a number of times. This heuristic is denoted IR. After all points in *M* have been assigned in Step 1, we randomly free points, 10 at a time, and try to assign them with the other *m* – 10 points fixed at their assigned locations. We repeat this until we have a cycle of all *m* points being reassigned with no change in the objective function value.

In Table 5.5, we show the results for IR on the same data sets we used for IP. For all 30 problems, IR does not find the global optimum. For all the 50-point problems though, IR produces better results than IP. That is, the objective function values are lower than those for IP. For nine of the ten 100-point problems and nine of the ten 150-point problems, IR gives better results than IP. The average running times are 41.52 seconds, 69.45 seconds, and 103.82 seconds, for the 50-point problems, 100-point problems, and 150-point problems, respectively. The average running times for IR are all slightly higher than those for IP, but we obtain better objective function values with IR.

5.3.3 IRN Heuristic

Even though we obtain better solutions with IR, the quality of the solutions is still poor as it does not find the global optimum for any problem. With IR, a point remains in the quadrant to which it is originally assigned. If a point is placed in a wrong quadrant at any step, this error cannot be corrected and this may contribute significantly to the poor performance. To eliminate this, we propose an IP heuristic (IRN) that allows points to move to neighboring points after they have been assigned.

The algorithm for IRN is similar to that for IR. However, after Steps 2, 3, and 4 in IRN, points assigned to a box are freed and may be reassigned to neighboring points, while points assigned to the other boxes are kept fixed at their current assignments. Recall, that each lattice point is divided into four subpoints after each step. A box is made up of the four subpoints, from a point in a previous step. After Step 2, one box is made up of the four points in quadrant one as shown in Figure 5.2. The neighborhood structure here is the same as that described for DACN. We continue to reassign points in boxes until there is no improvement in the objective function value after we have gone through a cycle with all of the points in *M*.

	50-point problems		100-point problems		150-point problems	
Problems	Best solution obtained	Running time (seconds)	Best solution obtained	Running time (seconds)	Best solution obtained	Running time (seconds)
1	0.00	154.03	2910.35	526.53	0.00	400.92
2	949.31	163.84	0.00	363.21	0.00	498.41
3	645.63	174.37	0.00	211.78	0.00	410.50
4	0.00	128.45	631.23	216.33	1813.38	584.38
5	124.87	136.78	0.00	161.72	0.00	339.82
6	0.00	166.78	0.00	175.80	0.00	422.39
7	567.56	146.79	1391.56	338.25	0.00	458.43
8	844.51	363.34	3466.19	335.73	0.00	427.25
9	618.57	188.31	0.00	210.65	0.00	459.86
10	95.44	139.05	0.00	206.26	0.00	502.32
Average		176.17		274.63		450.43

Table 5.6 Results for problem sets originally in two dimensions for IRN.

We applied IRN to our three two-dimensional data sets. In Table 5.6, we show the results for IRN on the 50-point, 100-point, and 150-point problem sets. In all 30 problems, IRN gives better results than IR. For the 50-point problems, IRN finds the global optimum in three of the 10 problems. In six of 10 problems of size 100 and nine of the 10 problems of size 150, IRN finds the global optimum. The average running times are 176.17 seconds, 274.63 seconds, and 450.43 seconds for the 50-point, 100-point, and 150-point problems, respectively. The average running times for IRN are much higher than those for IP and IR, but the quality of the solutions is much better.

5.3.4 IRNS Heuristic

	50-point problems		100-point problems		150-point problems	
Problems	Best solution obtained	Running time (seconds)	Best solution obtained	Running time (seconds)	Best solution obtained	Running time (seconds)
1	0.00	134.21	3019.37	332.15	0.00	252.12
2	892.27	155.96	0.00	324.33	0.00	310.17
3	645.63	142.71	0.00	196.28	0.00	271.29
4	0.00	113.14	631.23	187.03	1813.38	356.96
5	124.87	109.71	0.00	147.23	0.00	222.55
6	723.83	128.03	0.00	164.00	0.00	249.44
7	567.56	124.21	1391.56	274.62	0.00	297.56
8	673.83	225.41	0.00	206.54	0.00	312.19
9	618.57	141.08	0.00	159.63	0.00	267.90
10	95.44	118.60	0.00	151.66	0.00	273.37
Average		139.31		214.35		281.36

Table 5.7 Results for problem sets originally in two dimensions for IRNS.

After Step 2, when we free up the points assigned to a box, there are only four boxes to consider. The number of points assigned to a box can be relatively large when the size of *m* is greater than 50. In our preliminary computational experiments, we observed that the best results and running times were obtained when we restricted the maximum number of points that can be reassigned to 20.

The presence of empty boxes will result in a situation where time is spent in a box only to find that it there are no points assigned to them. After Step 4, there may be many empty boxes. To eliminate this, we refined the heuristic so that points assigned to boxes were not examined after Step 4. We select five points randomly and try to reassign them keeping all other points fixed at their current assignments. We then take five points randomly from the remaining *m* – 5 points and repeat the procedure until all points have been reassigned. Five points appears to produce fast running times and good results.

We apply this heuristic, denoted by IRNS, to our two-dimensional data sets. We show the results for these experiments in Table 5.7. IRNS finds the global optimum in two of the 10 problems of size 50. IRNS generates a better solution than IRN in two of the 50-point problems. For seven of the other 50-point problems, IRNS generates the same best solution as IRN. In seven of the 10 problems of size 100, IRNS finds the global optimum. IRNS generates a better solution than IRN in one of the 100-point problems. For eight of the other 100-point problems, IRNS generates the same best solution as IRN. IRNS finds the global optimum in nine of the ten 150-point problems. In all ten 150-point problems, IRN and IRNS generate the same solutions. The average running times for the 50-point, 100-point, and 150-point problems for IRNS are 139.31 seconds, 214.35 seconds, and 281.36 seconds, respectively.

IRN and IRNS produce similar results, but IRNS generates these results much faster than IRN.

5.4 Comparison of DACN and IRNS

	50-point problems			100-point problems			150-point problems		
Problems	Best solution obtained	Running time (seconds)	Percent increase over DACN results	Best solution obtained	Running time (seconds)	Percent increase over DACN results	Best solution obtained	Running time (seconds)	Percent increase over DACN results
1	21491.62	177.72	2.42	101621.10	442.52	0.16	248055.77	733.37	0.20
2	17157.00	184.29	5.58	106738.70	654.59	1.46	281965.46	451.45	6.01
3	26406.12	161.91	2.07	133297.12	457.75	5.13	306564.78	531.88	3.27
4	23122.98	205.00	2.09	121084.26	337.91	24.64	261537.31	580.15	5.99
5	30227.63	189.27	8.29	133698.21	355.42	0.02	324271.34	717.25	5.93
6	22630.76	158.20	0.09	101322.81	580.17	0.00	244848.28	564.43	0.03
7	15287.25	110.33	2.23	82577.58	261.88	0.30	266101.82	680.56	0.80
8	16684.51	211.40	3.61	85611.87	276.26	1.26	231903.50	505.57	0.39
9	22139.66	236.94	1.66	116188.76	326.45	0.46	300627.86	615.66	0.26
10	28797.60	185.97	7.65	91150.22	320.05	0.18	261777.81	546.30	0.23
Average		182.10	3.57		401.30	3.36		592.66	2.31

Table 5.8 Results for problem sets 1, 2, and 3 for IRNS. These problem sets are originally in three dimensions.

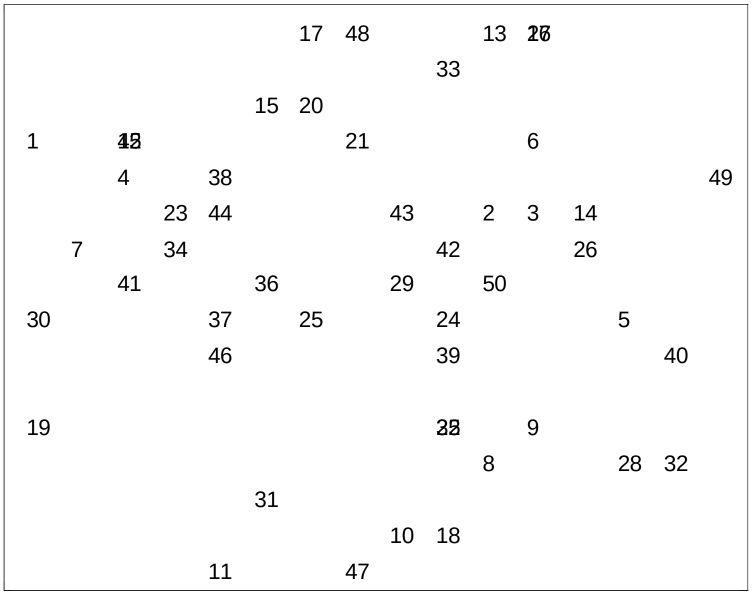


Figure 5.4 Plot for problem 2 from problem set 1.

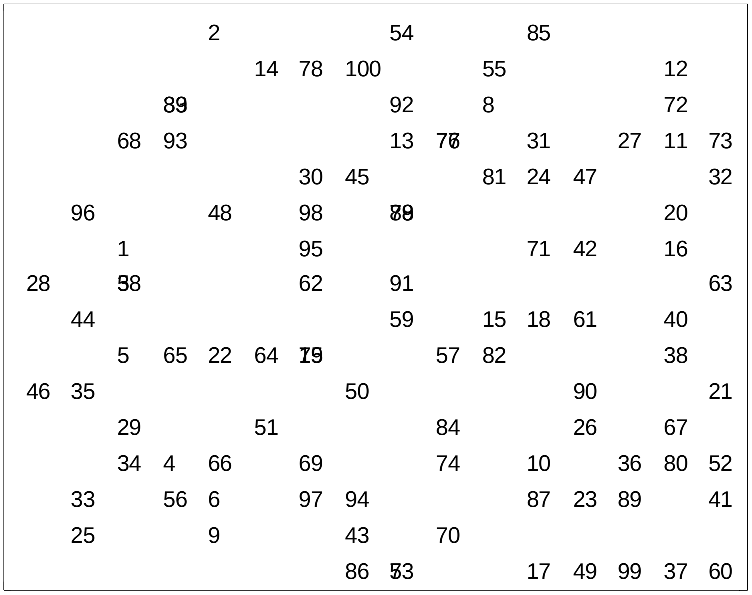


Figure 5.5 Plot for problem 9 from problem set 2.

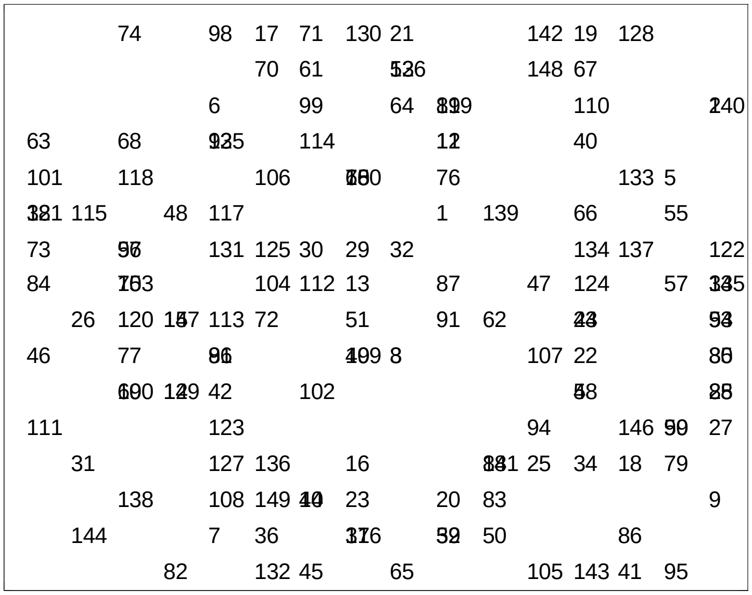


Figure 5.6 Plot for problem 4 from problem set 3.

We applied IRNS to problem sets 1 to 9 in Table 4.6. In Table 5.8, we show the results for problem sets 1, 2, and 3, that is, the data sets originally in three dimensions. IRNS produces the same solution as DACN for Problem 6 of the 100-points problems. For the other nine problems of size 100, and all 50-point and 150-point problems, DACN produces better solutions than IRNS. IRNS produces objective function values that are on the average 3.57%, 3.36%, and 2.31%, larger than the objective function values produced by DACN for the 50-point problems, 100-point problems, and 150-point problems, respectively.

In Figures 5.4, 5.5, and 5.6, we show the plots of the final results for IRNS for problem two of problem set 1, problem nine of problem set 2, and problem four of problem set 3, respectively. These plots are quite different from the plots produced by DACN (see Figures 4.4, 4.5, and 4.6).

The average running times are 182.10 seconds, 401.30 seconds, and 592.66 seconds, for the 50-point problems, 100-point problems, and 150-point problems, respectively. These running times are much longer than those for our local search experiments. For the three-dimensional data sets, DACN produces solutions with lower objective function values and much faster times than IRNS.

In Table 5.9, we show the results for the four-dimensional data sets, that is, problem sets 4, 5, and 6. For all 30 four-dimensional problems, the results produced by IRNS are worse than those produced by DACN. The objective function values produced by IRNS are greater than those produced by DACN by an average of 3.12%, 1.61%, and 4.15%, for the 50-point problems, 100-point problems, and 150-point problems, respectively. The average running times for the 50-point, 100-point, and 150-point problems are 216.31 seconds, 459.76 seconds, and 771.41 seconds, respectively. For these problem sets, DACN produces better objective function values than IRNS in much faster times.

In Figures 5.7, 5.8, and 5.9, respectively, we show the plots of the final results for IRNS for problem six of problem set 4, problem seven of problem set 5, and problem five of problem set 6, respectively. As expected these plots are different from those produced from DACN (see Figures 4.7, 4.8, and 4.9) since the objective function values produced by IRNS are different from those produced by DACN.

	50-point problems			100-point problems			150-point problems		
Problems	Best solution obtained	Running time (seconds)	Percent increase over DACN results	Best solution obtained	Running time (seconds)	Percent increase over DACN results	Best solution obtained	Running time (seconds)	Percent increase over DACN results
1	47094.86	135.71	4.93	224839.56	423.42	1.14	525050.19	551.05	0.19
2	42963.42	178.16	2.86	213458.02	580.74	0.37	559160.02	759.19	0.64
3	50975.45	272.80	0.48	247202.91	300.85	0.75	585377.67	693.63	2.19
4	44698.41	279.72	0.33	218369.49	575.95	2.09	514418.71	971.90	1.85
5	51032.22	291.84	4.79	228124.29	377.06	0.23	616474.44	688.62	5.52
6	58262.64	165.14	13.69	241455.22	443.61	6.79	541269.10	1169.47	5.99
7	33486.43	208.14	1.62	169025.90	274.87	0.08	489496.21	798.00	1.15
8	46545.88	219.97	2.27	183680.72	810.09	2.21	525759.25	746.29	10.52
9	51218.00	168.82	1.74	244400.01	319.68	1.28	573959.66	540.31	0.25
10	53901.03	242.77	0.47	232392.88	491.32	1.13	630463.38	855.60	13.19
Average		216.31	3.12		459.76	1.61		777.41	4.15

Table 5.9 Results for problem sets 4, 5, and 6 for IRNS. These problem sets are originally in four dimensions

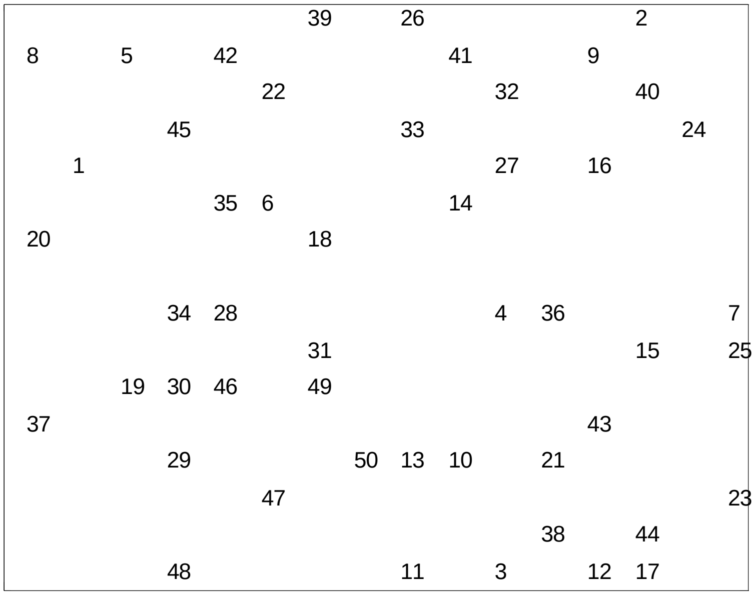


Figure 5.7 Plot for problem 6 from problem set 4.

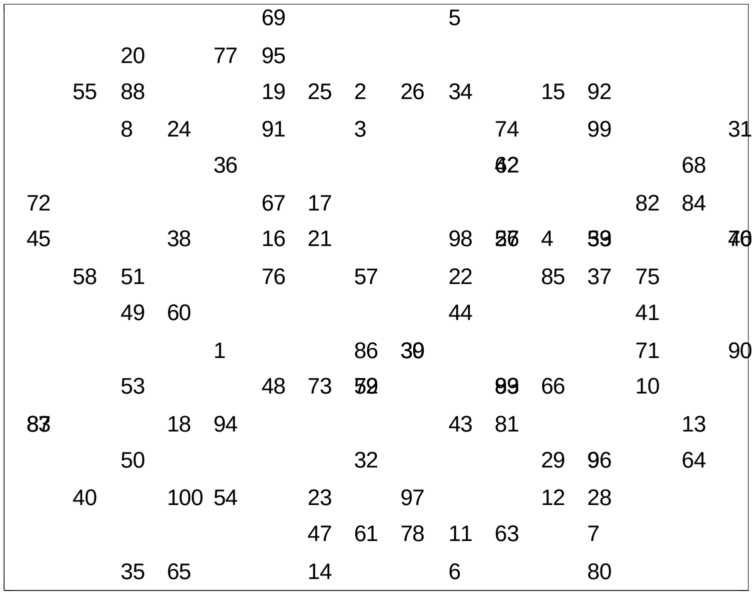


Figure 5.8 Plot for problem 7 from problem set 5.

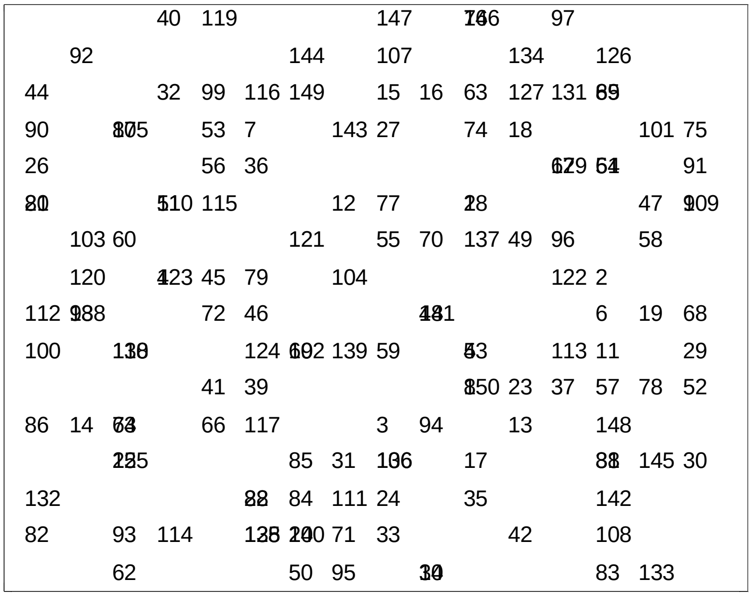


Figure 5.9 Plot for problem 5 from problem set 6.

	50-point problems			100-point problems			150-point problems		
Problems	Best solution obtained	Running time (seconds)	Percent increase over DACN results	Best solution obtained	Running time (seconds)	Percent increase over DACN results	Best solution obtained	Running time (seconds)	Percent increase over DACN results
1	82272.38	240.02	9.34	333816.59	905.08	2.66	837254.19	899.66	4.28
2	75471.14	259.10	16.44	341377.09	731.34	2.34	909307.80	699.35	8.07
3	80991.64	193.99	3.09	354212.62	568.34	1.56	873020.12	725.03	1.49
4	76126.05	174.75	0.38	367179.84	825.62	3.13	836426.00	1408.30	2.99
5	75144.71	205.47	3.20	347951.62	1151.31	5.44	822921.03	645.31	1.14
6	82056.33	198.77	0.26	372684.06	460.07	3.38	874079.30	745.46	2.08
7	73751.18	149.94	14.62	302035.76	394.86	1.58	751511.17	729.28	0.50
8	86034.34	277.59	20.04	310912.96	419.77	1.74	782511.42	1127.75	3.87
9	86484.18	265.01	1.48	371934.41	397.75	0.81	888507.91	655.37	1.30
10	78393.78	215.79	0.37	321339.57	403.45	0.46	816186.90	1028.75	4.10
Average		218.04	6.92		625.76	2.31		866.43	2.98

Table 5.10 Results for problem sets 7, 8, and 9 for IRNS. These problem sets are originally in five dimensions.

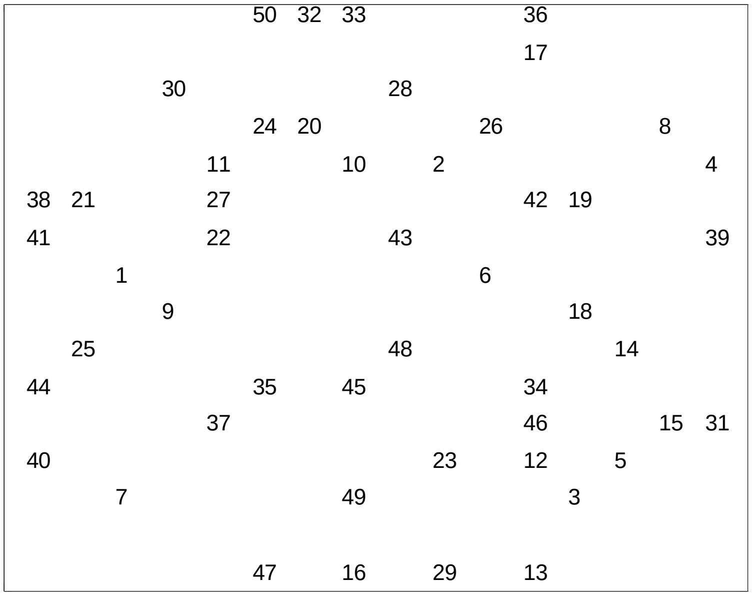


Figure 5.10 Plot for problem 1 from problem set 7.

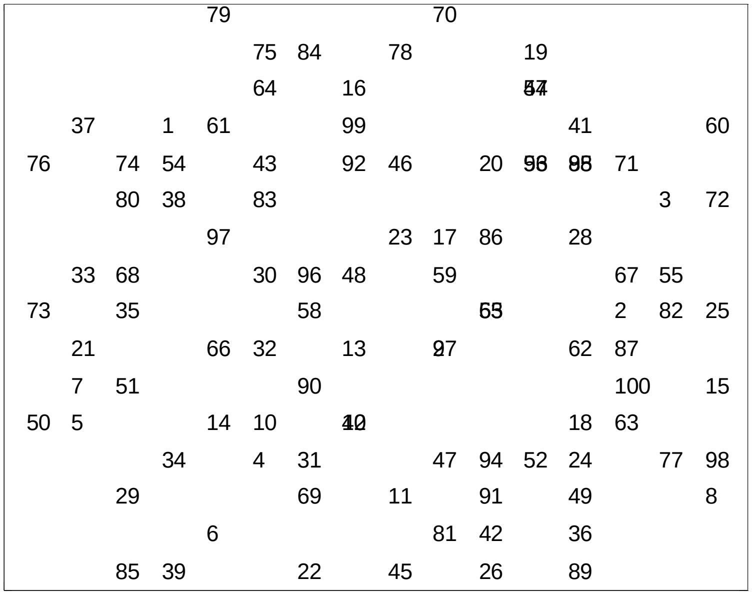


Figure 5.11 Plot for problem 8 from problem set 8.

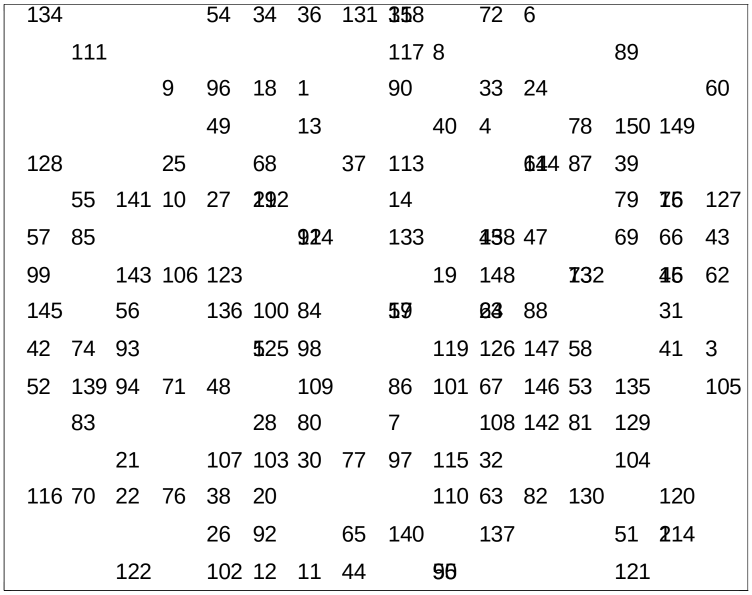


Figure 5.12 Plot for problem 10 from problem set 9.

In Table 5.10, we show the results for problem sets 7, 8, and 9, that is, for data sets originally in five-dimensions. In all 30 problems, IRNS produces results that are worse the results produced by DACN. The objective function values produced by IRNS are greater than those produced by DACN by an average of 6.92%, 2.31%, and 2.98%, for the 50-point problems, 100-point problems, and 150-point problems, respectively.

In Figures 5.10, 5.11, and 5.12, we show the plots of the final results for IRNS for problem one of problem set 7, problem eight of problem set 8, and problem 10 of problem set 9. Again the plots produced by IRNS are different from those produced by DACN (see Figures 4.10, 4.11, and 4.12)

The average running times are 218.04 seconds, 625.76 seconds, and 866.43 seconds for the 50-point problems, 100-point problems, and 150-point problems, respectively. DACN again produces better objective function values than IRNS in much faster times.

5.5 Improvement Heuristic

We observed that DACN almost always generates good final solutions. We use the final solution from DACN as the starting solution in an integer program heuristic. We randomly free five points at a time, as we did after Step 4 in IRNS, and reassign these points keeping the remaining m – 5 points fixed at their current locations. We then randomly select another set of five points and repeat this until all m points have been reassigned. We repeat this process until we have a cycle of all m points being reassigned with no change in the objective function value. This is referred to as the improvement heuristic (IMP).

	50-point problems		100-point problems		150-point problems	
Problems	Best solution obtained	Running time (seconds)	Best solution obtained	Running time (seconds)	Best solution obtained	Running time (seconds)
1	20982.00	16.39	101459.02	24.29	247552.56	39.79
2	16250.96	11.53	105203.22	23.77	265991.16	38.99
3	25871.46	11.43	126796.13	23.78	296864.55	39.07
4	22649.58	11.28	97143.60	23.80	246756.14	38.89
5	27901.62	23.17	133668.46	23.97	306127.24	39.19
6	22609.51	11.82	101322.81	23.38	244770.86	38.33
7	14946.64	11.92	82331.57	24.38	263992.07	39.48
8	16103.25	11.57	84547.23	23.69	231010.09	39.54
9	21778.13	11.76	115650.96	24.11	299858.64	39.02
10	26752.19	12.02	90984.91	23.42	261189.53	39.20
Average		13.29		23.86		39.15

Table 5.11 Results for problem sets 1, 2, and 3 for IMP. These problem sets are originally in three dimensions.

	50-point problems		100-point problems		150-point problems	
Problems	Best solution obtained	Running time (seconds)	Best solution obtained	Running time (seconds)	Best solution obtained	Running time (seconds)
1	44880.14	11.41	222313.12	24.13	524053.15	39.71
2	41767.38	11.47	212677.13	23.98	555620.19	40.71
3	50730.55	11.42	245369.56	24.57	572831.62	39.53
4	44550.63	11.62	213891.84	24.95	505089.33	40.25
5	48698.86	11.28	227596.04	24.33	584245.47	40.09
6	51246.53	12.56	226109.03	23.72	510667.94	40.45
7	32925.57	11.48	168891.25	24.70	483935.61	40.46
8	45511.42	11.33	179716.50	24.35	475716.73	40.59
9	50341.09	11.46	241308.77	24.16	572518.94	40.73
10	53649.65	11.41	229803.00	24.09	556992.94	39.70
Average		11.54		24.30		40.22

Table 5.12 Results for problem sets 4, 5, and 6 for IMP. These problem sets are originally in four dimensions.

	50-point problems		100-point problems		150-point problems	
Problems	Best solution obtained	Running time (seconds)	Best solution obtained	Running time (seconds)	Best solution obtained	Running time (seconds)
1	75242.43	22.48	325175.37	24.17	802908.30	39.62
2	64814.27	11.86	333587.21	24.73	841420.05	39.86
3	78566.40	11.82	348776.99	24.98	860219.43	40.17
4	75835.44	11.72	356028.47	24.39	812147.00	40.18
5	72817.28	11.35	329984.73	24.76	813605.93	40.33
6	81840.26	11.29	360490.54	24.26	856291.05	39.97
7	64342.71	11.63	297337.37	24.44	747778.81	40.65
8	71669.14	11.71	305608.28	24.48	753433.64	40.69
9	85219.88	11.55	368938.86	24.91	877125.17	40.66
10	78102.88	11.70	319881.60	24.31	784065.03	40.64
Average		12.71		24.54		40.23

Table 5.13 Results for problem sets 7, 8, and 9 for IMP. These problem sets are originally in five dimensions.

We applied IMP to problem sets 1 to 9. In Tables 5.11, 5.12, and 5.13, we show IMP's results. The running times reported for IMP are just those for the mathematical programming. They do not contain the running times to run DACN to generate the starting solutions. For most of the problems, IMP was not able to improve the solutions produced by DACN. IMP improved only two of the 120 problems. For Problem 5 of the 50-point problems in three dimensions, DACN produced an objective function value of 27914.8 in 4.49 seconds, while IMP produced an objective function value of 27901.62 in 23.17 seconds. For Problem 1 of the 50-point problems in five dimensions, the objective function values produced by DACN and IMP were 75247.8 and 75242.43, respectively. DACN's running time for this problem is 4.57 seconds and IMP's running time is 22.48 seconds. The two improvements obtained by IMP are very modest.

5.6 Conclusions

DACN produces much better objective function values than IRNS. The running times for DACN are also much shorter than those of IRNS. DACN is more accurate and efficient than IRNS.

DACN produces solutions by reassigning points, one point at a time. On the other hand, IMP reassigns five points at a time. We expected that IMP would be able to improve the results of DACN. However, that is not what we observed with our experiments. It appears that the local search procedure of assigning points one at a time works well for the data visualization problem.

Chapter 6: Genetic Algorithm

A technique that is commonly used to find near optimal solutions to QAP is the genetic algorithm. Since we obtained reasonable results with DACN, we now propose a hybrid heuristic that combines local search with a genetic algorithm. We refer to this heuristic as HGA.

HGA builds on DACN by applying genetic algorithms techniques to the final solution from DACN. For each generation, we produce m_c offspring by performing a crossover between the best solution and m_c randomly selected solutions from the t current solutions. We then randomly select m_m solutions from the current $t + m_c$ solutions and perform mutation on them to produce m_m offspring from mutation. After mutation we have $t + m_c + m_m$ solutions. We select the best t solutions and repeat the procedure until some stopping criterion is met.

Our crossover operator is illustrated below. Let B be the best solution and P be a randomly chosen solution for crossover. The offspring first inherits all assignments common to both B and P. Unassigned sites, that is, those with different assignments in B and P, are scanned from left to right. The offspring inherits the assignment with a better contribution to the objective function value. Let the assignment for point i in B and P be k_B and k_P, respectively. The contribution to the objective function value, that is, $\sum_{j \in M'} [od(i, j) - nd(k, l)]^2$, is calculated for i, for $k = k_B$ and $k = k_P$, where $j \in M'$ is the set of points with assignments l in the offspring. We assign point i to the lattice point k_B or k_P that gives the smallest contribution to the objective function.

We now consider the following example.

Let

B: 1 4 2 3 2

P: 1 4 4 3 1.

Since points 1, 2, and 4 are assigned to the same lattice points in both B and P, they have the same assignment in the offspring, that is,

O: 1 4 - 3 -.

To find the assignment to point 3, we find:

$c_B = [\mathrm{od}(3,1) – \mathrm{nd}(2,1)]^2 + [\mathrm{od}(3,2) – \mathrm{nd}(2,4)]^2 + [\mathrm{od}(3,4) – \mathrm{nd}(2,3)]^2$ and

$c_P = [\mathrm{od}(3,1) – \mathrm{nd}(4,1)]^2 + [\mathrm{od}(3,2) – \mathrm{nd}(4,4)]^2 + [\mathrm{od}(3,4) – \mathrm{nd}(4,3)]^2$.

If $c_B < c_P$, then point 3 is assigned to lattice point 2; otherwise it is assigned to lattice point 4. Let's assume $c_B < c_P$, and so point 3 is assigned to lattice point 2 and we have

O: 1 4 2 3 -.

To find the assignment for point 5, we find

$c_B = [\mathrm{od}(5,1) – \mathrm{nd}(2,1)]^2 + [\mathrm{od}(5,2) – \mathrm{nd}(2,4)]^2 + [\mathrm{od}(5,3) – \mathrm{nd}(2,2)]^2$

$+ [\mathrm{od}(5,4) – \mathrm{nd}(2,3)]^2$ and

$c_P = [\mathrm{od}(5,1) – \mathrm{nd}(1,1)]^2 + [\mathrm{od}(5,2) – \mathrm{nd}(1,4)]^2 + [\mathrm{od}(5,3) – \mathrm{nd}(1,4)]^2$

$+ [\mathrm{od}(5,4) – \mathrm{nd}(1,3)]^2$.

If $c_B < c_P$, then point 5 is assigned to lattice point 2; otherwise it is assigned to lattice point 1. Let's assume $c_P < c_B$, and so point 5 is assigned to lattice point 1 and we have

O: 1 4 2 3 1.

For mutation, we find the n_t lattice points to which the greatest number of points are assigned. We remove these n_t points from our set of lattice points and reassign the points in M to the remaining $n – n_t$ lattice points using one pass of local search. For example, consider a problem with $n = 4$ and $n_t = 1$. Let lattice point 3 be the lattice point to which the most points in M are assigned. Then we assign points to only lattice points 1, 2, and 4. We put back the n_t points and apply local search to reassign the points in M to all n lattice points. We repeat local search until there are $|M|$ consecutive iterations with no improvement in the objective function value.

6.1 Results and Analysis for GA

We implemented our HGA heuristic in the C++ programming language. We used Microsoft Visual C++ 6.0 and ran our experiments using Windows 2000 with an 800 MHz Pentium III processor and 512 MB RAM. From DACN, we have $t = 100$. We use $m_c = 20$, $m_m = 10$, $n_t = 8$, and stop after 10 generations.

In Table 6.1, we show the results for our HGA heuristic for the 50-point, 100-point and 150-point problems originally in two dimensions. In nine of 10 problems of size 50, DACN finds the global optimum (see Table 4.5). Recall, that these problems are originally in two dimensions and so the global optimum is known and is equal to zero. For these problems, HGA cannot improve on the best solution obtained by DACN as it is the global optimum. It can only increase on the frequency of the number of solutions out of 100 that converge to the best solution. The best solution found by DACN for problem 3 is 131.77. HGA is not able to improve this solution. In all 10 problems, HGA increases the frequency of solutions that converge to the best solution obtained. In all 10 problems, all 100 solutions for HGA converge to the best solution obtained.

	50-point problems			100-point problems			150-point problems		
Problem	Best solution	Freq	Running time (secs)	Best solution	Freq	Running time (secs)	Best solution	Freq	Running time (secs)
1	0.00	100	13.85	0.00	100	45.20	0.00	100	89.41
2	0.00	100	13.65	0.00	100	45.51	0.00	100	92.95
3	131.77	100	13.85	0.00	100	43.60	0.00	100	88.72
4	0.00	100	13.47	631.23	100	43.60	0.00	100	88.21
5	0.00	100	14.17	0.00	100	42.88	0.00	100	85.56
6	0.00	100	13.35	0.00	100	44.07	0.00	100	87.68
7	0.00	100	13.82	0.00	100	44.77	0.00	100	87.53
8	0.00	100	14.14	0.00	100	44.67	0.00	100	88.96
9	0.00	100	13.74	0.00	100	45.01	0.00	100	90.21
10	0.00	100	13.34	0.00	100	44.73	0.00	100	87.30
Average			13.74			44.40			88.65

Table 6.1 Results for problem sets originally in two dimensions for HGA.

In nine of the 10 problems of size 100, DACN finds the global optimum (see Table 4.5). The best solution found by DACN for problem 4 is 631.23. All 100 solutions generated by DACN converge to this solution. In Table 6.1, we see that HGA finds the same best solution for this problem. It is not able to improve on the solution produced by DACN. For the remaining nine problems, all 100 solutions produced by HGA converge to the optimal solution.

For all 10 problems of size 150, DACN finds the global optimum (see Table 4.5). In all 10 problems, HGA increases the frequency of solutions that converge to the best solution. All 100 solutions in all 10 problems converge to the best solution obtained.

The average running times for HGA for the 50-point problems, the 100-point problems, and the 150-point problems, originally in two dimensions, are 13.74 seconds, 44.40 seconds, and 88.65 seconds, respectively. The running times reported for HGA include both the running times for DACN and those for the genetic algorithm techniques. For DACN, the average running times for the 50-point problems, the 100-point problems, and the 150-point problems are 4.07 seconds, 13.24 seconds, and 27.95 seconds, respectively (see Table 4.5). The average running times for DACN are much shorter than those for HGA. However, the frequencies for HGA are much larger than those for DACN in 29 of the 30 problems in two dimensions.

We also applied HGA to problem sets 1 to 9 in Table 4.6. In Tables 6.2, 6.3, and 6.4, we show the results for problem sets 1, 2, and 3, respectively, that is, the data sets originally in three dimensions. In Table 6.2, we see that in six of the 10 problems of size 50, HGA produces better solutions than DACN. In the remaining four problems, DACN and HGA produce the same best solution; however, more solutions converge to the best solution with HGA. On average, HGA produces solutions that are 0.065 % less than those produced by DACN for the 50-point problems. In Table 6.3, we see that in five of the 10 problems of size 100, HGA produces better solutions than DACN. In the remaining five problems, GA does not improve the objective function value produced by DACN, but it improves the frequencies. On the average, HGA produces solutions that are 0.0717 % less than those produced by DACN for the 100-point problems. In Table 6.4, we see that HGA produces the same result for five of the 10 problems of size 150. In four of these problems, HGA produces better frequencies than DACN. However for problem 1, HGA and DACN produce the same frequency. In the remaining five problems, HGA finds a better solution than DACN. For the 150-point problems, HGA produces solutions that are on the average 0.0509 % less than those produced by DACN.

Problem	Best solution	Frequency	Running time (seconds)	Percent decrease over DACN results
1	20982.0	9	15.23	0.000000
2	16217.7	7	16.66	0.204910
3	25864.3	1	15.87	0.027830
4	22645.7	6	16.31	0.017219
5	27901.6	1	16.60	0.047287
6	22541.3	2	16.45	0.301643
7	14946.6	8	15.59	0.000000
8	16095.6	3	15.67	0.047196
9	21778.1	43	16.43	0.000000
10	26752.2	85	16.36	0.000000
Average			16.12	0.064609

Table 6.2 Results for problem set 1 for HGA. These are the 50-point problems originally in three dimensions.

Problem	Best solution	Frequency	Running time (seconds)	Percent decrease over DACN results
1	101459	13	57.74	0.000000
2	105203	41	56.69	0.000000
3	126677	1	57.00	0.093852
4	97109.3	13	53.49	0.035086
5	133668	42	55.54	0.000000
6	101311	50	55.39	0.011843
7	82283.7	1	54.49	0.058179
8	84547.2	93	56.73	0.000000
9	115651	8	58.77	0.000000
10	90513.7	9	56.71	0.517888
Average			56.26	0.071707

Table 6.3 Results for problem set 2 for HGA. These are the 100-point problems originally in three dimensions.

Problem	Best solution	Frequency	Running time (seconds)	Percent decrease over DACN results
1	247553	1	117.11	0.000000
2	265991	3	119.93	0.000000
3	296768	1	114.36	0.032675
4	246756	7	112.22	0.000000
5	306124	18	121.17	0.000980
6	243635	2	112.33	0.464107
7	263978	2	116.37	0.005303
8	231010	7	132.62	0.000000
9	299859	25	121.64	0.000000
10	261174	3	117.00	0.006126
Average			118.46	0.050919

Table 6.4 Results for problem set 3 for HGA. These are the 150-point problems originally in three dimensions.

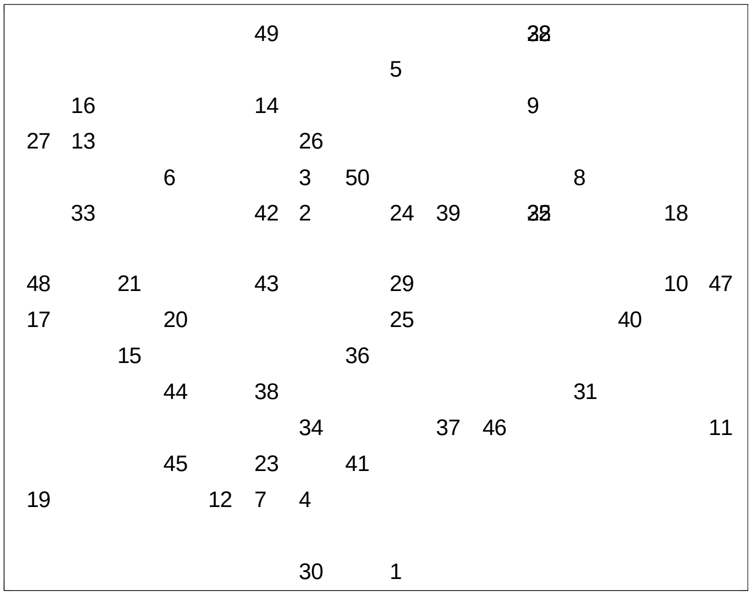


Figure 6.1 Plot for problem 2 from Problem Set 1.

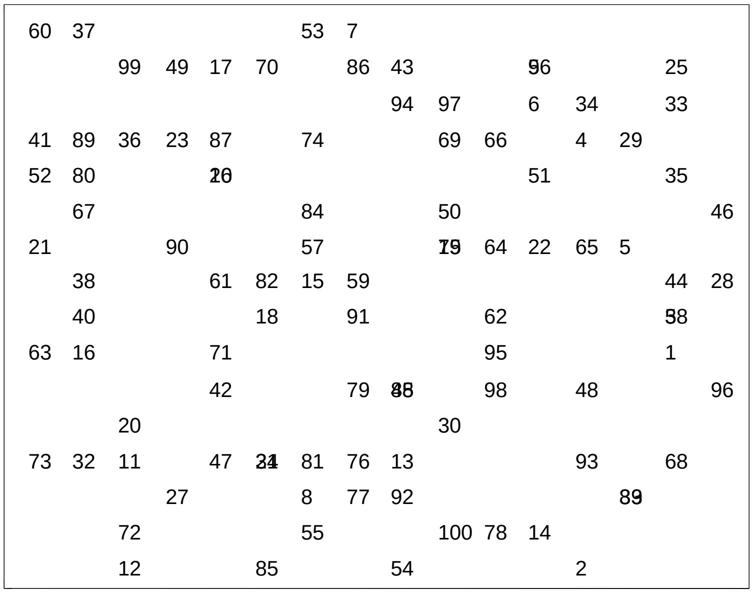


Figure 6.2 Plot for problem 9 from problem set 12.

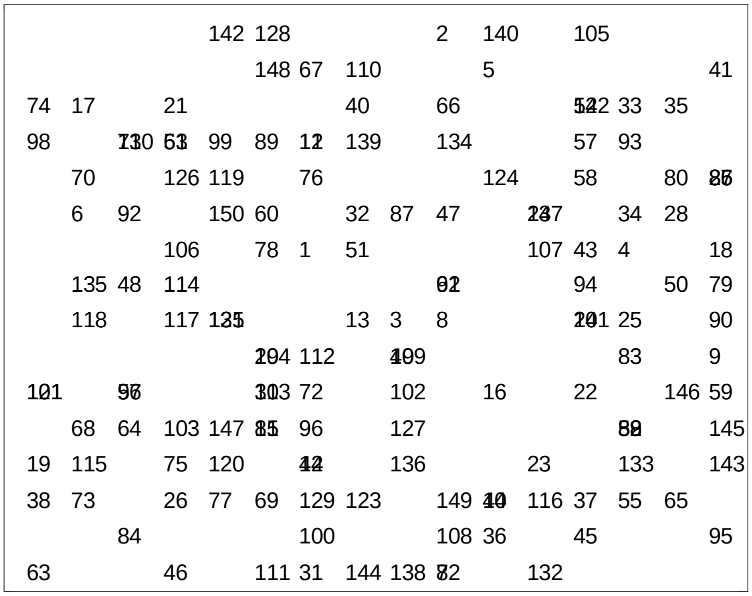


Figure 6.3 Plot for Problem Set 3: Problem 4.

In Figure 6.1, we show the plot of the final results for HGA for problem two of problem set 1. The plot of this problem for DACN is shown in Figure 4.4. The plots are almost the same as is to be expect since the objective function values are almost the same. It should be noted that Figure 6.1 is a reflection of Figure 4.4 in the *x*-axis. In Figures 6.2 and 6.3, we show the plots for problem nine of problem set 2 and problem four of problem set 3, respectively. The plots for these problems are the same as those produced by DACN (see Figures 4.5 and 4.6) as HGA and DACN produce the same objective function values for these problems. Note that Figure 6.3 is the reflection of Figure 4.6 in the line $y = x$.

The average running times for HGA for the 50-point problems, the 100-point problems, and the 150-point problems, originally in three dimensions, are 16.12 seconds, 56.26 seconds, and 118.46 seconds, respectively. For DACN, the average running times for the 50-point problems, the 100-point problems, and the 150-point problems are 4.33 seconds, 15.73 seconds, and 36.65 seconds, respectively (see Table 4.7). The average running times for DACN are much shorter than those for HGA. HGA finds better solutions than DACN in 16 of the 30 problems originally in three dimensions. In 14 of the remaining 15 problems, HGA produces larger frequencies than DACN.

In Tables 6.5, 6.6, and 6.7, we show the results for problem sets 4, 5, and 6, respectively. These are the problem sets originally in four dimensions. In Table 6.5, for the 50-point problems, HGA gives better solutions than DACN in nine of the 10 problems. For problem 9, both HGA and DACN find a best solution of 50341.1. However, DACN has a frequency of one while HGA has a frequency of seven. On the average HGA produces solutions that are 0.070 % less than those produced by DACN, for the 50-point problems. In Table 6.6, in eight of the 100-point problems, HGA gives a better solution than DACN. For the remaining two problems HGA has a greater frequency than DACN. On average HGA produces solutions that are 0.0363 % less than those produced by DACN for the 100-point problems. In Table 6.7, in eight of the 150-point problems, HGA gives a better result than DACN. For the other two problems, HGA gives a greater frequency than DACN in problem 10 and the same frequency as DACN in problem 2. On average HGA produces solutions that are 0.0349 % less than those produced by DACN for the 150-point problems.

In Figures 6.4, 6.5, and 6.6, we show the plots of the final solutions for HGA for problem six of problem set 4, problem seven of problem set five, and problem five of problem set six, respectively. These plots are similar to those produced by DACN for the same problems as is to be expected (see Figures 4.7, 4.8, and 4.9) since the objective function values produced by HGA and DACN are almost the same.

The average running times for HGA for the 50-point, 100-point, and 150-point problems are 15.73 seconds, 53.35 seconds, and 111.54 seconds, respectively. These running times are much longer than those for DACN, which are 4.99 seconds, 18.69 seconds, and 44.29 seconds, for the 50-point, 100-point, and 150-point problems, respectively. For 25 of the 30 problems originally in four dimensions, HGA gives better results than DACN. HGA gives a better frequency than DACN in four of the remaining problems.

Problem	Best solution	Frequency	Running time (seconds)	Percent decrease over DACN results
1	44846.2	10	15.31	0.075535
2	41746.7	2	15.77	0.049560
3	50692.6	1	16.11	0.074905
4	44531.1	6	15.87	0.043770
5	48680.0	30	15.80	0.038810
6	51229.6	1	16.00	0.032978
7	32925.6	36	15.70	0.084968
8	45379.2	5	15.48	0.290477
9	50341.1	7	15.68	0.000000
10	53644.6	2	15.53	0.009506
Average			15.73	0.070051

Table 6.5 Results for problem set 4 for HGA. These are the 50-point problems originally in four dimensions.

Problem	Best solution	Frequency	Running time (seconds)	Percent decrease over DACN results
1	222313	11	52.97	0.000000
2	212677	24	52.04	0.000000
3	245229	1	54.82	0.057464
4	213861	1	51.63	0.014493
5	227173	5	53.63	0.185856
6	226018	1	54.27	0.040246
7	168886	1	52.29	0.002960
8	179705	3	54.94	0.006121
9	241251	1	53.77	0.024036
10	229729	4	53.17	0.032201
Average			53.35	0.036338

Table 6.6 Results for problem set 5 for HGA. These are the 100-point problems originally in four dimensions.

Problem	Best solution	Frequency	Running time (seconds)	Percent decrease over DACN results
1	523987	1	115.01	0.012594
2	555620	1	111.25	0.000000
3	572456	2	111.87	0.065639
4	504728	1	109.16	0.071473
5	584229	2	109.91	0.027386
6	510445	1	119.79	0.043668
7	483908	1	106.48	0.005786
8	475337	4	106.33	0.079879
9	572133	1	116.17	0.067421
10	556993	7	109.43	0.000000
Average			111.54	0.034920

Table 6.7 Results for problem set 6 for HGA. These are the 150-point problems originally in four dimensions.

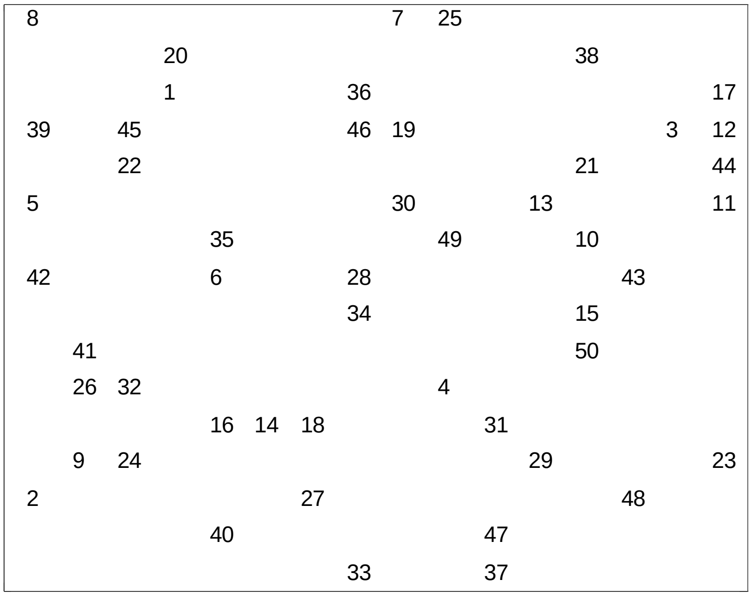


Figure 6.4 Plot for Problem Set 4: Problem 6.

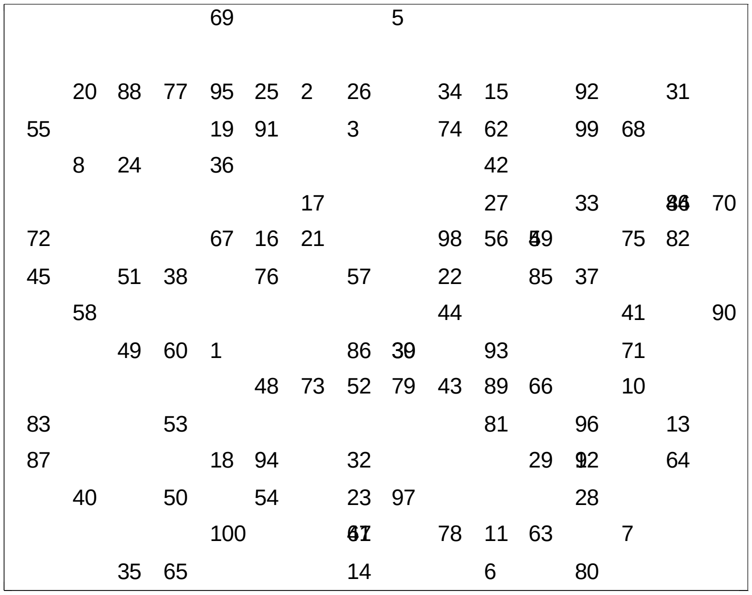


Figure 6.5 Plot for problem 7 from problem set 5.

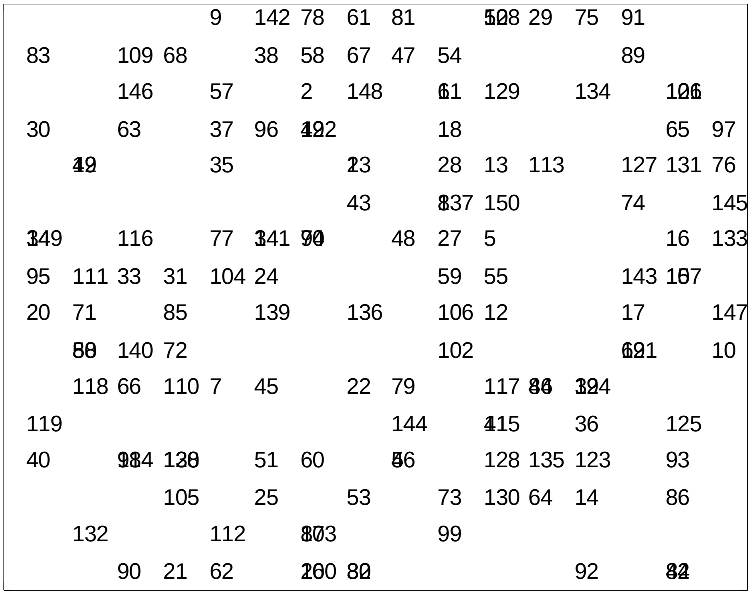


Figure 6.6 Plot for problem 5 from problem set 6.

In Tables 6.8, 6.9, and 6.10, we give the results for the problem sets originally in five dimensions, that is, problem sets 7, 8, and 9, respectively. In Table 6.8, in seven of the 50-point problems, HGA gives a better solution than DACN. For the other three problems, HGA gives a greater frequency than DACN. On average, HGA produces solutions that are 0.071 % less than those produced by DACN for the 50-point problems. In Table 6.9, for the 100-point problems, HGA gives better results than DACN in all 10 problems. On average HGA, produces solutions that are 0.0463 % less than those produced by DACN, for the 100-point problems. In Table 6.10, in five of the 150-point problems, HGA gives better results than DACN. In three of the remaining five problems, HGA gives greater frequencies than DACN. On average HGA, produces solutions that are 0.0281 % less than those produced by DACN for the 150-point problems.

In Figures 6.7, 6.8, and 6.9, we show the plots of the final solutions for HGA for problem one of problem set 7, problem eight of problem set eight, and problem 10 of problem set nine, respectively. Figures 6.7 and 6.8 are similar to those produced by DACN for the same problems as is to be expected (see Figures 4.10 and 4.11) since the objective function values produced by HGA and DACN are almost the same. The plots for DACN and HGA are the same as they produce the same objective function value (see Figure 4.12).

Problem	Best solution	Frequency	Running time (seconds)	Percent decrease over DACN results
1	75011.8	1	16.99	0.313630
2	64802.5	6	17.90	0.018206
3	78469.3	1	16.99	0.123590
4	75825.7	4	17.53	0.012628
5	72742.6	2	16.66	0.102586
6	81785.2	1	17.11	0.067326
7	64342.7	7	16.90	0.000000
8	71669.1	21	16.93	0.000000
9	85162.5	3	17.04	0.067355
10	78102.9	6	17.59	0.000000
Average			17.16	0.070532

Table 6.8 Results for problem set 7 for HGA. These are the 50-point problems originally in five dimensions.

Problem	Best solution	Frequency	Running time (seconds)	Percent decrease over DACN results
1	325164	3	60.20	0.003383
2	333538	1	60.45	0.014689
3	348745	4	59.66	0.009175
4	355959	3	59.33	0.019380
5	329939	17	56.98	0.013940
6	360454	2	61.83	0.012638
7	296948	3	58.91	0.130828
8	305507	2	59.36	0.033049
9	368775	1	58.45	0.044451
10	319291	1	60.84	0.184756
Average			59.60	0.046392

Table 6.9 Results for problem set 8 for HGA. These are the 100-point problems originally in five dimensions.

Problem	Best solution	Frequency	Running time (seconds)	Percent decrease over DACN results
1	802906	9	120.34	0.000249
2	841420	6	124.39	0.000000
3	860219	3	126.89	0.000000
4	811990	2	124.65	0.019331
5	812123	1	126.27	0.182275
6	856291	16	124.78	0.000000
7	747410	1	121.03	0.049346
8	753376	1	126.73	0.007698
9	877125	1	124.87	0.000000
10	784065	1	126.13	0.000000
Average			124.61	0.025890

Table 6.10 Results for problem set 9 for HGA. These are the 150-point problems originally in five dimensions.

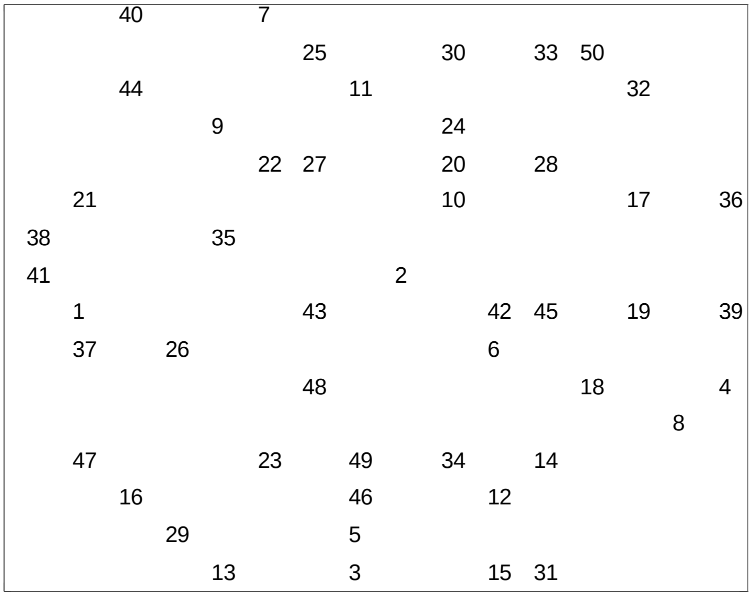


Figure 6.7 Plot for problem 1 from problem set 7.

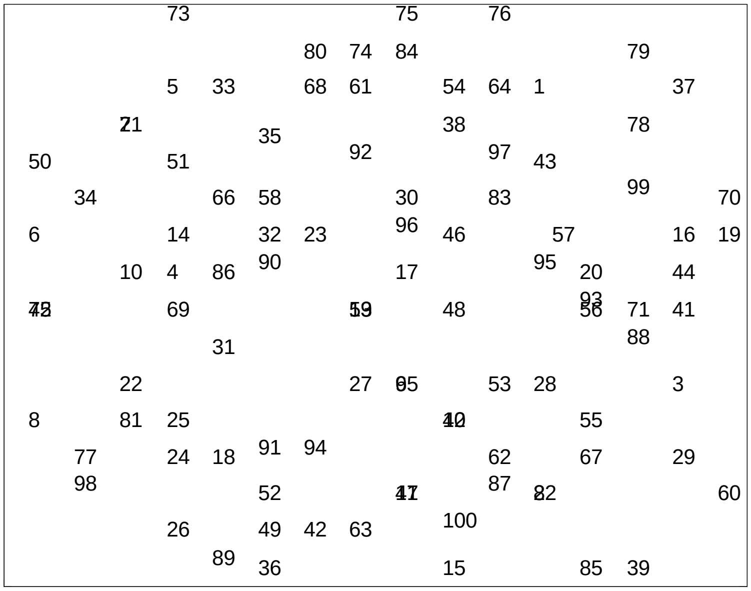


Figure 6.8 Plot for problem 8 from problem set 8.

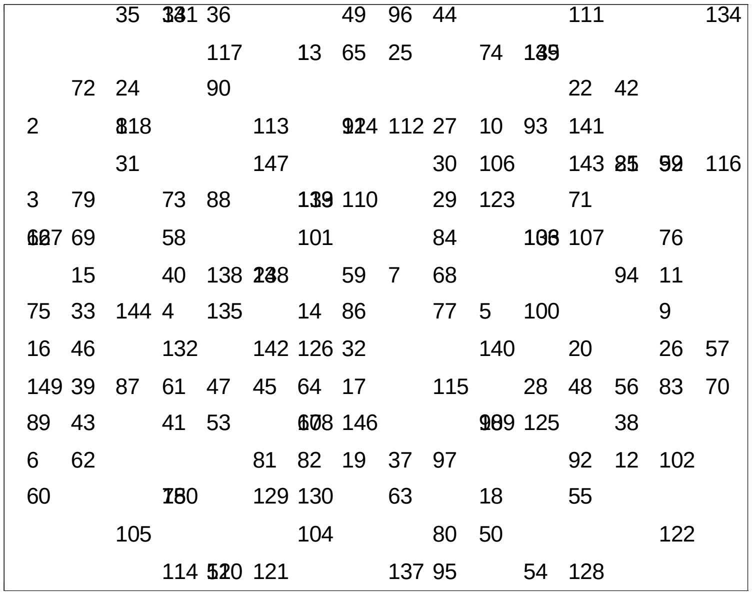


Figure 6.9 Plot for problem 10 from problem set 9.

The average running times for HGA are 17.18 seconds, 59.60 seconds, and 124.61 seconds, for the 50-point, 100-point, and 150-point problems. These running times are longer than those for DACN. The average running times for DACN for the 50-point, 100-point, and 150-point problems are 4.66 seconds, 17.41 seconds, and 40.27seconds, respectively. HGA gives better results in 22 of the 30 problems in five dimensions. In six of the remaining eight problems, HGA gives a larger frequency.

6.2 Conclusions

HGA can be used to improve the solutions obtained by DACN. However, the improvements are very marginal. In addition, the running times of HGA are longer than those of DACN. When our heuristic is used as a stand-alone approach or a very accurate solution is required, HGA can be used rather than DACN. On the other hand, if our heuristic is used to produce a starting solution for a nonlinear method or an approximate solution is required, then DACN should be used rather than HGA for faster computation times.

Chapter 7: Comparing Discrete Local Search to a Nonlinear Optimization Technique – Sammon map

In this chapter, we compare the results of DACN to the results generated by a nonlinear Sammon map (NLSM). The Sammon map objective function is different from (but similar to) the least squares scaling objective function that we have used in our experiments so far. However, as stated earlier, we can easily apply our methods to this objective function. Therefore, we use the Sammon map objective function in our next set of experiments. This shows how easily we can change the objective function in our technique.

Our motivation for comparing DACN to NLSM is threefold. First, since neither approach generates solutions that are guaranteed to be globally optimal, a simple comparison is of interest. Second, we seek to ascertain the quality of the discrete optimization approximation for this continuous optimization problem. Third, we investigate whether the two techniques (DACN and NLSM) can be combined in order to obtain superior results.

7.1 Sammon map

The Sammon map procedure is an algorithm used to analyze multivariate data. The algorithm is a point mapping from a high dimensional space to a lower dimensional space such that the inherent data structure is preserved approximately (Sammon, 1969). The Sammon map procedure randomly assigns starting coordinates to the points in M. A steepest descent procedure is then used to reassign the points in order to reduce the objective function value. The procedure is repeated until a stopping criterion is met. We point out that the nonlinear (steepest descent) Sammon map procedure does not guarantee a global minimum. The Sammon map objective function is:

$$\text{minimize } \frac{1}{\sum_{i\in M}\sum_{\substack{j\in M\\ j>i}} od(i,j)} \sum_{i\in M}\sum_{\substack{j\in M\\ j>i}}\sum_{k\in N}\sum_{l\in N} \frac{[od(i,j)-nd(k,l)]^2 x_{ik}x_{jl}}{od(i,j)}.$$

We use the Sammon map procedure of Condon, Golden, and Wasil (2003), as coded in Mathematica. The stopping criterion is either 100 iterations or when the sum of the differences in the objective function values of 10 successive iterations is less than 1% of the current objective function value, whichever occurs first.

7.2 Combined Heuristics

In NLSM, points can be assigned anywhere in *q*-space. However, with DACN, points can be assigned only to the lattice points. Therefore, NLSM is likely to generate better results than DACN. However, the speed of convergence of nonlinear optimization techniques is dependent upon the starting solution. In earlier chapters, we observed that DACN generates good approximate final solutions. This suggests the following heuristic (COMB). Instead of starting NLSM with a random solution, we can use the final solution from DACN as the starting solution.

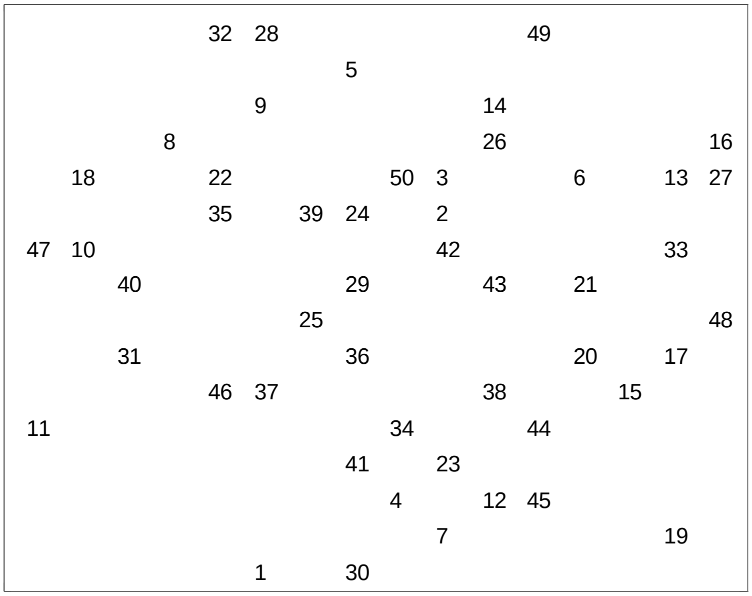
7.3 Results

We apply DACN with the Sammon map objective function, NLSM and COMB, to the 50-point and 100-point data sets (problem sets 1, 2, 4, 5, 7, and 8 from Table 4.6). We do not apply NLSM to the 150-point data sets as the problem size becomes too large to run in Mathematica.

In Table 7.1, we give the results for 50-point problems generated from a lattice in three dimensions (that is, problem set 1). In problems one, three, and five, DACN gives better objective function values than NLSM. For the remaining seven problems, NLSM produces better objective function values than DACN. However, DACN's results are

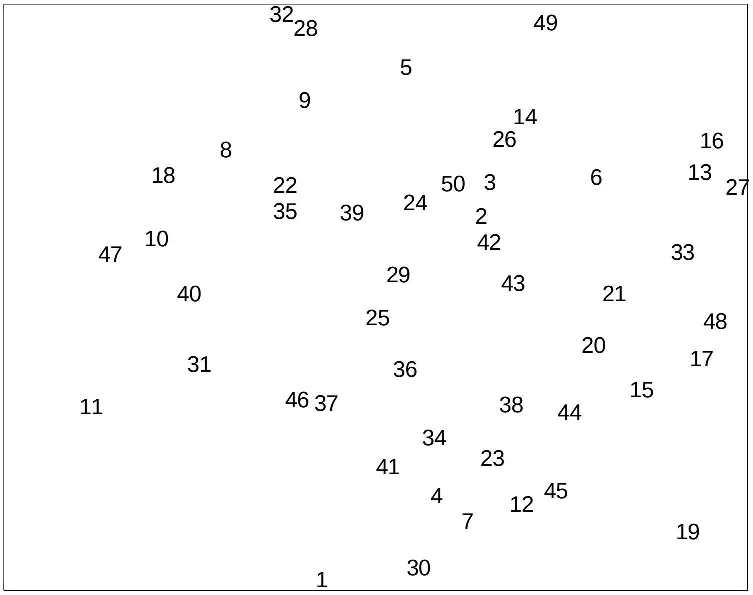
	DACN		NLSM		COMB	
Problem	Best solution	Running time (seconds)	Best solution	Running time (seconds)	Best solution	Running time (seconds)
1	0.04248	8.20	0.04261	95.08	0.04092	9.76
2	0.03640	9.13	0.03600	74.13	0.03416	12.856
3	0.04841	8.92	0.05220	78.84	0.04691	9.80
4	0.04365	8.49	0.04004	108.39	0.03994	11.32
5	0.05376	7.71	0.05500	68.02	0.05175	11.23
6	0.03665	8.48	0.03335	56.98	0.03334	14.35
7	0.03473	8.18	0.03341	103.20	0.03286	11.33
8	0.03397	8.38	0.03237	74.30	0.03214	12.88
9	0.04010	8.24	0.03842	84.05	0.03808	11.38
10	0.04774	8.41	0.04578	88.44	0.04473	14.60
Average		8.41		83.14		11.95

Table 7.1 Results for Problem Set 1: 50-point problems originally in three dimensions.

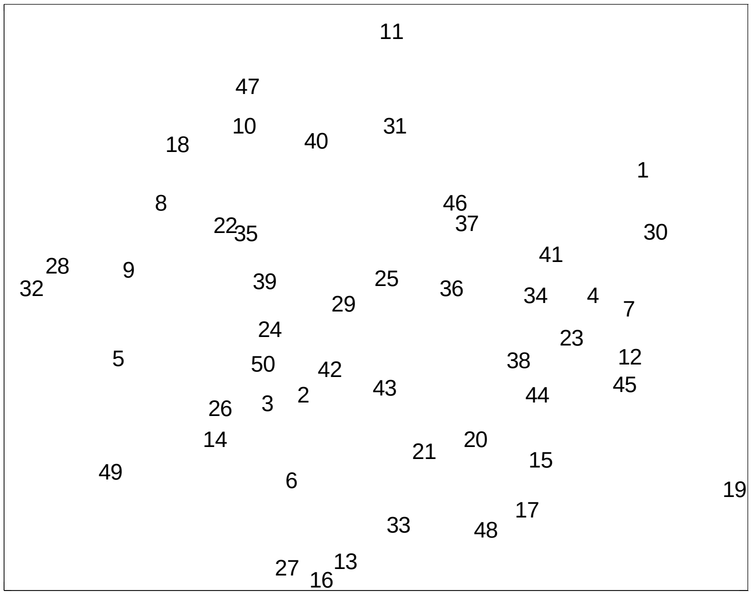


(a) DACN

Figure 7.1 Plots for problem 2 from problem set 1.

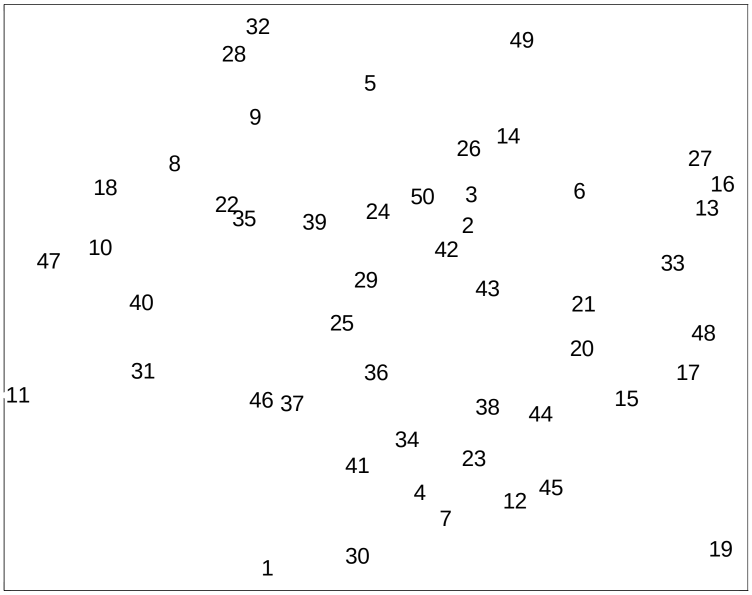


(b) COMB



(c) NLSM

Figure 7.1 (continued).



(d) Translated NLSM
Figure 7.1 (continued).

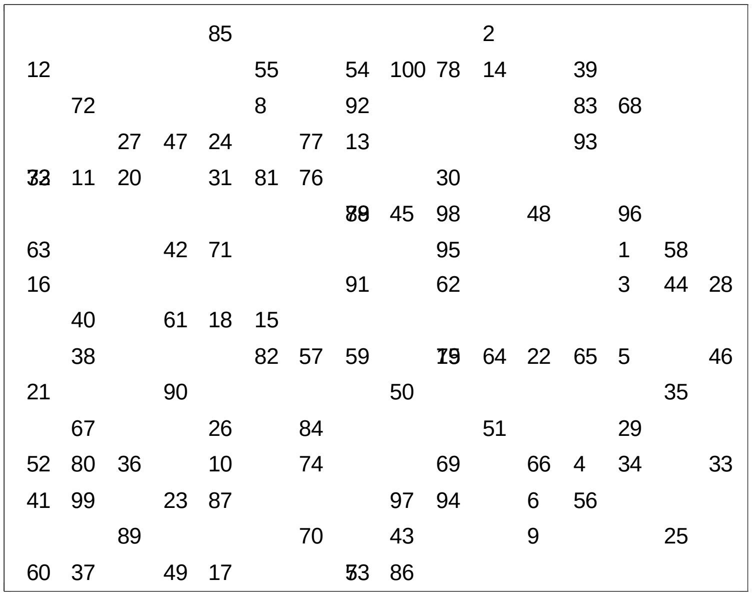
very close to those for NLSM. COMB gives the best solution to each of the 10 problems. In Figure 7.1, we show plots of the final results generated by DACN, NLSM, and COMB for Problem 2. We also show the plot of a linear transformation of the coordinates produced by NLSM in Figure 7.1(d). The objective function values indicate that the three figures for DACN, COMB, and NLSM, should be similar. Figures 7.1(a) and 7.1(b) support this. Figure 7.1(c) appears to be different. This is due to the arbitrary orientation of the problem. We, therefore, used a Procrustes rotation in Matlab to translate the plot of NLSM, obtaining Figure 7.1(d). DACN requires that the original points be assigned to lattice points. COMB and NLSM have no such restriction. Figure 7.1(d) is similar to Figures 7.1(a) and 7.1(d).

The average running times for DACN, NLSM, and COMB are 8.41 seconds, 83.14 seconds, and 11.95 seconds, respectively. The running times we report for COMB are only those for the nonlinear code and do not include the running times for the divide-and-conquer heuristic. The average running time for COMB is much smaller than that for NLSM. We do not compare the running times for DACN and NLSM since the codes are written in different programming languages.

In Table 7.2, we give the results for Problem Set 2. NLSM produces better objective function values than DACN and COMB produces better objective function values than NLSM for all 10 problems. The average running times are 36.67 seconds, 454.38 seconds, and 76.68 seconds for DACN, NLSM, and COMB, respectively. The results are similar to those observed with the first problem set: COMB does better than NLSM, which does better than DACN. On all 10 problems, COMB has much lower running times than NLSM. In Figure 7.2, we show plots of the final results generated by DACN, NLSM, and COMB for Problem 9. We also show the plot of a linear transformation of the coordinates produced by NLSM in Figure 7.2(d). Figures 7.2(a) and 7.2(b) are very similar and support the fact that the objective function values produced by DACN and COMB are close to each other. Arbitrary orientation accounts for Figure 7.2(c) appearing different from Figures 7.2(a) and 7.2(b). Figure 7.2(d) is similar to Figures 7.2(a) and 7.2(b) as expected.

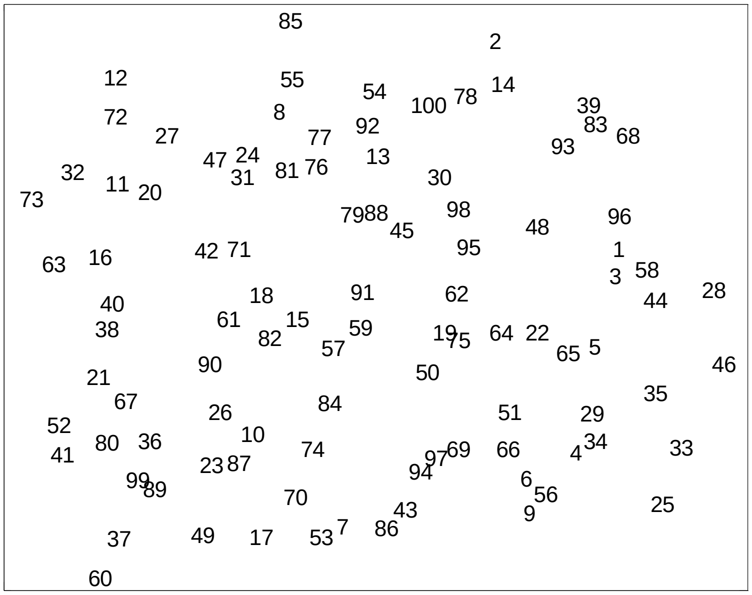
	DACN		NLSM		COMB	
Problem	Best solution	Running time (seconds)	Best solution	Running time (seconds)	Best solution	Running time (seconds)
1	0.05051	37.06	0.04975	784.63	0.04869	55.97
2	0.05086	38.46	0.04916	335.56	0.04894	48.03
3	0.05668	36.66	0.05454	412.27	0.05334	156.57
4	0.04590	33.47	0.04315	395.07	0.04292	62.98
5	0.05877	36.55	0.05839	421.33	0.05643	55.60
6	0.04440	36.13	0.04054	621.28	0.03971	78.90
7	0.04430	33.60	0.04316	564.12	0.04228	55.22
8	0.04297	39.38	0.04088	348.70	0.04065	63.06
9	0.05387	40.57	0.05291	315.36	0.05162	119.01
10	0.04339	34.86	0.04021	345.48	0.03967	71.48
Average		36.67		454.38		76.68

Table 7.2 Results for Problem Set 2: 100-point problems originally in three dimensions.

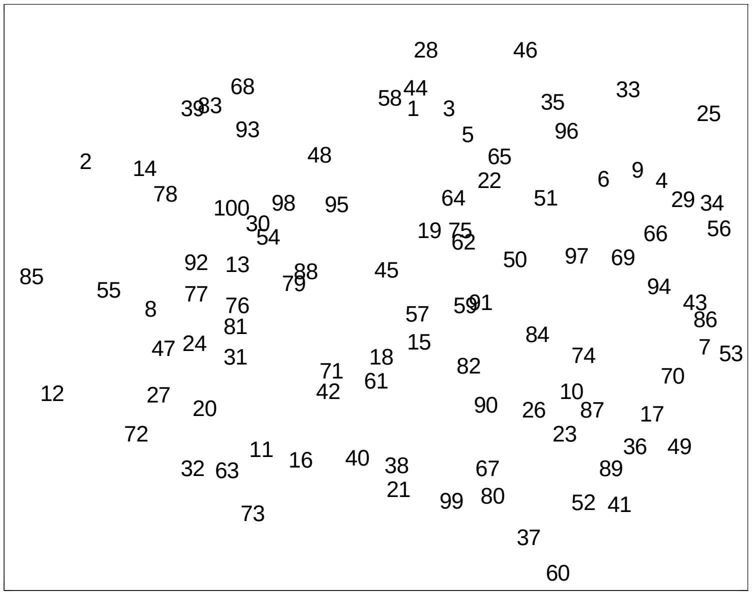


(a) DACN

Figure 7.2 Plots for problem 9 from Problem Set 2.

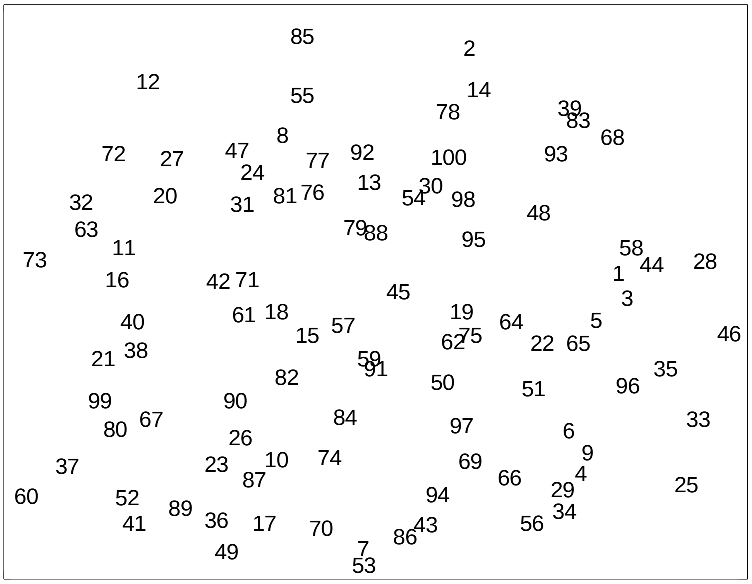


(b) COMB



(c) NLSM

Figure 7.2 (continued).



(d) Translated NLSM
Figure 7.2 (continued).

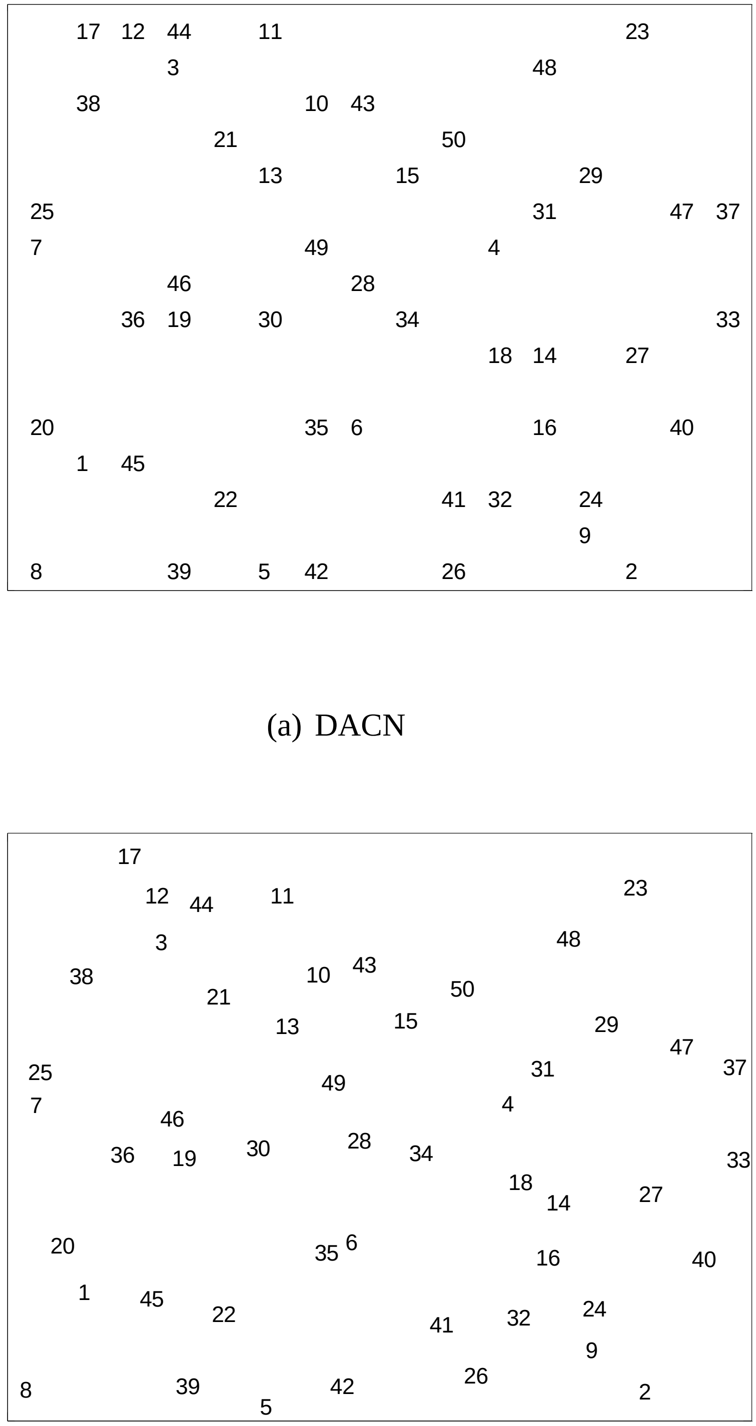
In Tables 7.3 and 7.4, we give the results for Problem Sets 4 and 5. For Problem Set 4, NLSM generates better objective function values than DACN in problems one, two, and seven. In the remaining seven problems, DACN produces better objective function values. For Problem Set 5, DACN produces better objective function values than NLSM except for problems two, six, and eight. For both Problem Sets 4 and 5, COMB generates the best objective function values. For all 20 problems in the two problem sets, COMB generates better results than DACN and NLSM. In Figures 7.3 we show plots of the final results generated by DACN, NLSM, and COMB for Problem 6 in Problem Set 4. In Figure 7.4, we show plots for Problem 7 in Problem Set 5. We also show the plots of a linear transformation of the coordinates produced by NLSM in Figures 7.3(d) and 7.4(d). Since the objective function values produced by the three methods are all close to each other, we expect the plots to be similar. Indeed, the plots produced by DACN and

	DACN		NLSM		COMB	
Problem	Best solution	Running time (seconds)	Best solution	Running time (seconds)	Best solution	Running time (seconds)
1	0.06155	7.71	0.06046	103.23	0.06009	9.81
2	0.06409	8.07	0.06306	102.49	0.06237	9.78
3	0.06891	7.53	0.07600	61.61	0.06750	9.71
4	0.06388	7.55	0.06830	91.58	0.06220	9.77
5	0.06986	7.74	0.07355	69.43	0.06813	9.78
6	0.06458	7.54	0.06863	104.54	0.06275	12.89
7	0.05363	7.30	0.05289	69.77	0.05222	9.82
8	0.06796	7.96	0.06853	68.05	0.06625	8.24
9	0.06454	7.67	0.06461	131.58	0.06296	8.07
10	0.07000	8.09	0.07786	75.81	0.06867	8.26
Average		7.712		87.81		9.61

Table 7.3 Results for Problem Set 4: 50-point problems originally in four dimensions.

	DACN		NLSM		COMB	
Problem	Best solution	Running time (seconds)	Best solution	Running time (seconds)	Best solution	Running time (seconds)
1	0.07539	32.43	0.07731	409.78	0.07388	40.67
2	0.07339	31.52	0.07313	498.60	0.07168	103.85
3	0.08081	32.50	0.08120	447.68	0.07855	47.26
4	0.07280	30.31	0.07368	708.25	0.07073	47.45
5	0.07475	30.81	0.07603	384.84	0.07227	56.06
6	0.07247	34.35	0.07024	482.25	0.07005	55.71
7	0.06424	29.29	0.07641	574.48	0.06262	48.33
8	0.06863	32.61	0.06824	732.20	0.06681	56.45
9	0.07900	32.79	0.08367	363.95	0.07698	47.64
10	0.07505	32.98	0.08129	322.34	0.07336	48.27
Average		31.96		492.44		55.17

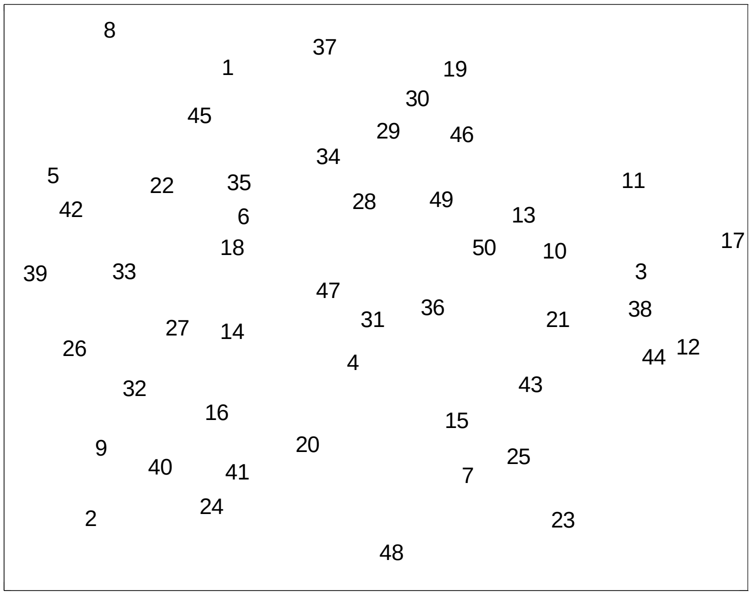
Table 7.4 Results for Problem Set 5: 100-point problems originally in four dimensions.



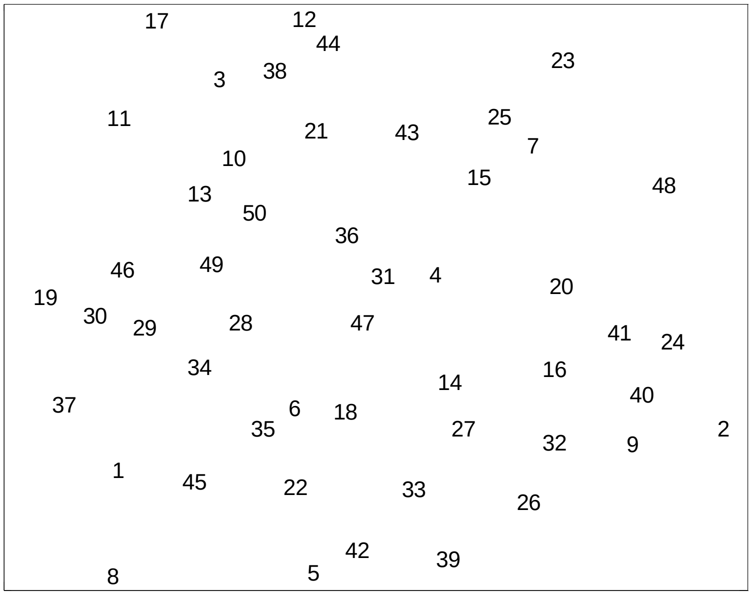
(a) DACN

(b) COMB

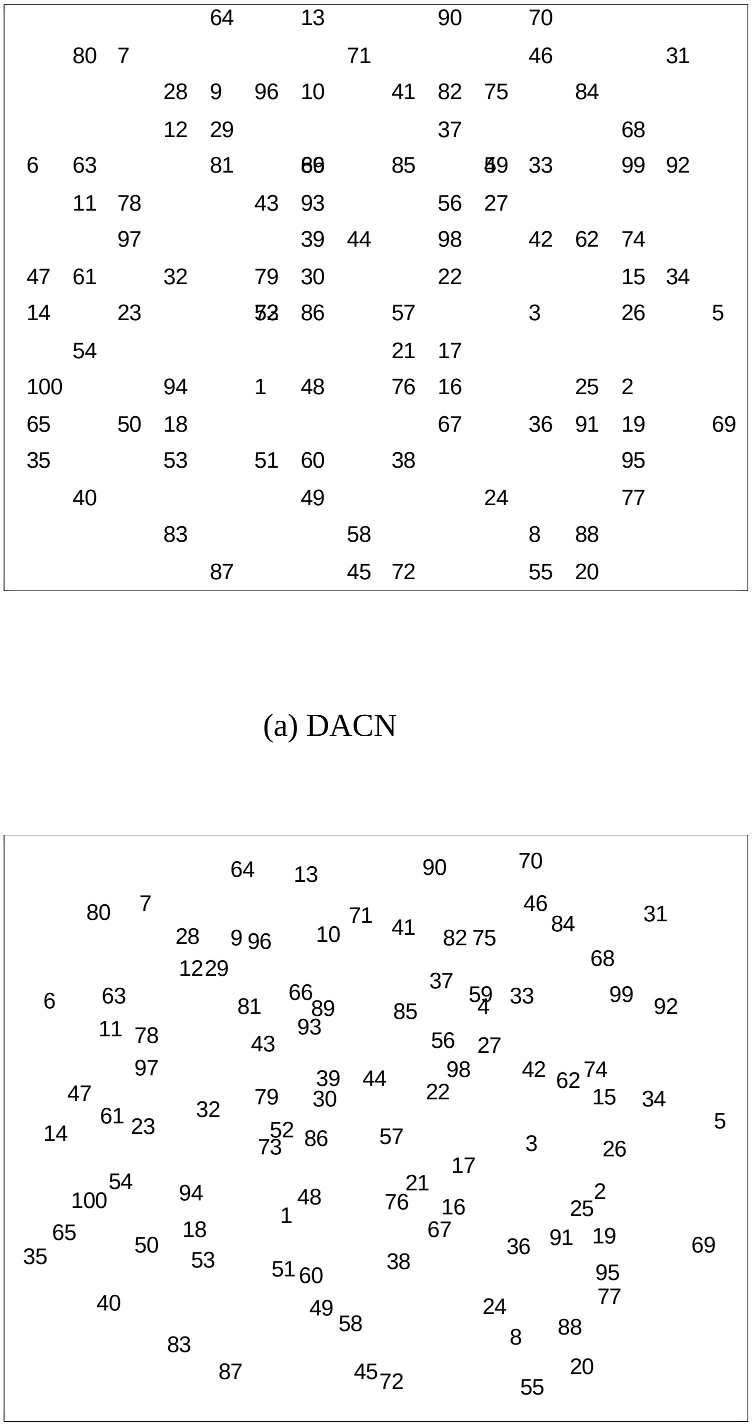
Figure 7.3 Plots for problem 6 from problem set 4.



(c) NLSM



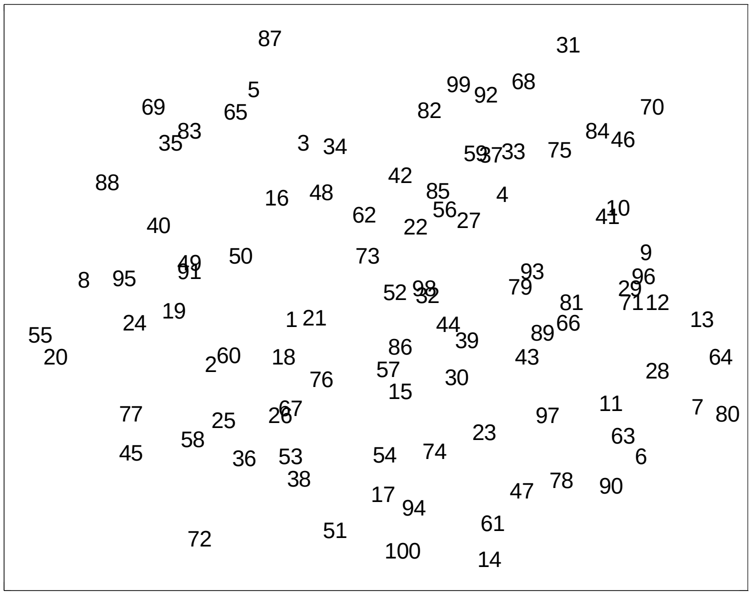
(d) Translated NLSM
Figure 7.3 (continued).



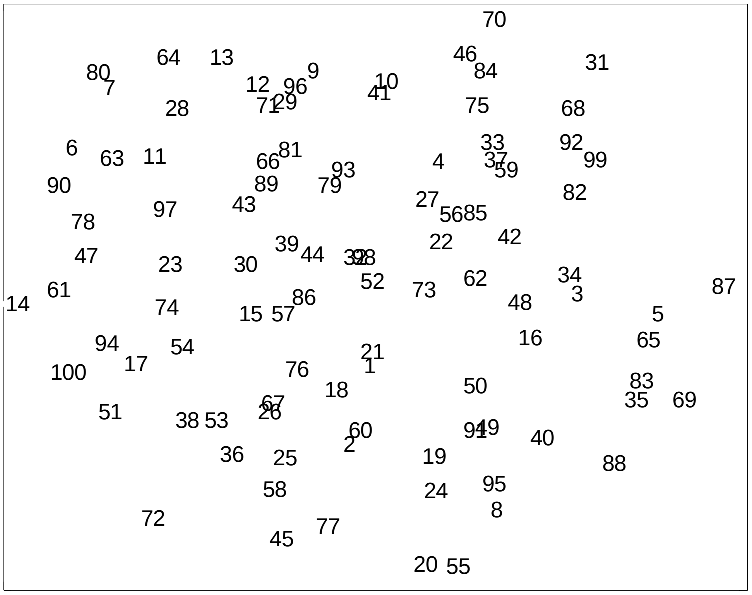
(a) DACN

(b) COMB

Figure 7.4 Plots for problem 7 from problem set 5.



(c) NLSM



(d) Translated NLSM
Figure 7.4 (continued).

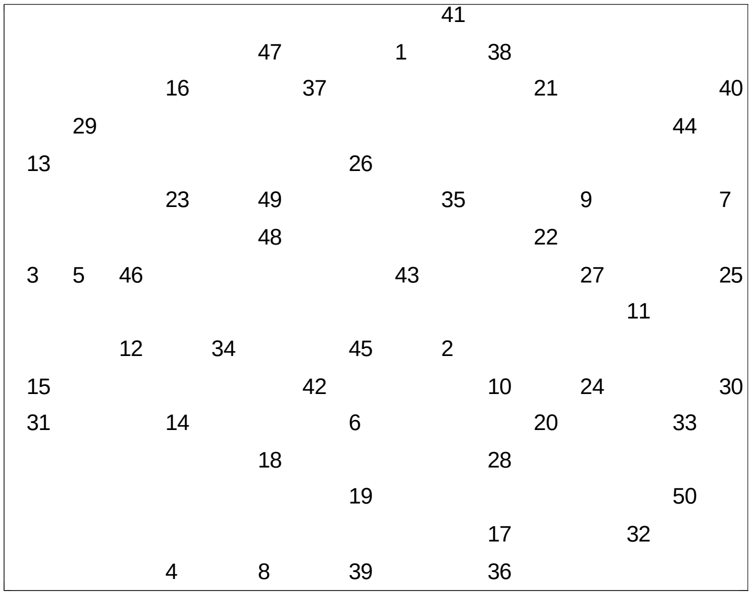
COMB is very similar. The plots produced by NLSM appear to be different. However, this difference is caused by the arbitrary orientation of the solutions produced by these methods. The translated NLSM plots are similar to the plots produced by DACN and COMB.

In Table 7.3, the average running times for DACN, NLSM, and COMB are 7.72 seconds, 87.81 seconds, and 9.61 seconds, respectively. In Table 7.4, the average running times are 31.96 seconds, 492.44 seconds, and 55.17 seconds for DACN, NLSM, and COMB, respectively. Once again, the running times for COMB are much smaller than those for NLSM.

In Table 7.5, we give results for Problem Set 7. In all 10 problems, DACN produces a better objective function value than NLSM. COMB generates the best objective function values in all 10 problems. The average running times for DACN, NLSM, and COMB are 9.28 seconds, 91.23 seconds, and 7.91 seconds, respectively. In Figure 7.5, we show the plots of the final results generated by DACN, NLSM, and COMB for Problem 1. We also show the plot of a linear transformation of the coordinates produced by NLSM in Figure 7.5(d). We observe that Figures 7.5(a) and 7.5(b) are very similar, while Figure 5(c) appears different. This difference is accounted for by the arbitrary orientation of the solutions produced by the methods. When orientation is taken into account, Figure 7.5(d) is actually similar to Figures 7.5(a) and 7.5(b).

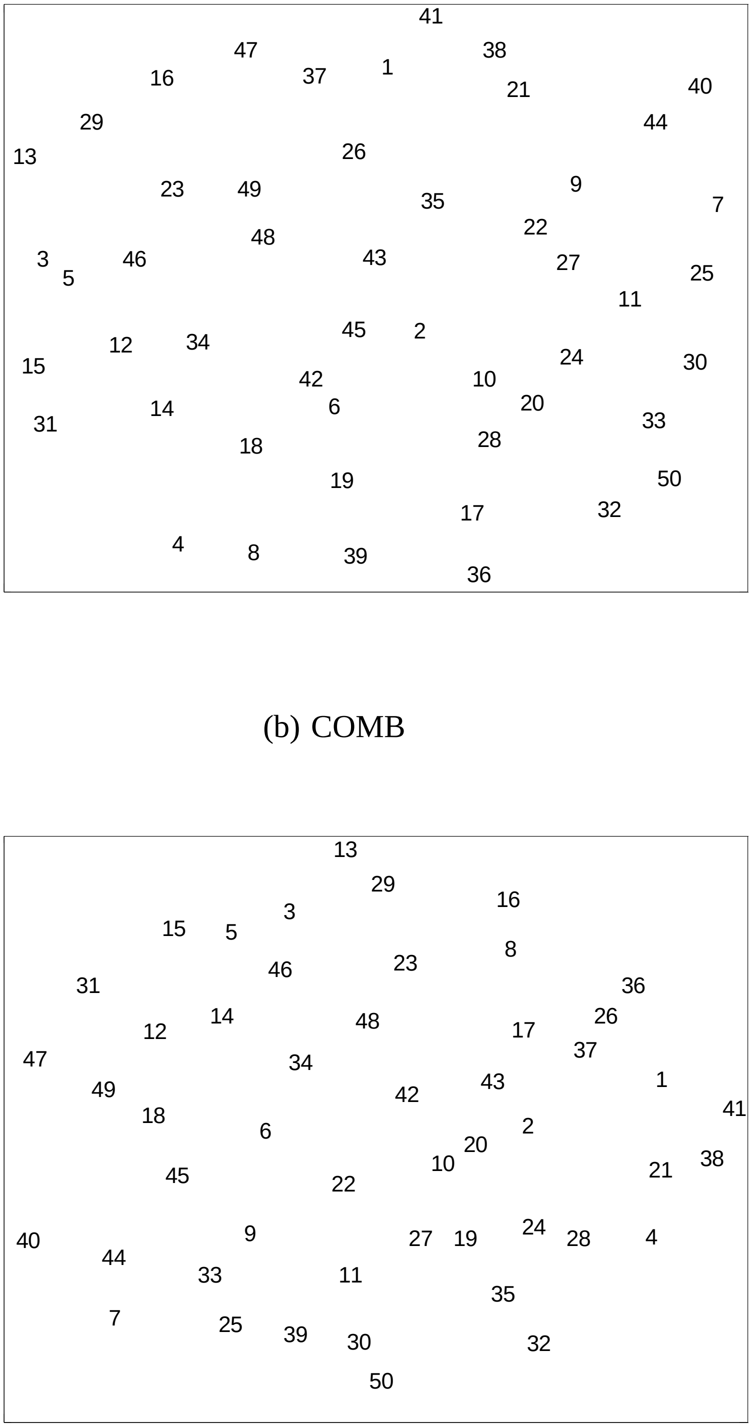
	DACN		NLSM		COMB	
Problem	Best solution	Running time (seconds)	Best solution	Running time (seconds)	Best solution	Running time (seconds)
1	0.07919	9.40	0.08216	97.28	0.07794	8.32
2	0.07602	9.78	0.08221	157.88	0.07479	8.12
3	0.08532	9.29	0.08584	96.38	0.08430	6.72
4	0.08139	9.02	0.09696	96.41	0.08015	8.11
5	0.08254	9.20	0.08845	63.42	0.08153	6.65
6	0.07823	9.52	0.07918	71.04	0.07646	9.75
7	0.07561	9.69	0.08222	95.23	0.07425	8.28
8	0.08371	9.26	0.09750	82.44	0.08249	8.22
9	0.08106	8.57	0.09230	69.98	0.07960	8.23
10	0.08318	9.10	0.08655	82.24	0.08232	6.70
Average		9.28		91.23		7.91

Table 7.5 Results for Problem Set 7: 50-point problems originally in five dimensions.



(a) DACN

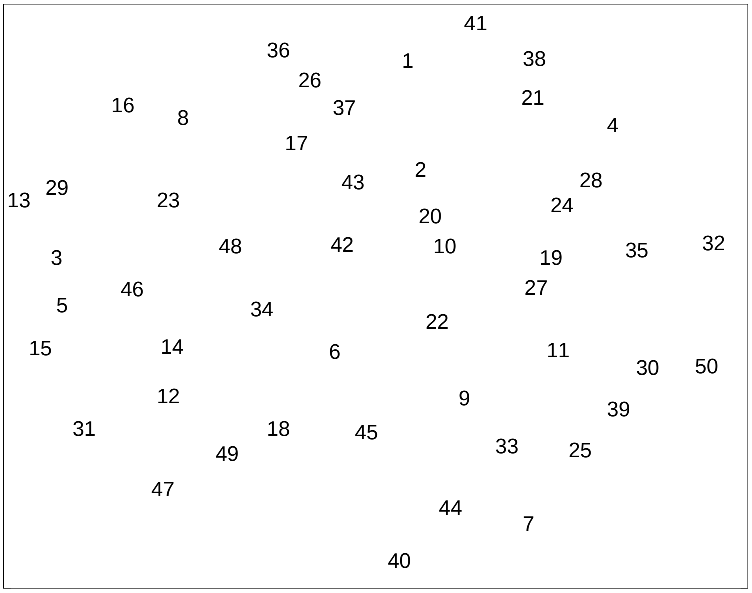
Figure 7.5 Plots for problem 1 from problem set 7.



(b) COMB

(c) NLSM

Figure 7.5 (continued).

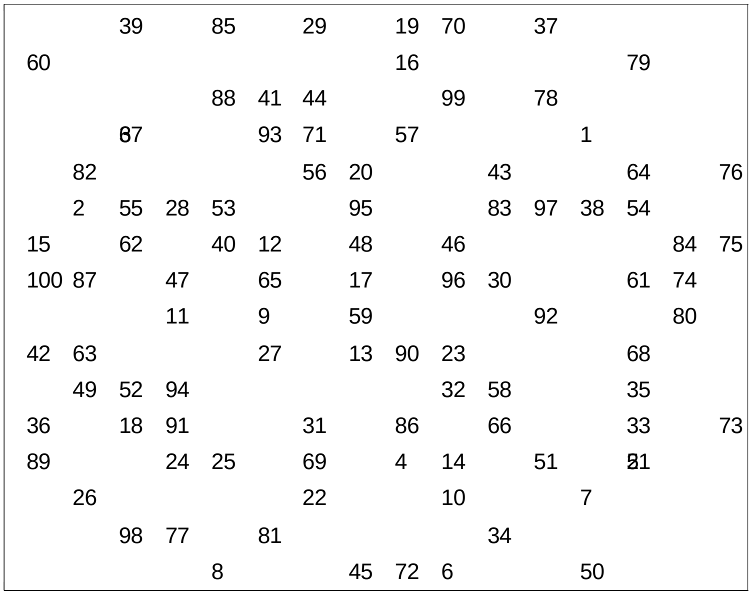


(d) Translated NLSM
Figure 7.5 (continued).

In Table 7.6, we give the results for Problem Set 8. In seven of the 10 problems DACN produces a better objective function value than NLSM. COMB generates the best objective function values in all 10 problems. In Figure 7.6, we show the plots of the final results generated by DACN, NLSM, and COMB for Problem 8. We also show the plot of a linear transformation of the coordinates produced by NLSM in Figure 7.3(d). The three plots produced by DACN, NLSM, and COMB, are similar, once arbitrary orientation is taken into account for in Figure 7.6(d). The average running times for DACN, NLSM, and COMB, are 36.38 seconds, 583.62 seconds, and 39.25 seconds, respectively.

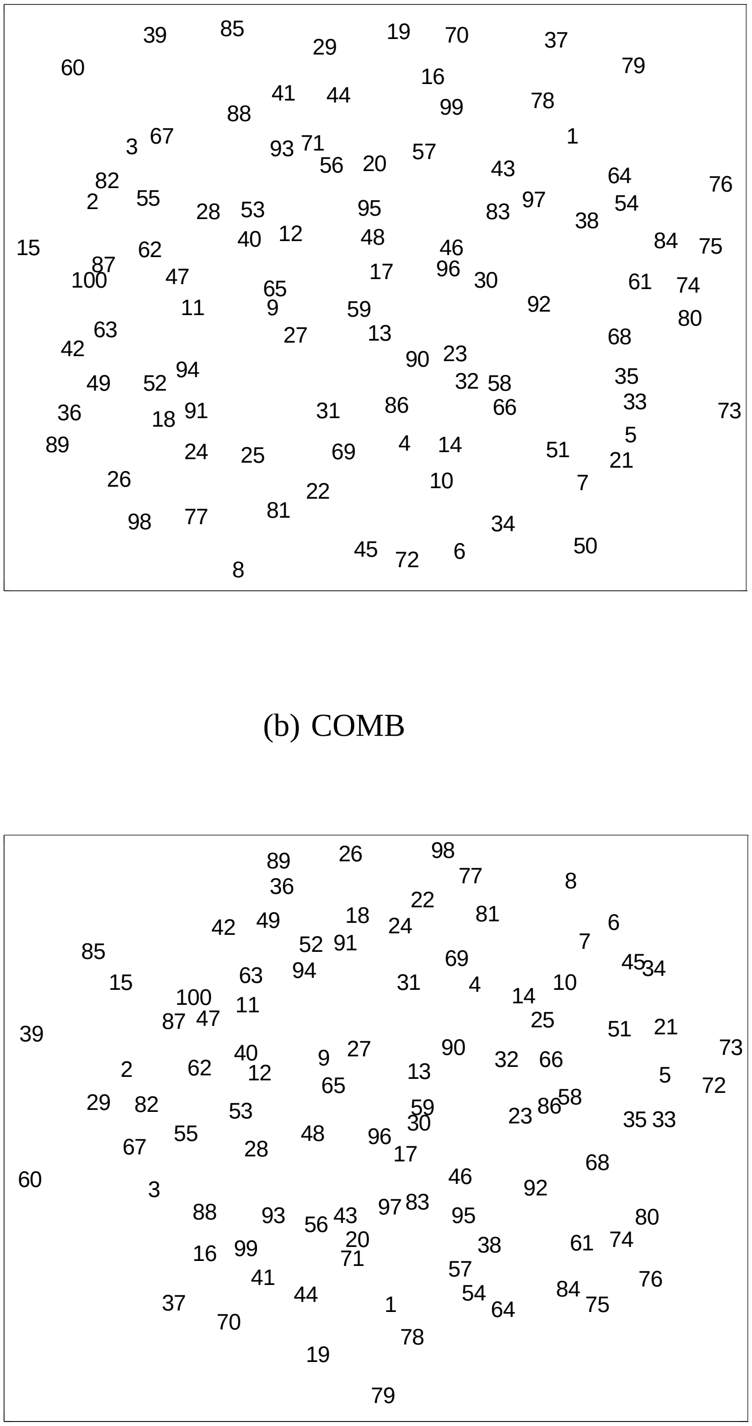
	DACN		NLSM		COMB	
Problem	Best solution	Running time (seconds)	Best solution	Running time (seconds)	Best solution	Running time (seconds)
1	0.08559	39.54	0.09744	514.09	0.08405	40.03
2	0.09069	36.55	0.09419	645.05	0.08881	56.09
3	0.09174	36.63	0.09628	505.83	0.09078	31.91
4	0.09118	33.88	0.09377	548.01	0.08995	39.71
5	0.08694	33.64	0.08950	577.65	0.08533	39.58
6	0.08752	40.36	0.08668	420.01	0.08545	48.24
7	0.08531	34.96	0.08760	722.03	0.08412	39.74
8	0.08699	36.00	0.08974	642.65	0.08575	40.29
9	0.09153	34.80	0.09113	584.15	0.09020	32.64
10	0.08703	37.39	0.08618	676.75	0.08591	24.31
Average		36.38		583.62		39.25

Table 7.6 Results for Problem Set 8: 100-point problems originally in five dimensions.



(a) DACN

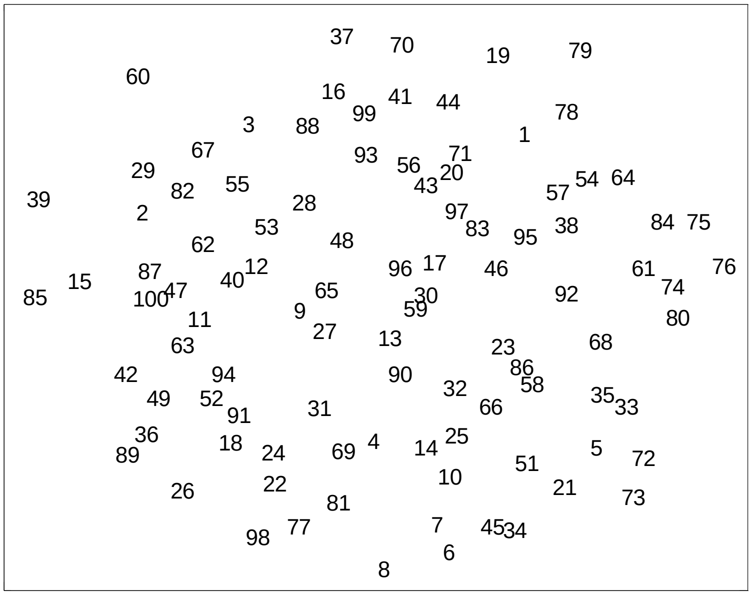
Figure 7.6 Plots for problem 8 from problem set 8.



(b) COMB

(c) NLSM

Figure 7.6 (continued).



(d) Translated NLSM
Figure 7.6 (continued).

7.4 Conclusions

For the 3-dimensional data sets (problem sets 1 and 2), NLSM generates better solutions than DACN, on the average. For the higher dimensional data sets (problem sets 4, 5, 7, and 8), DACN produces better solutions than NLSM. DACN can only make assignments to specific points in the lattice space. NLSM can assign points anywhere in the plane. Despite this limitation, DACN generates results that are comparable in quality to those generated by NLSM. When the final solution from DACN is used as a starting solution in NLSM – this is the COMB heuristic – the final solutions generated by COMB are always better than the results generated by DACN and NLSM. Also, COMB has much smaller average running times than NLSM which are, in fact, quite reasonable. The divide-and-conquer approach provides a good approximate solution in a small amount of computing time. Furthermore, we have demonstrated that this solution is a good starting solution for the nonlinear method. It significantly speeds up convergence and improves solution quality. It should be possible to solve relatively large problems using the COMB heuristic.

Chapter 8: Comparing Discrete Local Search to a Nonlinear Optimization Technique – Proxscal

In this chapter, we compare the results of DACN to the results generated by a nonlinear multidimensional scaling map (NLIM). We use the majorization technique as implemented in SPSS Proxscal (Borg and Groenen, 1997; Laudau and Everitt, 2004). The Proxscal objective function is different (but similar) to the objective functions that we have used in our experiments so far. However, it is easy for us to change the objective function in our technique. DACN is modified in this chapter to use the same objective function used in Proxscal.

In the previous chapter, we compared DACN to a nonlinear Sammon map. We now compare DACN to another nonlinear map because the Sammon map we used was coded in Mathematica and we were not able to consider large-size problems (*n* > 100). Also, the Sammon map code is an experimental research code. We wanted to see how the technique we have developed compared to a standard algorithm.

8.1 Iterative Majorization

The principle of iterative majorization is an easy and powerful strategy for minimization. The general idea is to replace iteratively the original complicated function to be minimized by a simpler function. Iterative majorization generates a sequence of monotonically nonincreasing function values (Borg and Groenen, 1997). So,if a function is bounded from below, iterative majorization usually gives a local minimum.

Problem Set	Dimensions	Number of Points
10	3	150
11	3	300
12	3	500
13	4	150
14	4	300
15	4	500
16	5	150
17	5	300
18	5	500

Table 8.1 Problem sets.

A majorization algorithm for MDS randomly assigns starting coordinates to the points in *M*. The majorization technique is used to reassign the points in *M* so that the objective function value is reduced. The procedure continues until a stopping criterion is met. For NLIM, we use the majorization algorithm that is implemented in SPSS Proxscal. The Proxscal objective function is given by

$$\text{Minimize } \frac{\sum_{i\in M}\sum_{\substack{j\in M\\ j>i}}\sum_{k\in N}\sum_{l\in N}[od(i,j)-nd(k,l)]^2 x_{ik}x_{jl}}{\sum_{i\in M}\sum_{\substack{j\in M\\ j>i}} od(i,j)^2}.$$

We applied DACN and NLIM to several data sets with 150, 300, and 500 points. The data sets were randomly generated from lattice sets in three, four, and five dimensions. For each combination of dimension and size, 10 different problems were generated. The nine problem sets (problem sets 10 – 18) are described in Table 8.1.

For these experiments with DACN, we use *n* = 1024, that is, we stop after Step 5. However, we also record the results after Step 4 also. DACN4 refers to DACN stopped after Step 4 while DACN5 refers to DACN stopped after Step 5. In addition, we use the final solution from DACN as the stating solution for a COMB heuristic for NLIM.

COMB4 refers to the combined heuristic with the final solution from Step 4 used as the starting solution for NLIM and COMB5 uses the final solution from Step 5 as the starting solution.

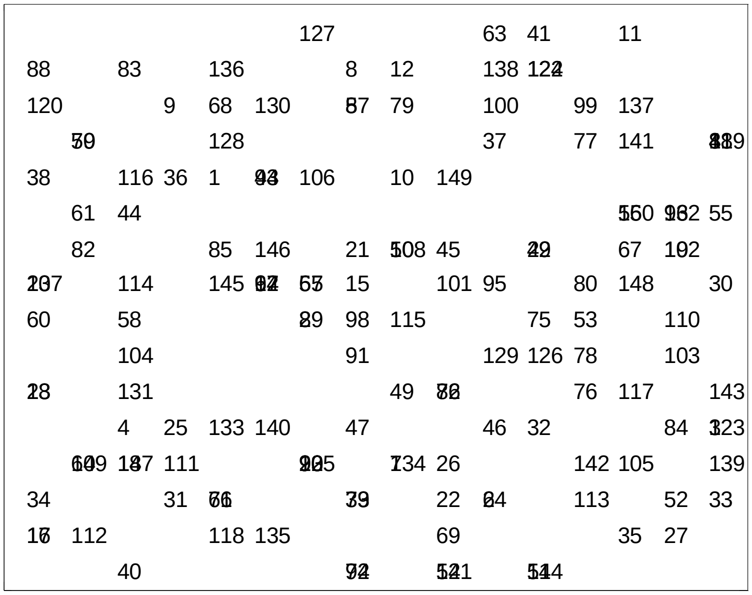
8.2 Results and Analysis

In Table 8.2, we give the results for 150-point problems generated from a lattice in three dimensions (that is, problem set 10). For all 10 problems, DACN5 generates better solutions than DACN4. In seven of the problems, NLIM produces better solutions than DACN4. For problem 8, DACN5 and NLIM, both produce an objective function value of 0.04849. For problem 4, NLIM produces a better solution than DACN5. For the remaining eight problems DACN5 generates better objective function values than NLIM. COMB4 and COMB5 produce better solutions than DACN4, DACN5, and NLIM in all 10 problems. In problem 4, COMB4 and COMB5 generate the same solution. For problem 7 and problem 8, COMB4 produces slightly better objective function values than COMB5. In the remaining seven problems COMB5 generates slightly better solutions than COMB4.

In Figure 8.1, we show plots of the final results generated by DACN4, DACN5, COMB4, COMB5, and NLIM, for Problem 7. We also show the plot of a linear transformation of the coordinates produced by NLIM in Figure 8.1(f). The objective function values indicate that the figures for DACN4, DACN5, COMB4, COMB5, and NLSM, should be similar. Figures 8.1(a), 8.1(b), 8.1(c), and 8.1(d) support this. Figure 8.1(e) appears to be different. This is due to the arbitrary orientation of the problem. We, therefore, used a Procrustes rotation in Matlab to translate the plot of NLIM, obtaining Figure 8.1(f). Figure 8.1(f) is similar to the other figures as expected.

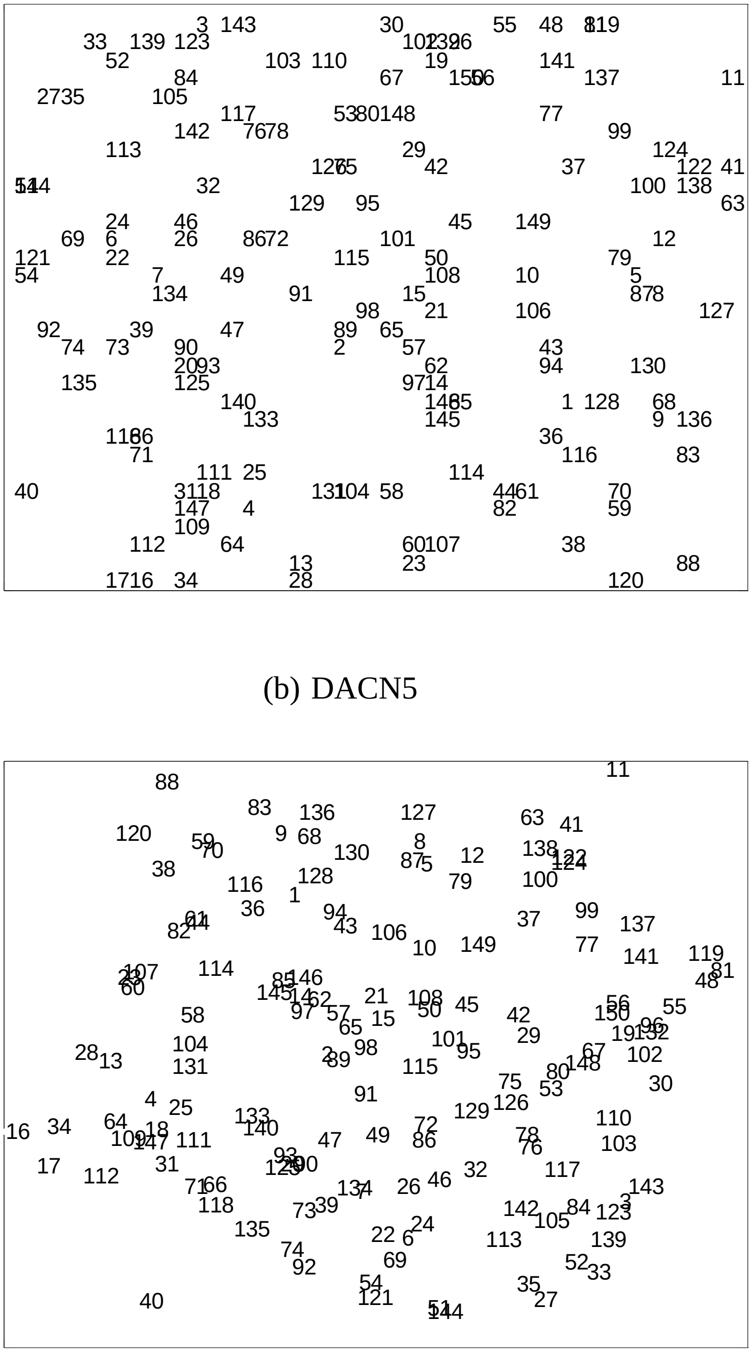
	DACN4		DACN5		NLIM	COMB4	COMB5
Problem	Best solution	Running time (seconds)	Best solution	Running time (seconds)	Best solution	Best solution	Best solution
1	0.04795	36.51	0.04684	101.65	0.04688	0.04637	0.04632
2	0.04760	37.15	0.04614	112.76	0.05252	0.04532	0.04526
3	0.04999	37.13	0.04876	116.87	0.05376	0.04811	0.04803
4	0.04006	40.24	0.03879	93.78	0.03873	0.03829	0.03829
5	0.04291	40.16	0.04154	109.15	0.04331	0.04060	0.04058
6	0.04460	35.09	0.04305	101.46	0.04311	0.04185	0.04166
7	0.04743	40.76	0.04602	108.79	0.04697	0.04477	0.04480
8	0.04984	38.60	0.04849	112.51	0.04849	0.04750	0.04753
9	0.04696	36.91	0.04559	106.19	0.04680	0.04505	0.04499
10	0.04885	41.27	0.04775	126.20	0.04846	0.04732	0.04729
Average		38.38		108.93			

Table 8.2 Results for Problem Set 10: 150-point problems originally in three dimensions.



(a) DACN4

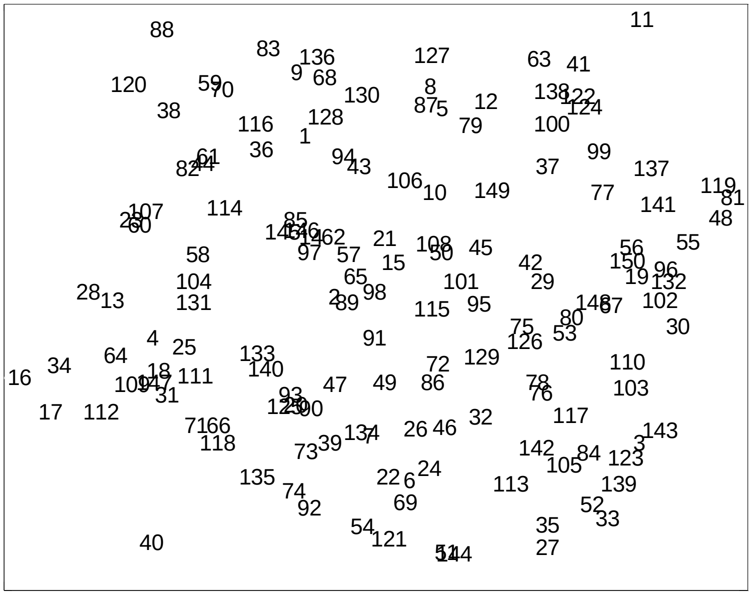
Figure 8.1 Plots for Problem Set 10: Problem 7.



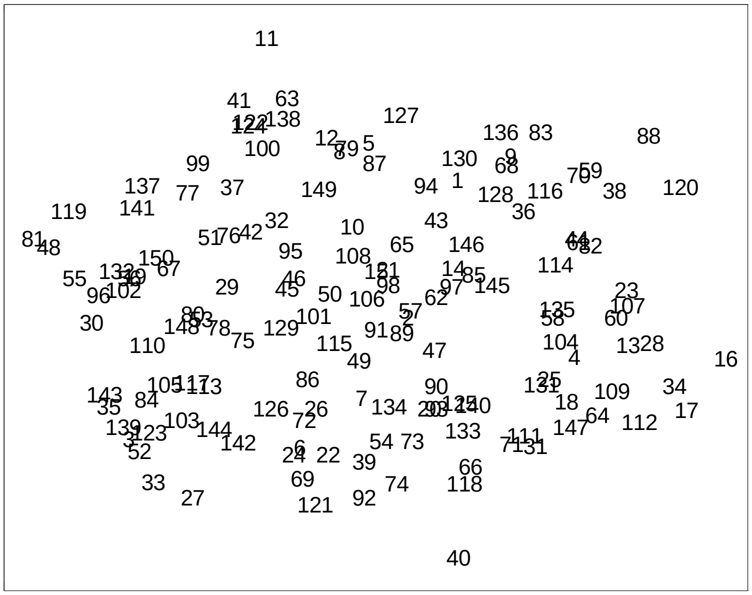
(b) DACN5

(c) COMB4

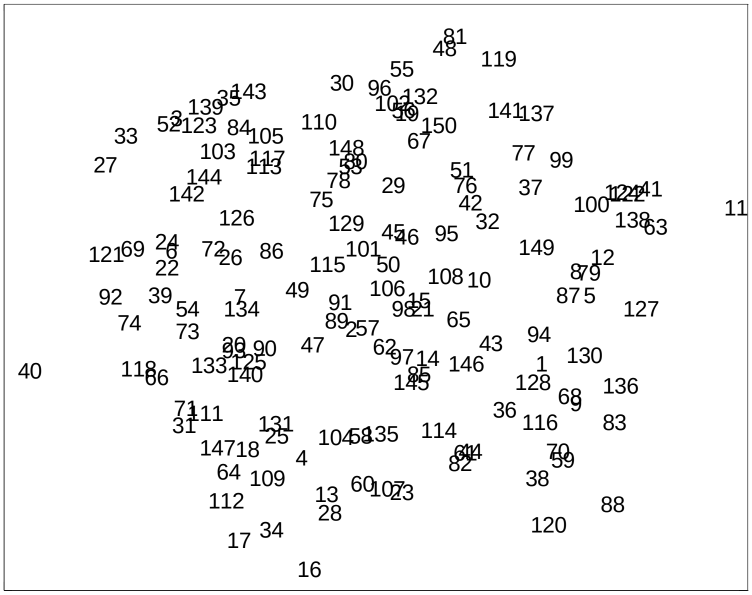
Figure 8.1 (continued).



(d) COMB5



(e) NLIM
Figure 8.1 (continued).



(f) Translated NLIM
Figure 8.1 (continued).

For problem set 10, that is the 150-point problems originally in three dimensions, NLIM produces better solutions than DACN4, and DACN5 produces better solutions than NLIM. The COMB heuristics produce the best solutions of all, with COMB5 producing slightly better solutions than COMB4.

The average running times for DACN4 and DACN5 are 38.38 seconds and 108.93 seconds, respectively. SPSS does not report the running times for Proxscal. Proxscal runs much faster than our heuristic. For example, it takes about one second to generate results for a 150-point problem. It should be noted that NLIM finds one solution, while DACN finds 100 solutions. Also, Proxscal is a commercial solver and our code is a research code.

	DACN4		DACN5		NLIM	COMB4	COMB5
Problem	Best solution	Running time (seconds)	Best solution	Running time (seconds)	Best solution	Best solution	Best solution
1	0.05286	146.31	0.05166	400.13	0.05343	0.05112	0.05107
2	0.05212	166.42	0.05083	396.60	0.05263	0.05032	0.05014
3	0.05233	163.77	0.05091	391.30	0.05358	0.05011	0.05003
4	0.04590	165.22	0.04468	341.98	0.05684	0.04408	0.04402
5	0.05067	158.58	0.04945	369.10	0.05618	0.04880	0.04874
6	0.04938	154.63	0.04810	348.06	0.04871	0.04737	0.04732
7	0.05344	136.33	0.05212	335.28	0.06464	0.05137	0.05128
8	0.05359	148.20	0.05224	361.13	0.05303	0.05136	0.05135
9	0.05282	161.02	0.05148	332.40	0.05418	0.05067	0.05055
10	0.05624	169.85	0.05501	374.11	0.57780	0.05439	0.05434
Average		157.03		365.01			

Table 8.3 Results for Problem Set 11: 300-point problems originally in three dimensions.

In Table 8.3, we show the results for the 300-point problems originally in three dimensions, that is, problem set 11. In all 10 problems DACN5 produces better solutions than DACN4 as is expected. NLIM produces better objective function values than DACN4 in problem 6 and problem 8. For the remaining eight problems, DACN4 produces better objective function values than NLIM. For all 10 problems, DACN5 produces better solutions than NLIM. The COMB heuristics produce the best solutions for all 10 problems. COMB5 produces slightly better objective function values than COMB4.

For the 300-point problems in three dimensions, the average running times are 157.03 seconds and 365.01 seconds, for DACN4 and DACN5, respectively. DACN produces better solutions than NLIM, and COMB produces the best solutions of all. DACN5 produces better solutions than DACN4 and COMB5 produces slightly better solutions than COMB4.

	DACN4		DACN5		NLIM	COMB4	COMB5
Problem	Best solution	Running time (seconds)	Best solution	Running time (seconds)	Best solution	Best solution	Best solution
1	0.05572	409.95	0.05451	896.57	0.06514	0.05396	0.05390
2	0.05136	498.29	0.05013	1031.77	0.05601	0.04956	0.04950
3	0.05569	422.81	0.05456	838.92	0.05615	0.05392	0.05385
4	0.05030	473.21	0.04903	898.45	0.05800	0.04839	0.04832
5	0.05304	437.62	0.05179	918.22	0.05424	0.05119	0.05114
6	0.05022	609.51	0.04888	1223.60	0.04884	0.04818	0.04806
7	0.05584	442.29	0.05449	964.58	0.05585	0.05399	0.05381
8	0.05478	389.08	0.05343	891.31	0.05604	0.05272	0.05262
9	0.05527	393.05	0.05401	844.27	0.05703	0.05315	0.05313
10	0.05433	495.94	0.05304	1026.49	0.05884	0.05250	0.05243
Average		457.18		953.42			

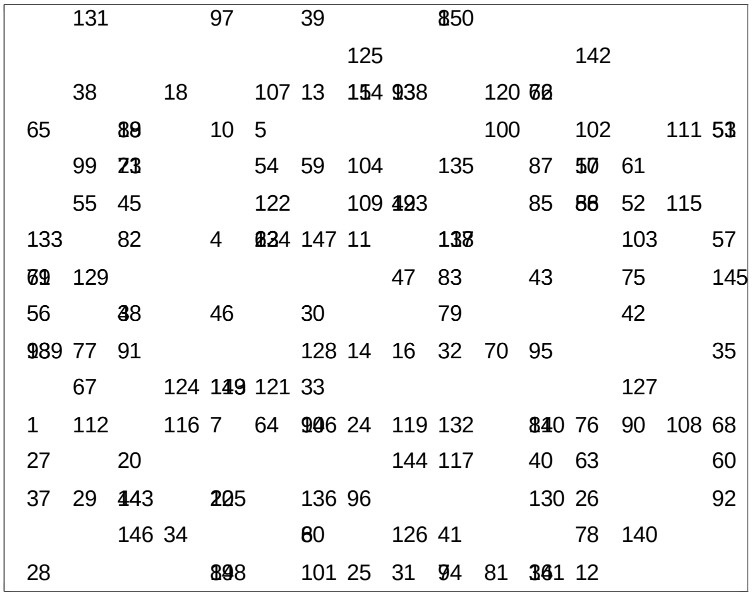
Table 8.4 Results for Problem Set 12: 500-point problems originally in three dimensions.

We show the results for the 500-point problems originally in three dimensions, that is, problem set 12, in Table 8.4. The results are similar to those for problem set 11 discussed above. For problem 6, NLIM produces a better objective function value than DACN4 and DACN5. For the remaining nine problems, DACN4 and DACN5 produce better solutions than NLIM, with DACN5 generating better solutions than DACN4. The COMB heuristics produce the best solutions in all 10 problems. COMB5 produces slightly better solutions than COMB4. The average running times for DACN4 and DACN5 are 457.18 seconds and 953.42 seconds, respectively.

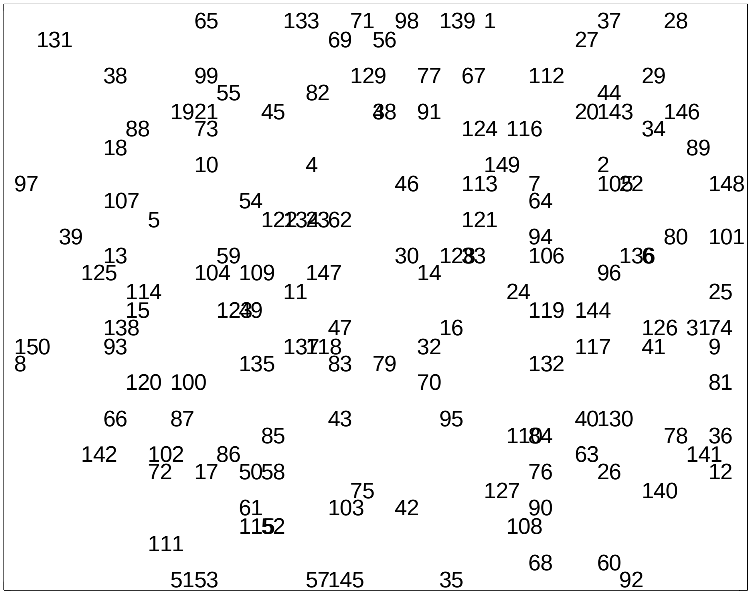
For the problem sets originally in three dimensions, DACN5 produces better objective function values than DACN4 as we expected. DACN5 produces better solutions than NLIM. For smaller problems NLIM appears to do better than DACN4. As the problem size increases DACN4 does better than NLP. COMB always produces the best solutions with COMB5 producing slightly better solutions than COMB4.

	DACN4		DACN5		NLIM	COMB4	COMB5
Problem	Best solution	Running time (seconds)	Best solution	Running time (seconds)	Best solution	Best solution	Best solution
1	0.06958	46.99	0.06848	124.83	0.07633	0.06805	0.06796
2	0.07350	45.30	0.07250	122.55	0.07810	0.07217	0.07211
3	0.07111	48.09	0.06964	129.50	0.07263	0.06884	0.06861
4	0.06974	41.99	0.06867	120.89	0.07584	0.06814	0.06805
5	0.06742	47.59	0.06626	127.73	0.06706	0.06531	0.06519
6	0.07253	44.67	0.07143	115.66	0.07216	0.07087	0.07080
7	0.06579	46.11	0.06453	116.41	0.06481	0.06387	0.06375
8	0.06203	45.56	0.06100	127.64	0.06271	0.06058	0.06046
9	0.06826	51.88	0.06713	123.34	0.07750	0.06639	0.06641
10	0.07680	49.78	0.07579	141.99	0.08051	0.07533	0.07527
Average		46.80		125.05			

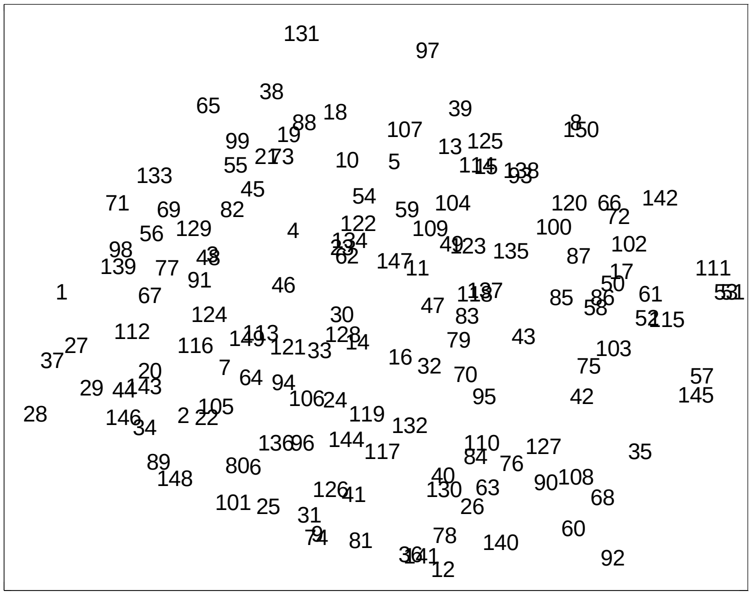
Table 8.5 Results for Problem Set 13: 150-point problems originally in four dimensions.



(a) DACN4
Figure 8.2 Plots for Problem Set 13: Problem 1.

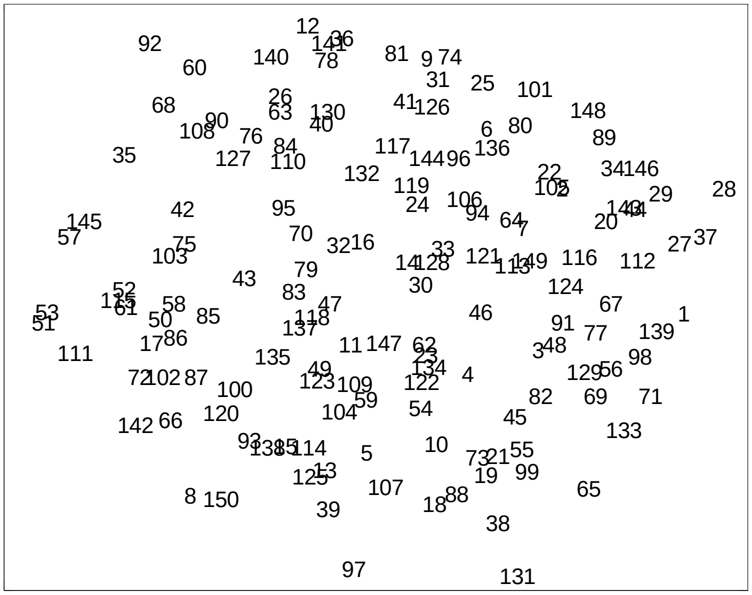


(b) DACN5

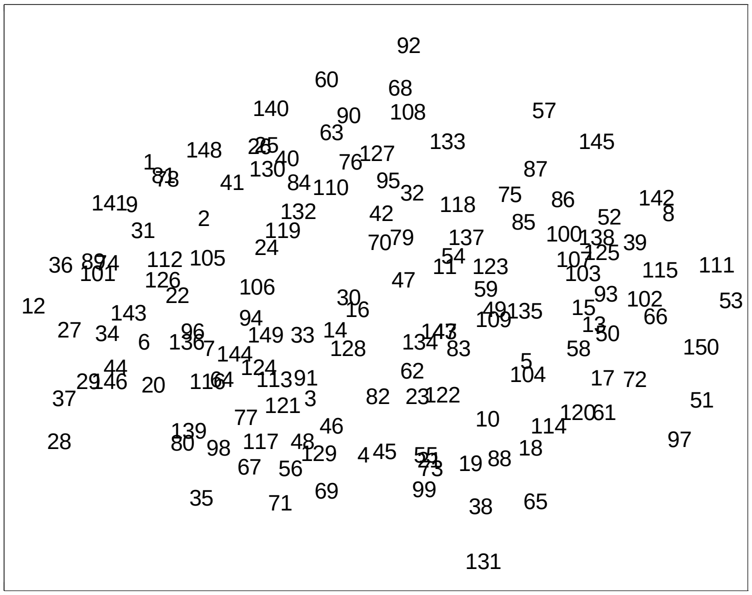


(c) COMB4

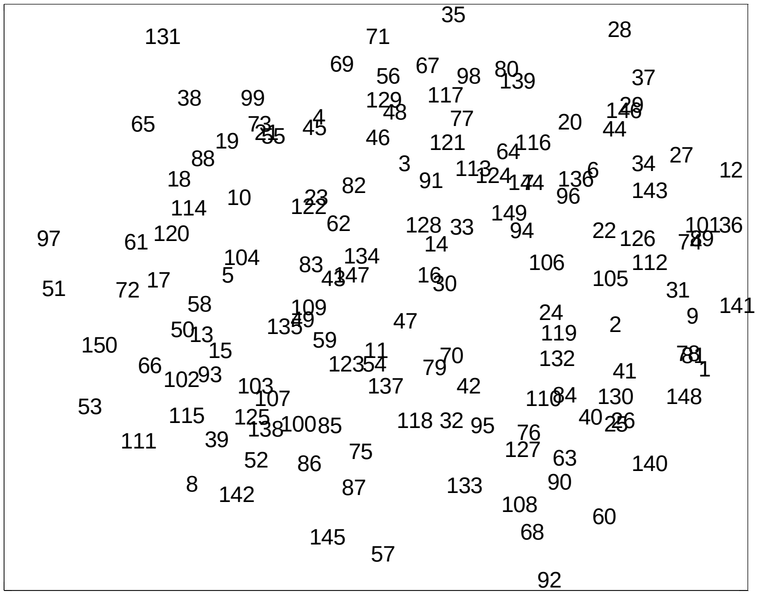
Figure 8.2 (continued).



(d) COMB5



(e) NLIM
Figure 8.2 (continued).



(f) Translated NLIM
Figure 8.2 (continued).

In Table 8.5, we show the results for problem set 13, that is, the 150-point problems originally in four dimensions. NLIM produces better solutions than DACN4 in three of the 10 problems. In all 10 problems, DACN5 produces better solutions than DACN4 and NLIM. COMB4 and COMB5 produce better solutions than DACN4, DACN5, and NLIM in all 10 problems. For problem 9, COMB4 produces a slightly better solution than COMB5. In the remaining nine problems, COMB5 produces better solutions than COMB4. The average running times for DACN4 and DACN5 are 46.80 seconds and 125.05 seconds, respectively.

In Figure 8.2, we show plots of the final results generated by DACN4, DACN5, COMB4, COMB5, and NLIM, Problem 1. We also show the plot of a linear transformation of the coordinates produced by NLIM in Figure 8.2(f). Figures 8.2 (a), 8.2(b), 8.2(c), 8.2(d), and 8.2(f) are similar as expected from their similar objective function values.

	DACN4		DACN5		NLIM	COMB4	COMB5
Problem	Best solution	Running time (seconds)	Best solution	Running time (seconds)	Best solution	Best solution	Best solution
1	0.07676	244.07	0.07570	521.36	0.08072	0.07510	0.07504
2	0.07693	197.88	0.07573	483.84	0.08157	0.07494	0.07482
3	0.07320	241.75	0.07190	612.70	0.07841	0.07113	0.07092
4	0.07602	191.10	0.07496	438.09	0.07566	0.07431	0.07424
5	0.07534	196.25	0.07426	442.94	0.08930	0.07363	0.07357
6	0.07551	188.58	0.07436	445.15	0.07590	0.07370	0.07359
7	0.07330	214.54	0.07196	552.23	0.07720	0.07109	0.07104
8	0.07371	197.77	0.07246	437.05	0.07879	0.07165	0.07171
9	0.07444	181.26	0.07327	393.39	0.07740	0.07262	0.07249
10	0.07986	203.27	0.07879	489.25	0.08452	0.07827	0.07824
Average		205.65		481.60			

Table 8.6 Results for Problem Set 14: 300-point problems originally in four dimensions.

	DACN4		DACN5		NLIM	COMB4	COMB5
Problem	Best solution	Running time (seconds)	Best solution	Running time (seconds)	Best solution	Best solution	Best solution
1	0.08233	566.39	0.08127	1234.63	0.08898	0.08083	0.08076
2	0.07865	573.55	0.07755	1212.13	0.07923	0.07706	0.07694
3	0.07882	537.59	0.07771	1096.30	0.08688	0.07717	0.07706
4	0.07923	502.92	0.07815	1023.73	0.08037	0.07754	0.07746
5	0.07815	531.06	0.07722	1213.91	0.08190	0.07686	0.07677
6	0.07780	763.19	0.07665	1484.77	0.07938	0.07609	0.07594
7	0.07621	599.04	0.07502	1147.43	0.08677	0.07428	0.07422
8	0.08073	514.09	0.07963	1070.72	0.08543	0.07902	0.07898
9	0.07841	561.62	0.07727	1221.12	0.08336	0.07669	0.07662
10	0.08112	516.07	0.08001	1090.98	0.08364	0.07944	0.07930
Average		566.55		1179.57			

Table 8.7 Results for Problem Set 15: 500-point problems originally in four dimensions.

We show the results for the 300-point problems and the 500-point problems originally in four dimensions in Tables 8.6 and 8.7, respectively. In Table 8.6, for problem 4, NLIM produces a better objective function value than DACN4. In the remaining nine problems, DACN4 produces better objective function values than NLIM. DACN5 produces better solutions than DACN4 and NLIM in all 10 problems. The COMB heuristics produce the best solutions of all. COMB4 produces a better objective function value than COMB5 for problem 8. In the remaining nine problems, COMB5 produces better solutions than COMB4.

For the 500-point problems, DACN5 produces better solutions than DACN4 in all 10 problems. Both DACN4 and DACN5 produce better solutions than NLIM in all 10 problems. The COMB heuristics produce the best solutions with COMB5 producing slightly better objective function values than COMB4.

The average running times are 205.65 seconds and 481.60 seconds for DACN4 and DACN5, respectively, for the 300-point problems. For the 500-point problems, the average running times for DACN4 and DACN5 are 566.55 seconds and 1179.57 seconds, respectively.

The results for the problem sets originally in four dimensions are similar to those for the problem sets originally in three dimensions. DACN5 produces better objective function values than DACN4. DACN5 always produces better solutions than NLIM. DACN4 does about the same as NLIM for the 150-point problems. As the problem size increases, DACN4 appears to do better than NLIM. COMB always produces the best solutions with COMB5 producing slightly better solutions than COMB4.

	DACN4		DACN5		NLIM	COMB4	COMB5
Problem	Best solution	Running time (seconds)	Best solution	Running time (seconds)	Best solution	Best solution	Best solution
1	0.08794	42.21	0.08710	117.82	0.09525	0.08675	0.08659
2	0.08600	41.60	0.08504	117.71	0.08783	0.08472	0.08457
3	0.08758	43.73	0.08653	116.98	0.09199	0.08630	0.08600
4	0.08676	41.36	0.08567	113.91	0.09005	0.08532	0.08505
5	0.08192	41.44	0.08082	112.89	0.08199	0.07994	0.07990
6	0.08140	42.25	0.08047	112.91	0.09506	0.08009	0.08005
7	0.07984	39.93	0.07873	106.15	0.08963	0.07845	0.07813
8	0.08002	40.35	0.07914	114.11	0.08874	0.07894	0.07880
9	0.08527	39.75	0.08431	104.30	0.09860	0.08409	0.08396
10	0.08974	43.90	0.08863	119.17	0.09910	0.08852	0.08805
Average		41.65		113.60			

Table 8.8 Results for Problem Set 16: 150-point problems originally in five dimensions.

	DACN4		DACN5		NLIM	COMB4	COMB5
Problem	Best solution	Running time (seconds)	Best solution	Running time (seconds)	Best solution	Best solution	Best solution
1	0.09145	189.34	0.09026	442.05	0.09567	0.08994	0.08945
2	0.09441	180.47	0.09338	442.72	0.10082	0.09322	0.09271
3	0.08949	197.28	0.08851	489.52	0.10114	0.08814	0.08796
4	0.09292	197.99	0.09197	459.78	0.09693	0.09159	0.09148
5	0.08971	182.06	0.08861	416.43	0.08913	0.08799	0.08784
6	0.08933	159.66	0.08832	352.96	0.09426	0.08798	0.08780
7	0.08814	170.52	0.08703	365.46	0.09357	0.08642	0.08635
8	0.09073	188.73	0.08971	413.96	0.09457	0.08937	0.08924
9	0.08950	194.51	0.08838	468.07	0.10040	0.08788	0.08779
10	0.09310	197.26	0.09216	466.11	0.09867	0.09193	0.09184
Average		185.78		431.71			

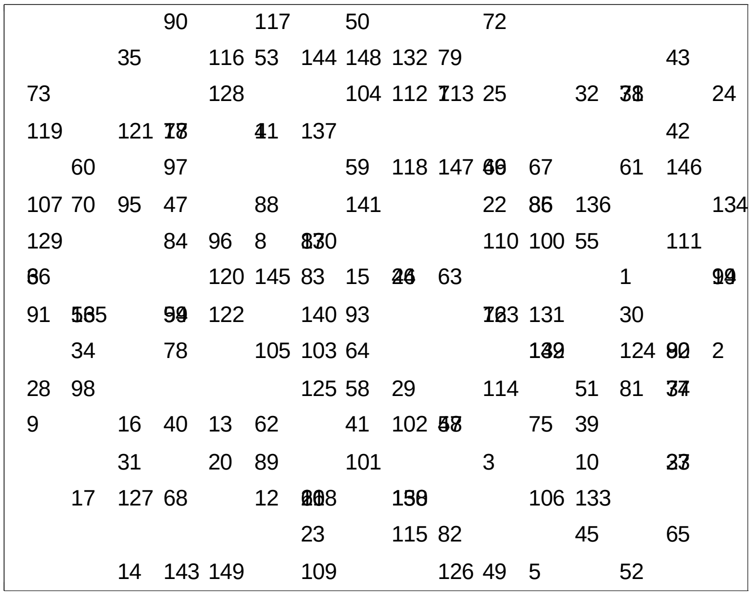
Table 8.9 Results for Problem Set 17: 300-point problems originally in five dimensions.

	DACN4		DACN5		NLIM	COMB4	COMB5
Problem	Best solution	Running time (seconds)	Best solution	Running time (seconds)	Best solution	Best solution	Best solution
1	0.09653	540.23	0.09564	1092.88	0.10238	0.09540	0.09537
2	0.09470	525.70	0.09374	1220.85	0.09722	0.09334	0.09321
3	0.09402	519.81	0.09305	1045.87	0.10623	0.09266	0.09255
4	0.09612	514.73	0.09521	1030.91	0.09920	0.09496	0.09487
5	0.09281	550.84	0.09181	1131.11	0.10135	0.09130	0.09118
6	0.09175	522.75	0.09075	1076.01	0.10150	0.09042	0.09026
7	0.09333	514.90	0.09226	1024.48	0.10401	0.09171	0.09162
8	0.09592	466.96	0.09492	986.55	0.10012	0.09454	0.09442
9	0.09421	532.36	0.09330	1100.63	0.10032	0.09301	0.09284
10	0.09382	554.03	0.09282	1149.14	0.10260	0.09245	0.09233
Average		524.23		1085.84			

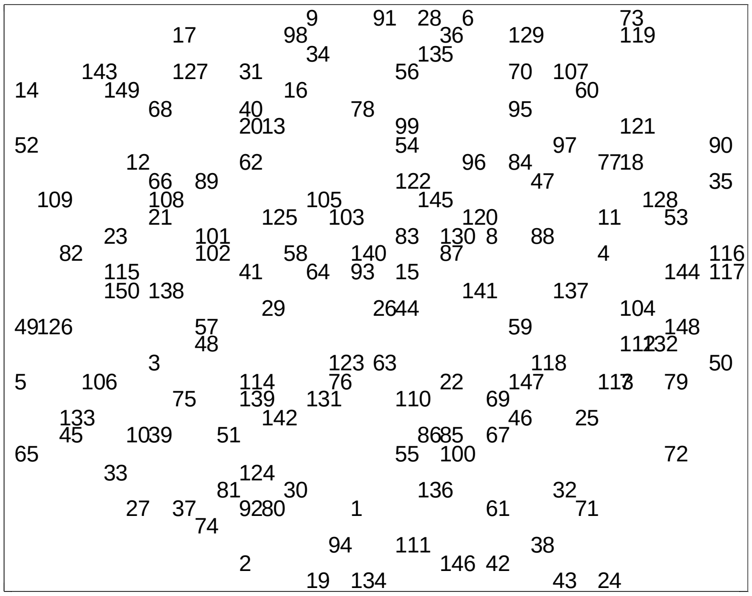
Table 8.10 Results for Problem Set 18: 500-point problems originally in five dimensions.

In Tables 8.8, 8.9, and 8.10, we show the results for the 150-point problems, 300-point problems, and 500-point problems, respectively, originally in five dimensions. In all 30 problems, DACN5 produces better objective function values than DACN4. For problem 5 of the 300-point problems, NLIM generates a better objective function value than DACN4. In the remaining 300-point problems, and all the 150-point problems and 500-point problems, DACN4 generates better solutions than NLIM. COMB4 generates better solutions than DACN5, while COMB5 generates the best solutions of all, generating slightly better objective function values than COMB4.

In Table 8.8, the average running time for the 150-point problems for DACN4 and DACN5 are 41.65 seconds and 113.60 seconds, respectively. The average running times for DACN4 and DACN5 for the 300-point problems are 185.78 seconds and 431.71 seconds, respectively. For the 500-point problems, the average running times for DACN4 and DACN5 are 524.23 seconds and 1085.84 seconds, respectively.

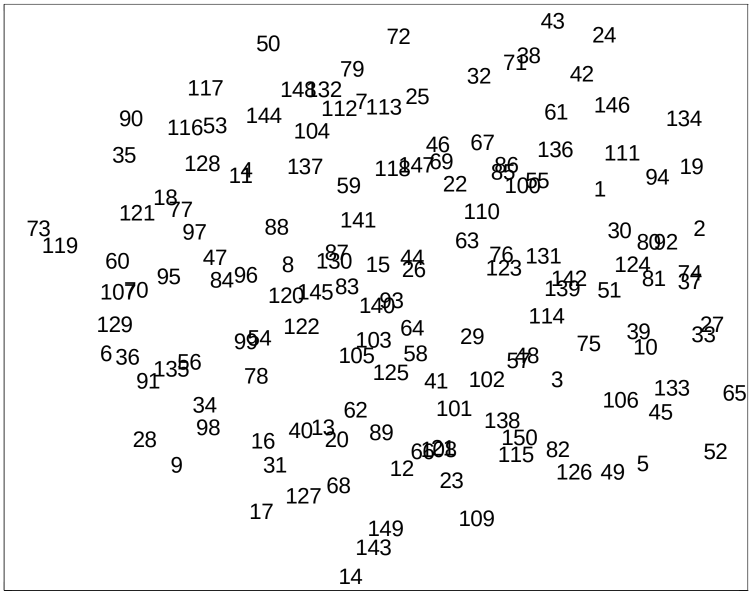


(a) DACN4

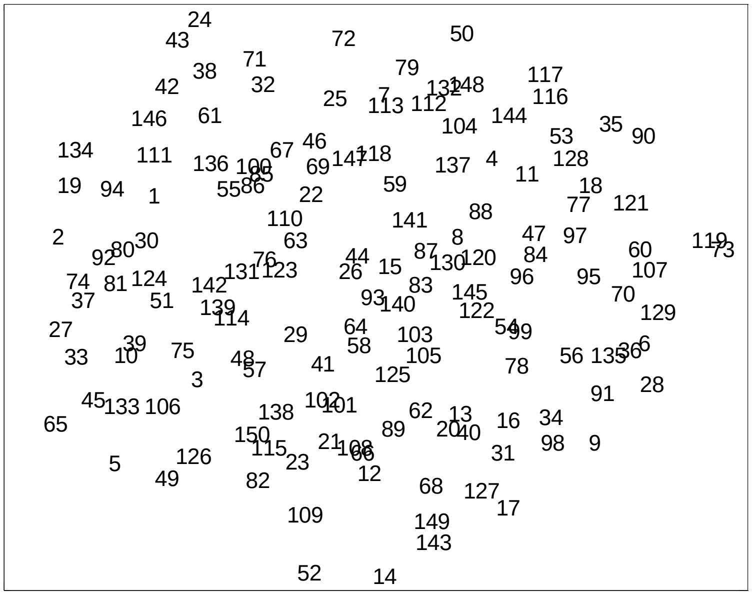


(b) DACN5

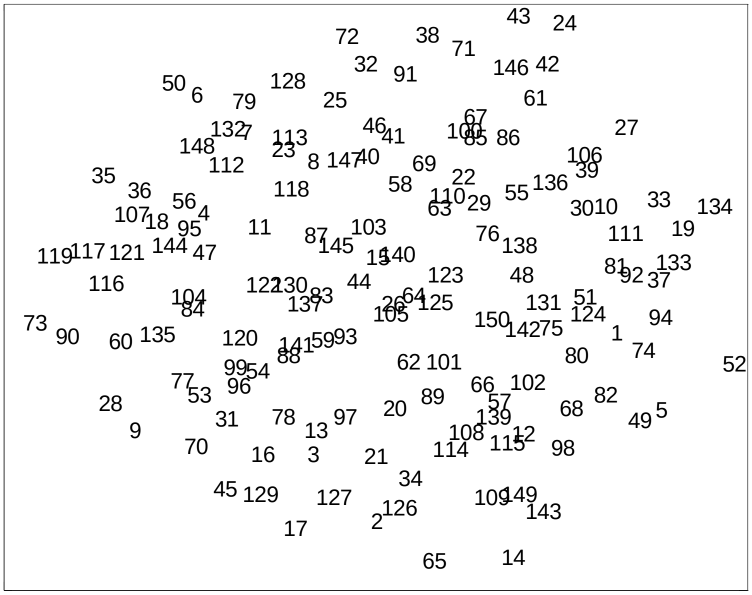
Figure 8.3 Plots for Problem Set 16: Problem 3.



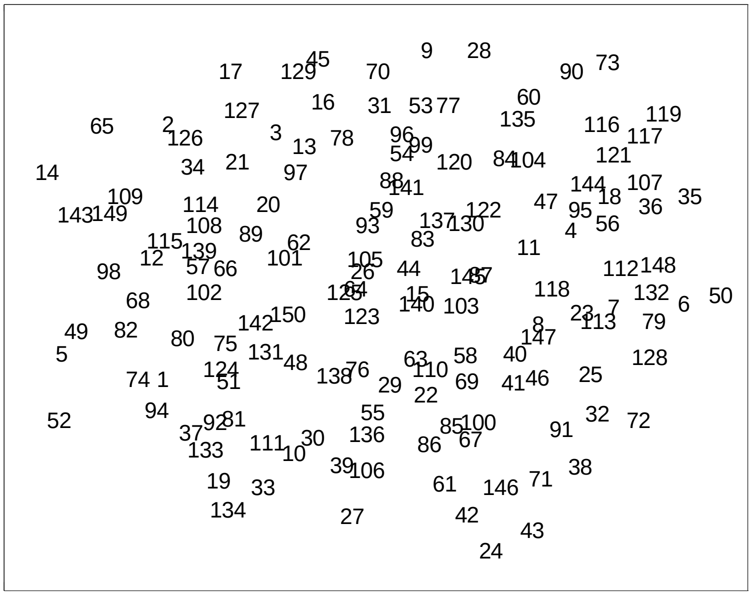
(c) COMB4



(d) COMB5
Figure 8.3 (continued).



(e) NLIM



(f) Translated NLIM
Figure 8.3 (continued).

In Figure 8.3, we show plots of the final results generated by DACN4, DACN5, COMB4, COMB5, and NLIM, Problem 3. We also show the plot of a linear transformation of the coordinates produced by NLIM in Figure 8.3(f). Figures 8.3 (a), 8.3(b), 8.3(c), 8.3(d), and 8.3(f) are similar as expected from their similar objective function values.

Recall, that DACN finds 100 solutions while NLIM finds only one solution. Since DACN is producing better quality solutions than NLIM but in a longer time we investigate running DACN with only 10 solutions instead of 100. We also report the solution and running time after the points have been assigned to 64 lattice points (DACN3) and when this result is used as the starting solution for NLIM (COMB3). We give the results for these experiments in Table 8.11 to Table 8.19. The results for with 100 solutions are slightly better than those with only 10 solutions. There is a great improvement in the running times though. The running times using 100 solutions are about 9.72 times those using only 10 solutions. On the average DACN4 and DACN5 still produce better quality solutions than NLIM. COMB4 and COMB5 produce the best results of all. The results from DACN3 are not too good. However, the running times are very low for DACN3. As the number of points and dimensions increase COMB3 does better than NLIM.

	DACN3		DACN4		DACN5		NLIM	COMB3	COMB4	COMB5
Problem	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Best solution	Best solution	Best solution
1	0.05852	1.89	0.04799	3.82	0.04690	10.83	0.04688	0.05233	0.04640	0.04635
2	0.05442	2.14	0.04764	4.38	0.04615	12.23	0.05252	0.04599	0.04533	0.04525
3	0.05711	1.66	0.05036	3.44	0.04885	10.06	0.05376	0.05078	0.04804	0.04797
4	0.05687	1.95	0.04006	4.65	0.03879	10.25	0.03873	0.04963	0.03833	0.03829
5	0.04773	1.82	0.04295	4.10	0.04163	12.23	0.04331	0.04075	0.04062	0.04064
6	0.05010	1.91	0.04460	4.03	0.04306	11.99	0.04311	0.04208	0.04185	0.04172
7	0.05848	2.03	0.04743	4.10	0.04602	11.67	0.04697	0.05248	0.04475	0.04481
8	0.05496	2.27	0.05000	5.15	0.04862	14.42	0.04849	0.04908	0.04755	0.04754
9	0.06491	1.96	0.04695	4.54	0.04559	12.53	0.04680	0.05743	0.04510	0.04499
10	0.05668	1.99	0.04886	4.07	0.04777	11.23	0.04846	0.04898	0.04733	0.04729
Average		1.96		4.23		11.74				

Table 8.11 Results for Problem Set 10: 150-point problems originally in three dimensions with 10 solutions for DACN.

	DACN3		DACN4		DACN5		NLIM	COMB3	COMB4	COMB5
Problem	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Best solution	Best solution	Best solution
1	0.06439	8.02	0.05285	17.10	0.05166	41.58	0.05343	0.05797	0.05113	0.05107
2	0.06244	10.24	0.05226	18.64	0.05085	39.42	0.05263	0.05494	0.05036	0.05021
3	0.05738	10.18	0.05234	18.45	0.05091	46.33	0.05358	0.05016	0.05007	0.05003
4	0.05091	12.84	0.04590	21.37	0.04469	42.12	0.05684	0.04413	0.04407	0.04403
5	0.06151	8.33	0.05068	15.49	0.04945	37.75	0.05618	0.05543	0.04876	0.04874
6	0.05450	8.13	0.04940	15.47	0.04810	41.67	0.04871	0.04757	0.04739	0.04732
7	0.06019	8.60	0.05354	16.36	0.05223	41.42	0.06464	0.05387	0.05156	0.05156
8	0.05848	9.61	0.05370	18.28	0.05239	40.34	0.05303	0.05201	0.05196	0.05181
9	0.05772	9.18	0.05282	20.28	0.05148	39.86	0.05418	0.05113	0.05067	0.05055
10	0.06344	9.66	0.05625	18.03	0.05501	41.53	0.57780	0.05644	0.05439	0.05434
Average		9.48		17.95		41.20				

Table 8.12 Results for Problem Set 11: 300-point problems originally in three dimensions with 10 solutions for DACN.

	DACN3		DACN4		DACN5		NLIM	COMB3	COMB4	COMB5
Problem	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Best solution	Best solution	Best solution
1	0.06468	24.93	0.05572	43.89	0.05451	99.58	0.06514	0.05794	0.05397	0.05389
2	0.05654	27.84	0.05137	54.88	0.05014	112.01	0.05601	0.04961	0.04955	0.04949
3	0.06048	30.18	0.05569	48.29	0.05456	92.47	0.05615	0.05782	0.05392	0.05386
4	0.06193	29.79	0.05031	52.62	0.04903	94.98	0.05800	0.05465	0.04838	0.04833
5	0.05776	30.01	0.05305	49.20	0.05179	97.81	0.05424	0.05130	0.05120	0.05114
6	0.05590	37.65	0.05022	64.61	0.04888	127.37	0.04884	0.04798	0.04811	0.04805
7	0.06104	28.22	0.05585	48.72	0.05455	102.63	0.05585	0.05428	0.05402	0.05389
8	0.06011	26.91	0.05478	45.45	0.05343	98.57	0.05604	0.06273	0.05272	0.05262
9	0.06023	24.26	0.05528	40.88	0.05400	89.58	0.05703	0.05409	0.05318	0.05313
10	0.06445	27.39	0.05433	52.65	0.05305	109.70	0.05884	0.05760	0.05248	0.05243
Average		28.72		50.12		102.47				

Table 8.13 Results for Problem Set 12: 500-point problems originally in three dimensions with 10 solutions for DACN.

	DACN3		DACN4		DACN5		NLIM	COMB3	COMB4	COMB5
Problem	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Best solution	Best solution	Best solution
1	0.07422	1.65	0.06962	3.66	0.06849	10.21	0.07633	0.06820	0.06804	0.06798
2	0.07904	2.15	0.07435	4.29	0.07300	11.29	0.07810	0.07304	0.07254	0.07233
3	0.07576	1.98	0.07128	4.55	0.06963	12.30	0.07263	0.06942	0.06909	0.06863
4	0.07393	1.60	0.06974	3.59	0.06867	10.67	0.07584	0.06825	0.06813	0.06805
5	0.07268	1.64	0.06762	3.61	0.06634	11.43	0.06706	0.06549	0.06534	0.06517
6	0.07693	1.72	0.07253	3.60	0.07145	10.19	0.07216	0.07100	0.07087	0.07079
7	0.07057	1.93	0.06579	3.67	0.06454	10.37	0.06481	0.06447	0.06387	0.06377
8	0.06651	2.20	0.06203	4.36	0.06101	13.35	0.06271	0.06062	0.06055	0.06049
9	0.07262	2.11	0.06827	4.21	0.06714	10.01	0.07750	0.06639	0.06640	0.06642
10	0.08136	2.10	0.07687	4.97	0.07581	12.24	0.08051	0.07542	0.07541	0.07532
Average		1.91		4.05		11.21				

Table 8.14 Results for Problem Set 13: 150-point problems originally in four dimensions with 10 solutions for DACN.

	DACN3		DACN4		DACN5		NLIM	COMB3	COMB4	COMB5
Problem	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Best solution	Best solution	Best solution
1	0.08098	8.93	0.07680	27.58	0.07571	58.45	0.08072	0.07543	0.07514	0.07505
2	0.08139	9.53	0.07696	19.22	0.07575	46.63	0.08157	0.07507	0.07497	0.07484
3	0.07805	8.29	0.07324	17.75	0.07196	55.24	0.07841	0.07099	0.07099	0.07099
4	0.08024	9.11	0.07603	16.88	0.07498	38.06	0.07566	0.07441	0.07431	0.07427
5	0.07947	8.97	0.07535	15.57	0.07427	40.02	0.08930	0.07383	0.07359	0.07358
6	0.07989	7.98	0.07551	15.87	0.07436	35.74	0.07590	0.07390	0.07371	0.07360
7	0.07815	9.43	0.07336	19.34	0.07198	64.68	0.07720	0.07123	0.07116	0.07103
8	0.07844	8.55	0.07373	15.06	0.07251	33.15	0.07879	0.07181	0.07176	0.07173
9	0.07903	10.27	0.07446	16.60	0.07329	39.65	0.07740	0.07270	0.07262	0.07256
10	0.08474	9.77	0.07992	19.79	0.07888	45.10	0.08452	0.07853	0.07833	0.07826
Average		9.04		18.37		45.67				

Table 8.15 Results for Problem Set 14: 300-point problems originally in four dimensions with 10 solutions for DACN.

	DACN3		DACN4		DACN5		NLIM	COMB3	COMB4	COMB5
Problem	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Best solution	Best solution	Best solution
1	0.08658	25.05	0.08239	49.45	0.08130	106.48	0.08898	0.08095	0.08087	0.08075
2	0.08309	31.94	0.07867	54.39	0.07755	111.67	0.07923	0.07711	0.07704	0.07694
3	0.08340	26.77	0.07883	47.32	0.07772	105.38	0.08688	0.07729	0.07718	0.07707
4	0.08381	25.16	0.07923	45.56	0.07816	89.67	0.08037	0.07760	0.07756	0.07747
5	0.08233	23.61	0.07815	37.44	0.07723	91.50	0.08190	0.07691	0.07686	0.07677
6	0.08227	28.48	0.07780	68.92	0.07664	129.27	0.07938	0.07611	0.07609	0.07595
7	0.08105	30.01	0.07623	55.88	0.07506	100.38	0.08677	0.07442	0.07430	0.07429
8	0.08561	24.07	0.08093	38.16	0.07983	84.63	0.08543	0.07921	0.07919	0.07915
9	0.08294	27.12	0.07842	48.26	0.07727	122.60	0.08336	0.07682	0.07671	0.07662
10	0.08557	29.44	0.08114	47.19	0.08001	88.22	0.08364	0.07951	0.07942	0.07931
Average		27.17		49.26		102.98				

Table 8.16 Results for Problem Set 15: 500-point problems originally in four dimensions with 10 solutions for DACN.

	DACN3		DACN4		DACN5		NLIM	COMB3	COMB4	COMB5
Problem	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Best solution	Best solution	Best solution
1	0.09177	2.64	0.08834	5.07	0.08734	13.08	0.09525	0.08719	0.08713	0.08703
2	0.09033	2.27	0.08671	4.73	0.08572	13.07	0.08783	0.08566	0.08543	0.08538
3	0.09181	2.49	0.08759	5.16	0.08658	15.69	0.09199	0.08661	0.08619	0.08606
4	0.09071	2.22	0.08689	4.44	0.08583	11.64	0.09005	0.08548	0.08539	0.08531
5	0.08625	2.27	0.08206	4.74	0.08096	12.33	0.08199	0.08031	0.08008	0.08007
6	0.08538	2.28	0.08140	5.53	0.08048	16.94	0.09506	0.08027	0.08010	0.08005
7	0.08436	2.07	0.07996	4.56	0.07876	10.58	0.08963	0.07857	0.07847	0.07817
8	0.08427	2.14	0.08017	4.58	0.07919	12.07	0.08874	0.07917	0.07895	0.07886
9	0.08926	2.20	0.08528	4.45	0.08433	10.41	0.09860	0.08418	0.08411	0.08396
10	0.09376	2.28	0.08991	5.12	0.08903	12.90	0.09910	0.08892	0.08880	0.08830
Average		2.29		4.84		12.87				

Table 8.17 Results for Problem Set 16: 150-point problems originally in five dimensions with 10 solutions for DACN.

	DACN3		DACN4		DACN5		NLIM	COMB3	COMB4	COMB5
Problem	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Best solution	Best solution	Best solution
1	0.09594	13.06	0.09171	21.72	0.09058	45.77	0.09567	0.08968	0.08971	0.08964
2	0.09823	11.40	0.09445	22.77	0.09337	50.92	0.10082	0.09329	0.09303	0.09269
3	0.09494	14.91	0.09076	26.21	0.08981	53.32	0.10114	0.08965	0.08947	0.08943
4	0.09748	13.12	0.09353	21.47	0.09263	45.28	0.09693	0.09253	0.09231	0.09218
5	0.09391	12.31	0.08974	22.23	0.08867	52.13	0.08913	0.08811	0.08791	0.08784
6	0.09338	9.64	0.08935	16.57	0.08834	38.90	0.09426	0.08824	0.08793	0.08782
7	0.09255	10.91	0.08813	18.88	0.08703	42.28	0.09357	0.08653	0.08639	0.08633
8	0.09485	12.32	0.09075	19.16	0.08971	43.35	0.09457	0.08944	0.08938	0.08922
9	0.09408	11.88	0.08974	20.97	0.08864	42.19	0.10040	0.08829	0.08811	0.08801
10	0.09744	12.96	0.09342	21.23	0.09253	44.85	0.09867	0.09238	0.09225	0.09217
Average		12.25		21.12		45.90				

Table 8.18 Results for Problem Set 17: 300-point problems originally in five dimensions with 10 solutions for DACN.

	DACN3		DACN4		DACN5		NLIM	COMB3	COMB4	COMB5
Problem	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Best solution	Best solution	Best solution
1	0.10032	39.91	0.09656	58.83	0.09566	116.99	0.10238	0.09558	0.09547	0.09539
2	0.09924	40.01	0.09513	61.09	0.09416	151.90	0.09722	0.09408	0.09381	0.09364
3	0.09805	33.75	0.09409	58.69	0.09310	123.41	0.10623	0.09286	0.09271	0.09258
4	0.09999	34.97	0.09614	58.91	0.09523	120.31	0.09920	0.09512	0.09498	0.09488
5	0.09674	34.16	0.09283	52.69	0.09182	110.58	0.10135	0.09135	0.09129	0.09120
6	0.09587	33.98	0.09185	56.30	0.09091	109.63	0.10150	0.09064	0.09059	0.09045
7	0.09760	34.38	0.09334	56.97	0.09227	99.85	0.10401	0.09198	0.09172	0.09161
8	0.09977	36.45	0.09592	59.55	0.09492	122.30	0.10012	0.09462	0.09454	0.09442
9	0.09840	35.74	0.09427	64.90	0.09334	161.80	0.10032	0.09320	0.09306	0.09295
10	0.09790	36.58	0.09396	62.97	0.09294	122.93	0.10260	0.09277	0.09264	0.09247
Average		35.99		59.09		123.97				

Table 8.19 Results for Problem Set 18: 500-point problems originally in five dimensions with 10 solutions for DACN.

Next, we also applied DACN3, DACN4, DACN5, NLIM, and COMB to larger problems. We randomly generated problems of size 1000, 1250, 1500, 1750, 2000, 2250, and 2500 (one of each size) from a lattice set in three dimensions. For each problem, we solve it 10 times from 10 randomly generated starting solutions. In Table 8.20, we give the results for these problems. For all seven problems, DACN4 and DACN5 generate substantially better solutions (with respect to solution quality) than NLIM. The running times for DACN5 are nearly twice those of DACN4. The running times of DACN4 are roughly comparable to those of NLIM. However, as the number of points increases, the running time of NLIM increases more rapidly than that of DACN4. For the largest problem, NLIM required nearly twice as much time as DACN4. NLIM produced better solutions than DACN3, except for one problem. On the other hand, COMB3 clearly outperforms NLIM. COMB4 and COMB5 always outperform DACN5. As expected, COMB5 outperforms COMB4, which outperforms COMB3.

	DACN3		DACN4		DACN5		NLIM		COMB3		COMB4		COMB5	
Number of points	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Running time (secs)	Best solution	Running time (secs)
1000	0.06154	113.66	0.05650	160.04	0.05525	318.34	0.05954	180	0.05492	80	0.05468	50	0.05462	60
1250	0.06184	197.07	0.05703	342.55	0.05584	552.22	0.05920	230	0.05547	130	0.05533	100	0.05526	110
1500	0.06107	301.82	0.05620	466.36	0.05497	820.67	0.06227	410	0.05462	220	0.05444	140	0.05438	180
1750	0.06193	407.13	0.05704	573.25	0.05583	1009.59	0.06190	680	0.05547	495	0.05531	360	0.05524	410
2000	0.06210	494.62	0.05726	703.11	0.05607	1324.95	0.06171	1170	0.05564	830	0.05557	720	0.05551	770
2250	0.06237	720.02	0.05752	1079.74	0.05634	1856.91	0.06189	1620	0.05590	1120	0.05583	1170	0.05577	1210
2500	0.06257	811.88	0.05765	1439.35	0.05645	2921.63	0.06160	2655	0.05600	1930	0.05594	2110	0.05588	2020

Table 8.20 Results for larger problems originally in 3 dimensions with 10 replications.

8.3 Conclusions

Even though NLIM can assign points anywhere in the plane and DACN can only make assignments to specific points in the lattice space, DACN generates results that are comparable in quality to those generated by NLIM. DACN4 and DACN5 generate solutions that are better than those of NLIM. DACN5 gives better solutions than DACN4, but DACN5's running times are about twice as large. When the final solution from DACN is used as a starting solution in NLIM – this is the COMB heuristic – the final solutions generated by COMB4 and COMB5 are always better than the results generated by DACN and NLIM. COMB3 beats NLIM by a wide margin and requires a similar amount of computational effort. COMB5 generate the best solutions, but takes longer. The divide-and-conquer approach provides a good approximate solution. Furthermore, we have demonstrated that this solution is a good starting solution for the nonlinear method. Taking both solution quality and running time into account, we might recommend either DACN4 or DACN3 followed by COMB3 for solving large data visualization problems.

.

Chapter 9: Conclusions

9.1 Summary of Results

Data visualization applications are typically modeled and solved using nonlinear optimization techniques. In this dissertation, we proposed a discretization of the data visualization problem that allowed us to formulate it as a quadratic assignment problem. However, this formulation was computationally difficult to solve optimally with an exact approach. Consequently, we investigated the use of heuristics to solve our formulation. The space in which the data points are to be embedded was discretized using an *n* x *n* lattice. Conducting a local search on this *n* x *n* lattice was computationally inefficient. We proposed a divide-and-conquer approach that refined the lattice at each step. In Table 9.1 we give a description of the heuristic abbreviations used in this dissertation.

In Chapter 1, we gave an introduction and presented the objectives of the dissertation.

In Chapter 2, we gave an overview of the existing literature on data mining and data visualization. We presented background information on quadratic assignment problems, local search heuristics, integer programming problems, and genetic algorithms.

In Chapter 3, we presented the methodology that we used to formulate the data visualization problem as a quadratic assignment problem.

Heuristic	Description
LS	Local search heuristic
DAC	Divide-and-conquer local search heuristic
DACQ	Divide-and-conquer local search heuristic with quadrant restrictions
DACN	Divide-and-conquer local search heuristic with neighbor restrictions
IP	Integer program heuristic
IR	IP with Step 1 repeated
IRN	IR with points allowed to move to neighboring points
IRNS	IRN with a maximum of 20 points considered in reassigning points after Step 2 and 5 points reassigned at a time after Step 4
IMP	Improvement heuristic
HGA	Hybrid genetic algorithm

Table 9.1 Descriptions of heuristic abbreviations.

In Chapter 4, we developed a local search technique to solve our QAP and investigated four different discrete local search algorithms. Considering both solution quality and running time, DACN appeared to be the best heuristic, when compared to LS, DAC, and DACQ. DACN provided an approximate solution to the data visualization problem in a reasonable amount of computing time.

In Chapter 5, we gave an IP for the data visualization problem and proved it was equivalent to our QAP formulation. We developed a divide-and-conquer heuristic that solved a set of smaller problems at each stage instead of one large problem and showed that it gave a more manageable problem to solve at each stage. We found that DACN was more accurate and efficient than IRNS. DACN produced much better objective function values than IRNS. The running times for DACN were also much shorter than those of IRNS. In addition, we observed that that the local search procedure of assigning points one at a time worked well for the data visualization problem. Even though DACN produced solutions by reassigning points, one point at a time, IMP, which reassigns five points at a time, did not improve the results of DACN.

In Chapter 6, we developed a hybrid heuristic (HGA) that combined local search with a genetic algorithm by applying genetic algorithms techniques to the final solution from DACN. We found that HGA improved the solutions produced by DACN. However, the improvements were very small. In addition, the running times of HGA were longer than those of DACN. We recommended that, when our heuristic is used as a stand-alone approach or a very accurate solution is required, HGA should be used rather than DACN. On the other hand, if our heuristic is used to produce a starting solution for a nonlinear method or an approximate solution is required, then DACN should be used rather than HGA for faster computation times.

In Chapter 7, we compared the results of DACN to the results generated by a nonlinear Sammon map (NLSM). DACN can only make assignments to specific points in the lattice space. NLSM can assign points anywhere in the plane. Despite this limitation, DACN generated results that were comparable in quality to those generated by NLSM. When the final solution from DACN was used as a starting solution in NLSM – that is the COMB heuristic – the final solutions generated by COMB were always better than the results generated by DACN and NLSM. The running times for COMB were much smaller than those for NLSM and were, in fact, quite reasonable.

In Chapter 8, we compared the results of DACN to the results generated by a commercial nonlinear multidimensional scaling map (NLIM). We used the majorization technique as implemented in SPSS Proxscal. We solved DACN with 10 solutions instead of 100 solutions used in our other experiments. We applied DACN, NLIM, and COMB to larger problem sets, ranging in size from 1000 to 2500. DACN4 and DACN5 generated solutions that were better than those of NLIM. DACN5 gave better solutions than DACN4, but DACN5's running times were about twice as large. COMB always produced better results than those generated by DACN and NLIM. COMB3 beat NLIM by a wide margin and required a similar amount of computational effort. COMB5 generated the best solutions, but took longer. Taking both solution quality and running time into account, we recommend either DACN4 or DACN3 followed by COMB3 for solving large data visualization problems.

We summarize the research contributions of this dissertation as follows:

- The data visualization problem can be formulated as a QAP.
- We demonstrated that discrete optimization and a divide-and-conquer local search heuristic can be applied to continuous optimization problems arising in data visualization.
- DACN can only make assignments to specific points in the lattice space but NLSM and NLIM can assign points anywhere in the plane. Despite this limitation, DACN generated results that are comparable in quality to those generated by NLSM and NLIM (and superior for large problems).
- When the final solution from DACN is used as a starting solution in NLSM and NLIM – this is the COMB heuristic – the final solutions generated by COMB are always better than the results generated by DACN, NLSM, and NLIM.

- The running times for COMB (alone) are much smaller than those for NLSM and NLIM, and are, in fact, quite reasonable.
- The divide-and-conquer approach provides a good approximate solution in a small amount of computing time. Furthermore, we have shown that this solution is a good starting solution for the nonlinear method.
- Taking both solution quality and running time into account, we recommend either using DACN4 or DACN3 followed by COMB3 for solving large data visualization problems.

9.2 Future Research

There are several opportunities for future work on this topic. For example, in this dissertation, the lattice structure was made uniform over the entire rectangular grid. This need not be so. Where there is a higher density of points, the lattice can be made finer. Where there are fewer points, the lattice can be made coarser.

Different lattice structures can also be considered. For instance, we can investigate keeping the previous lattice point and considering five lattice points at each stage, instead of only considering the four new lattice points. It may be that the point was best assigned to the previous lattice point and not to one of the four new lattice points.

There is also an opportunity to investigate other well-known heuristics like simulated annealing and tabu search to see if they can provide good solutions to the QAP formulation for the data visualization problem.

In this dissertation, we used randomly generated data sets from a uniformly distributed data set. We can consider other data sets. For instance, we can generate a two-dimensional data set either randomly or using a known function. We can then transform this data into a higher dimensional data set. We can do this, for example, by finding linear combinations of the coordinates in two-dimensions and adding some noise. The noise should have a small variance so that the original structure in the data is not destroyed. With this type of data we can compare the final solution obtained with the original solution.

Glossary

LS	-	Local Search heuristic
DAC	-	Divide-and-conquer local search heuristic
DACQ	-	Divide-and-conquer local search heuristic with quadrant restrictions
DACN	-	Divide-and-conquer local search heuristic with neighbor restrictions
IP	-	Integer program heuristic
IR	-	IP with Step 1 repeated
IRN	-	IR with points allowed to move to neighboring lattice points
IRNS	-	IRN with a maximum of 20 points considered in reassigning points after Step 2 and 5 points reassigned at a time after Step 4
IMP	-	Improvement heuristic
HGA	-	Hybrid genetic algorithm

Appendix A

Source code for DACN and HGA

```
#include "ModelOne.h"
#include "ModelFive.h"
#include "RecipeRandGenImpl.h"
#include "ConstraintKnutStrRandomGenerator.h"
#include "ODMatrixGen.h"
#include "ObjectiveValueCalculator.h"
#include "CommandLineInterpreter.h"

RecipeRandGenImpl gRanGen(-1L);
ConstraintKnutStrRandomGenerator gKnutRanGen(gRanGen);

int main(int argc, char** argv)
{
	try {
		CommandLineInterpreter cmdl;
		cmdl.Parse(argc, argv);

		ODMatrixGen odMatGen(cmdl.PointsMatrixFile().c_str());

		ModelFive<ConstraintKnutStrRandomGenerator, IntMatrix, ObjValueCalculator>
				gModelFive(gKnutRanGen, odMatGen,
(cmdl.OutputFileName()).c_str(),
				cmdl.NoLatticePoints(), cmdl.NoPoints(), cmdl.SampleSize(),
				(cmdl.PointsMatrixFile()).c_str());

		gModelFive.Model();
	}
	catch(...)
	{
		fprintf(stderr, "An exception was thrown; check your input parameters");
		exit(-1);
	}

	return 0;
}
```

```
#include "CommandLineInterpreter.h"
#include <cstdio>

void CommandLineInterpreter::Parse(int argc, char** argv)
{

    if(argc>1)
    {
        for(int index = 1; index < argc; ++index)
       {
            if(strlen(argv[index])==2 && (argv[index][0]=='-' || argv[index][0]=='/'))
            {
                char key= argv[index][1];
                key = toupper(key);
                switch(key)
                {
                case 'N':
                    lattice_ = atoi(argv[++index]);
                    break;
                case 'M':
                    points_ = atoi(argv[++index]);
                    break;
                case 'S':
                    sampleSize_ = atoi(argv[++index]);
                    break;
                case 'O':
                    outputFileName_ = argv[++index];
                    break;
                case 'I':
                    pointsMatrixFile_ = argv[++index];
                    break;
                default:
                    fprintf(stderr, "Invalid usage\n");
                    fprintf(stderr, "Usage: localsearch [-n # lattice point ] [-m
# points] "

                              "[-s sample size] [-o output file] [-i input point

matrix file]\n");
                    break;
                    exit(-1);
                }
            }
            else if(strlen(argv[index]) > 2 && (argv[index][0] == '-' || argv[index][0] ==
'/'))
            {
                char key  = argv[index][1];
                key = toupper(key);
                char buffer[512];
                strcpy(buffer, argv[index] + 2);
                switch(key)
                {
                case 'N':

                    lattice_ = atoi(buffer);
                    break;
                case 'M':
                    points_ = atoi(buffer);
                    break;
                case 'S':
                    sampleSize_ = atoi(buffer);
                    index++;
                    break;
                case 'O':
                    outputFileName_ = buffer;
                    break;
                case 'I':
                    pointsMatrixFile_ = buffer;
                    break;
                default:
                    fprintf(stderr, "Invalid usage\n");
                    fprintf(stderr, "Usage: localsearch [-n # lattice point ] [-m
# points] "
                        "[-s sample size] [-o output file] [-i input point
matrix file]\n");
                    break;
                    exit(-1);
                }
            }
            else
            {
                printf("Invalid usage\n");
                printf("Usage: localsearch [-n # lattice point ] [-m # points] "
                        "[-s sample size] [-o output file] [-i input point
matrix file]\n");
                exit(-1);
            }
      }
    }
    else
    {
        fprintf(stderr, "WARNING: using the following default parameters: \n\tNo lattice
points = %d"
                "\n\tNo of points = %d\n\tSample size = %d\n\tOutput file name =
%s\n\t"
                "No input matirx file using,\n\t\t xcoorm[M] = {1, 3, 1, 3,   9, 11,
9, 11,   25, 27, 25, 27,   29, 31, 29, 31,    13, 15, 13, 15}"
                "\n\t\tycoorm[M] = {31, 31, 29, 29,  29, 29,  31, 31, 3, 3, 1, 1,    3,
3, 1, 1, 3, 3, 1, 1};",
                lattice_, points_, sampleSize_, outputFileName_.c_str());
    }

}
```

```
#include "ConstraintKnutStrRandomGenerator.h"

#include <ctime>
#include <list>


void ConstraintKnutStrRandomGenerator::generate(std::vector<std::string>& ranVec,
                                                                int howMany, int size, int
lowerValue, int upperValue)
{
    // we assume that the upperValue  will not be more than 999
    if(!ranVec.empty())
        ranVec.clear();

    const BUF = 20;
    char element[BUF];
    int bufLen = 3 * (size + 1);
    char* pItem = new char[bufLen];
    std::list<int> positions;

    for (int cnt = 0; cnt < howMany; ++cnt)
    {

        if(!positions.empty()) positions.clear();
        for(int pos = 0; pos < size; ++pos)
        {
            // put positions here to be allocated
            positions.push_back(pos);
        }


        for(int cnt = 0; cnt < bufLen; ++cnt)
        {
            pItem[cnt] = 'x';
        }
        pItem[2] = '1'; // position 1 should always be 1
                // terminate the string
        pItem[bufLen - 1 ] = '\0';
        pItem[bufLen - 2 ] = '\0';
        pItem[bufLen - 3 ] = '\0';

        positions.remove(0);

        // assign position for lattice point 2
        int pointElem2 = 1 + static_cast<int>(( (size - 2) * ran() + 0.1));
        pItem[3 * pointElem2 + 2] = '2';

        // remove from positions
        positions.remove(pointElem2);

        for(int i = 2; i < size; ++i)
        {
            int pointElem;
            int latticeElem = lowerValue +
                static_cast<int>(( (upperValue - lowerValue) * ran() + 0.1));
            if(!positions.empty())
            {
                bool found = true;
                do
                {
                    found = true;
                    pointElem = positions.front();
                    positions.pop_front();
                    if(latticeElem == 3 && pointElem == 1)
                    { // ensure that pt 2 is not assigned to latice point 3
                        positions.push_back(pointElem);
                        if(positions.size() == 1)
                        {
                            latticeElem = lowerValue +
                                static_cast<int>(( (upperValue
- lowerValue) * ran() + 0.1));
                        }
                        found = false;
                    }
                } while(!found);

                memset(element,0,sizeof(char) * BUF);
                _itoa( latticeElem, element, 10);
                int len = strlen(element);

                switch(len)
                {
                case 1:
                    pItem[3 * pointElem + 2] = element[0];
                    break;
                case 2:
                    pItem[3 * pointElem + 1] = element[0];
                    pItem[3 * pointElem + 2] = element[1];
                    break;
                case 3:
                    pItem[3 * pointElem] = element[0];
                    pItem[3 * pointElem + 1] = element[1];
                    pItem[3 * pointElem + 2] = element[2];
                default:
                    fprintf(stderr, "Error latticeElement %d is greater than
999", latticeElem);
                    exit(-1);

                } // switch

            }
        } // for
        ranVec.push_back(pItem);
    } // for

    if(pItem) delete [] pItem;

} // ConstraintKnutStrRandomGenerator::generate
#include "KnutIntRandomGenerator.h"

void KnutIntRandomGenerator::generate(std::vector<int>& ranVec,
                                                        int howMany, int size, int
lowerValue, int upperValue)
{
        if(!ranVec.empty())
                ranVec.clear();

        const BUF = 20;
        char element[BUF];
        for (int cnt = 0; cnt < howMany; ++cnt)
        {

                for(int i = 0; i < size; ++i)
                {
                        int elem = lowerValue +
                                static_cast<int>(( (upperValue - lowerValue) * ran() + 0.1));
                        memset(element,0,sizeof(char) * BUF);
                        ranVec.push_back(elem);
                }
        }

} // KnutIntRandomGenerator::generate
```

```
#include "KnutStrRandomGenerator.h"
#include <ctime>


void KnutStrRandomGenerator::generate(std::vector<std::string>& ranVec,
                                                                int howMany, int size, int
lowerValue, int upperValue)
{
        if(!ranVec.empty())
                ranVec.clear();

        const BUF = 20;
        const char SEPERATOR = 'x';
        char element[BUF];
        for (int cnt = 0; cnt < howMany; ++cnt)
        {
                std::string item;
                for(int i = 0; i < size; ++i)
                {
                        int elem = lowerValue +
                                static_cast<int>(( (upperValue - lowerValue) * ran() + 0.1));
                        memset(element,0,sizeof(char) * BUF);
                        _itoa( elem, element, 10);

                        // insert x as a seperator between the numbers
                        item += SEPERATOR;
                        item += element;
                }
                ranVec.push_back(item);
        }

} // KnutStrRandomGenerator::generate
```

```
#include "LocalSearchModeler.h"

void LocalSearchModeler::Model()
{
        GenerateInitialSolution();
        PerformLocalSearch();
        CalculateObjectiveValue();
        //GenerateInitialSolution();
```

```
#include "NDMatrixGen.h"
#include <iterator>
#include <math.h>

//n= 4
int xcoorn4 [] = {8, 24, 8, 24};
int ycoorn4 [] = {24, 24, 8, 8};


//n = 16
static int xcoorn16 [] = { 4, 12,  4, 12,  20, 28, 20, 28,   4, 12, 4, 12,  20, 28, 20, 28};
static int ycoorn16 [] = {28, 28, 20, 20,  28, 28, 20, 20,  12, 12, 4,  4,  12, 12,  4,  4};

//n = 64
static int xcoorn64 [] = {2,  6,  2,  6,   10, 14, 10, 14,   2, 6, 2, 6,  10, 14, 10, 14,  18, 22, 18, 22,  26, 30,
26, 30,  18, 22, 18, 22,  26, 30, 26, 30,   2, 6, 2, 6,  10, 14, 10, 14,  2, 6, 2, 6,  10, 14, 10, 14,   18, 22,
18, 22,  26, 30, 26, 30,  18, 22, 18, 22,   26, 30, 26, 30};
static int ycoorn64 [] = {30, 30, 26, 26,  30, 30, 26, 26,  22, 22, 18, 18,   22, 22, 18, 18,  30, 30, 26, 26,
30, 30, 26, 26,  22, 22, 18, 18,  22, 22, 18, 18,  14, 14, 10, 10,  14, 14, 10, 10,  6, 6, 2, 2,  6, 6, 2, 2,  14,
14, 10, 10,  14, 14, 10, 10,  6, 6, 2, 2,   6, 6, 2, 2};

//n = 256
static int xcoorn256 [] = {1, 3, 1, 3,   5, 7, 5, 7,  1, 3, 1, 3,   5, 7, 5, 7,   9, 11, 9, 11,   13, 15, 13, 15,   9,
11, 9, 11,   13, 15, 13, 15,  1, 3, 1, 3,   5, 7, 5, 7,  1, 3, 1, 3,   5, 7, 5, 7,   9, 11, 9, 11,   13, 15, 13, 15,   9,
11, 9, 11,   13, 15, 13, 15,
        17, 19, 17, 19,  21, 23, 21, 23,   17, 19, 17, 19,   21, 23, 21, 23,   25, 27, 25, 27,   29, 31, 29, 31,
25, 27, 25, 27,   29, 31, 29, 31,   17, 19, 17, 19,  21, 23, 21, 23,   17, 19, 17, 19,   21, 23, 21, 23,   25,
27, 25, 27,   29, 31, 29, 31,   25, 27, 25, 27,   29, 31, 29, 31,
        1, 3, 1, 3,   5, 7, 5, 7,  1, 3, 1, 3,   5, 7, 5, 7,   9, 11, 9, 11,   13, 15, 13, 15,   9, 11, 9, 11,   13, 15,
13, 15,  1, 3, 1, 3,   5, 7, 5, 7,  1, 3, 1, 3,   5, 7, 5, 7,   9, 11, 9, 11,   13, 15, 13, 15,   9, 11, 9, 11,   13, 15,
13, 15,
        17, 19, 17, 19,  21, 23, 21, 23,   17, 19, 17, 19,   21, 23, 21, 23,   25, 27, 25, 27,   29, 31, 29, 31,
25, 27, 25, 27,   29, 31, 29, 31,   17, 19, 17, 19,  21, 23, 21, 23,   17, 19, 17, 19,   21, 23, 21, 23,   25,
27, 25, 27,   29, 31, 29, 31,   25, 27, 25, 27,   29, 31, 29, 31};

static int ycoorn256 [] = {31, 31, 29, 29,  31, 31, 29, 29,   27, 27, 25, 25,   27, 27, 25, 25,  31, 31, 29,
29,  31, 31, 29, 29,   27, 27, 25, 25,   27, 27, 25, 25,   23, 23, 21, 21,   23, 23, 21, 21,  19, 19, 17, 17,
19, 19, 17, 17,    23, 23, 21, 21,   23, 23, 21, 21,  19, 19, 17, 17,   19, 19, 17, 17,
        31, 31, 29, 29,  31, 31, 29, 29,   27, 27, 25, 25,   27, 27, 25, 25,  31, 31, 29, 29,  31, 31, 29, 29,
27, 27, 25, 25,   27, 27, 25, 25,   23, 23, 21, 21,   23, 23, 21, 21,  19, 19, 17, 17,   19, 19, 17, 17,    23,
23, 21, 21,   23, 23, 21, 21,  19, 19, 17, 17,   19, 19, 17, 17,
        15, 15, 13, 13,   15, 15, 13, 13,   11, 11, 9, 9,  11, 11, 9, 9,  15, 15, 13, 13,   15, 15, 13, 13,   11,
11, 9, 9,  11, 11, 9, 9,   7, 7, 5, 5,    7, 7, 5, 5,   3, 3, 1, 1,    3, 3, 1, 1,   7, 7, 5, 5,    7, 7, 5, 5,   3, 3, 1, 1,
3, 3, 1, 1,
        15, 15, 13, 13,  15, 15, 13, 13,   11, 11, 9, 9,  11, 11, 9, 9,  15, 15, 13, 13,   15, 15, 13, 13,   11,
11, 9, 9,  11, 11, 9, 9,   7, 7, 5, 5,    7, 7, 5, 5,   3, 3, 1, 1,    3, 3, 1, 1,   7, 7, 5, 5,    7, 7, 5, 5,   3, 3, 1, 1,
3, 3, 1, 1};


NDMatrixGen::NDMatrixGen() : type_(0)
{

            matrixContainer_.insert(IntPointerPairMap::value_type(4, IntPointerPair(xcoorn4,
ycoorn4)));
            matrixContainer_.insert(IntPointerPairMap::value_type(16, IntPointerPair(xcoorn16,
ycoorn16)));
```

```
    matrixContainer_.insert(IntPointerPairMap::value_type(64, IntPointerPair(xcoorn64,
ycoorn64)));
    matrixContainer_.insert(IntPointerPairMap::value_type(256, IntPointerPair(xcoorn256,
ycoorn256)));


} // NDMatrixGen::NDMatrixGen

void NDMatrixGen::Generate(int size)
{


    if(type_ && type_ == size)
        return;

    if(!(size == 4 || size == 16 || size == 64 || size == 256))
    { // wrong input so wipe out everything from the matrix because
      // one would be tempted to use it
        type_ = 0;
        ndMatrix_.clear();
    }
    else
    {
        if(!ndMatrix_.empty()) ndMatrix_.clear();
        int value = 0;
        IntPointerPairMap::iterator iter = matrixContainer_.find(size);
        if(iter != matrixContainer_.end())
        {
            IntPointerPair vectorPair = (*iter).second;
            for (int i = 0; i < size; ++i)
            {
                IntVec vec;
                ndMatrix_.push_back(vec);
                for (int j = 0; j < size; ++j)
                {
                    value = ((vectorPair.first)[i] - (vectorPair.first)[j]) *
                        ((vectorPair.first)[i] - (vectorPair.first)[j]) -
                   ((vectorPair.second)[i] - (vectorPair.second)[j]) *
                        ((vectorPair.second)[i] - (vectorPair.second)[j]);
                    if(value > 0) value = sqrt((double)value);
                    ndMatrix_[i].push_back(value);

                } // for
            } // for
        }


    }

}
```

```
#include "ObjectiveValueCalculator.h"
#include <cassert>
#include <map>
#include <algorithm>


bool long_int_cmp (const LNGINTPAIR& first, const LNGINTPAIR& second )
{
	return (first.first < second.first) ? true : false;

}

void ObjValueCalculator::ExtractValues(const OBJVString& inStr, OBJIntVec& outVec)
{
	assert(inStr.size() % 3 == 0); // ensure that string is valid

	if(inStr.size() % 3 != 0)
		throw 1;

	if(!outVec.empty()) outVec.clear();

	// extract values form string
	char buffer[4];

	int size = inStr.size();
	char* ptr = const_cast<char*>(inStr.c_str());
	int index = 0;
	for(;  index < size && ptr != '\0'  && ptr ; ptr += 3, index += 3) // 3 = # representing a
position
	{
		memset(buffer, 0, 4 * sizeof(char));
		strncpy(buffer, ptr, 3);
		OBJVString digit;
		for(int i = 0; buffer[i] != 0; ++i)
		{
			if(isdigit(buffer[i]))
			{
				digit += buffer[i];
			}
		} // for

		if(!digit.empty())
		{
			outVec.push_back(atoi(digit.c_str()));
		}

	} // for

} // ObjValueCalculator::ExtractValues


void ObjValueCalculator::CalculatObjectiveValue
		(
			unsigned int pos,
			const OBJVString& str,
			IntMatrix& odMat,
            IntMatrix& ndMat,
            LNINTPAIRVEC& objectiveFunctionList

        )
{

    OBJIntVec strVec;
    ExtractValues(str, strVec);

    if(!objectiveFunctionList.empty())
            objectiveFunctionList.clear();

    // we are interested in the following summation:
    // we have a vector C =(c[l] l = 0, strVec.size() - 1
    // for each value of i (ie pos),
    // we calculate sum over k( sum over j(OD[i,j] - ND[k, c[j]])), j>i,
    // k = 0,...,ndMat.size() - 1

    unsigned int strVecSize = strVec.size();

    unsigned int ndSize = ndMat.size(); // to be removed

    for(int k = 0; k < ndMat.size(); ++k)
    {
        LNGINTPAIR sum(0, k);
        for(int j = pos + 1; j < strVec.size(); ++j)
        {

            long value = (odMat[pos])[j] - (ndMat[k])[strVec[j]];
            value *= value;
            sum.first += value;
        }
        objectiveFunctionList.push_back(sum);
    }

#if 0
        // test output
    for(LNINTPAIRVEC::iterator iter = objectiveFunctionList.begin();
        iter != objectiveFunctionList.end(); ++iter)
        {
            printf("value[%d] = %d\n", (*iter).second, (*iter).first);
        }

#endif

} // ObjValueCalculator::CalculatObjectiveValue





// calculates the objective value for the positon
// pos
LNGINTPAIR ObjValueCalculator::CalculatObjectiveValue
        (
            unsigned int pos,
                        const OBJVString& str,
                        IntMatrix& odMat,
                        IntMatrix& ndMat
                )
{

        LNINTPAIRVEC objectiveFunctionList;
        CalculatObjectiveValue(pos, str, odMat, ndMat, objectiveFunctionList);

        assert(!objectiveFunctionList.empty()); // check to make sure that we do not get empty vector

        LNGINTPAIR minValue(*std::min_element(objectiveFunctionList.begin(),
                objectiveFunctionList.end(), long_int_cmp));

        // since int hte calulateObjective function we started k from 0 we have
        // to increase its value by one to account for the actual value

        minValue.second++;

        return minValue;



        /*
        OBJIntVec strVec;
        ExtractValues(str, strVec);

        LNINTPAIRVEC objectiveFunctionList;

        // we are interested in the following summation:
        // we have a vector C =(c[l] l = 0, strVec.size() - 1
        // for each value of i (ie pos),
        // we calculate sum over k( sum over j(OD[i,j] - ND[k, c[j]])), j>i,
        // k = 0,...,ndMat.size() - 1

        unsigned int strVecSize = strVec.size();

        unsigned int ndSize = ndMat.size(); // to be removed

        for(int k = 0; k < ndMat.size(); ++k)
        {
                LNGINTPAIR sum(0, k);
                for(int j = pos + 1; j < strVec.size(); ++j)
                {

                        long value = (odMat[pos])[j] - (ndMat[k])[strVec[j]];
                        value *= value;
                        sum.first += value;
                }
                objectiveFunctionList.push_back(sum);
        }



        assert(!objectiveFunctionList.empty()); // check to make sure that we do not get empty vector
                LNGINTPAIR minValue(*std::min_element(objectiveFunctionList.begin(),
                objectiveFunctionList.end(), long_int_cmp));

#if 0
                // test output
        for(LNINTPAIRVEC::iterator iter = objectiveFunctionList.begin();
                iter != objectiveFunctionList.end(); ++iter)
                {
                        printf("value[%d] = %d\n", (*iter).second, (*iter).first);
                }

#endif

                // since we started k from 0 we have to increase its value
                // by one to account for the actual value
                minValue.second++;

        return minValue;
        */

} // ObjValueCalculator::CalculatObjectiveValue


LNGINTPAIR ObjValueCalculator::CalculatObjectiveValue
                (
                        unsigned int pos,
                        const OBJVString& str,
                        IntMatrix& odMat,
                        IntMatrix& ndMat,
                        INTSET& relLPts

                )
{

        LNINTPAIRVEC objectiveFunctionList;
        CalculatObjectiveValue(pos, str, odMat, ndMat, objectiveFunctionList);

        int size = objectiveFunctionList.size();
        LNINTPAIRVEC modifiedObjectiveFunctionList;
        INTSET::iterator setIterEnd = relLPts.end();

        for(int i=0; i < size; ++i)
        {
                if(relLPts.find(objectiveFunctionList[i].second) != setIterEnd)
                {
                        modifiedObjectiveFunctionList.push_back(objectiveFunctionList[i]);
                }
        }

        assert(!modifiedObjectiveFunctionList.empty()); // check to make sure that we do not get
empty vector

        LNGINTPAIR minValue(*std::min_element(modifiedObjectiveFunctionList.begin(),
                modifiedObjectiveFunctionList.end(), long_int_cmp));

        // since in the calulateObjective function we started k from 0, we have
```

```
        // to increase its value by one to account for the actual value

        minValue.second++;

        return minValue;

} // ObjValueCalculator::CalculatObjectiveValue


// calculates the final objective value for the string
long ObjValueCalculator::CalculatObjectiveValue
(
        const OBJVString& str,
        IntMatrix& odMat,
        IntMatrix& ndMat
)
{

        OBJIntVec strIntVec;
        ExtractValues(str, strIntVec);

        LNGVEC objectiveFunctionList;

        unsigned int strVecSize = strIntVec.size();
        long sum = 0;

        for(int i = 0; i < strVecSize; ++i)
        {
                for(int j = i + 1; j < strVecSize; ++j)
                {
                        int k = strIntVec[i] - 1; // -1 to reflect the fact
                        int l = strIntVec[j] - 1; // that we start from zero

                        long value = (odMat[i])[j] - (ndMat[k])[l];
                        value *= value;
                        sum += value;
                }
        }

        return sum;
} // ObjValueCalculator::CalculatObjectiveValue
```

```
#include "ODMatrixGen.h"
#include <iterator>
#include <math.h>
#include <fstream>


ODMatrixGen::ODMatrixGen(const char* fileName) : generated_(false)
{
	if(fileName)
	{
		if(strcmp(fileName, "") == 0)
		{
			Generate();
		}
		else
		{
			fileName_ = fileName;
			Generate(true);
		}
	}
	else
	{
		fileName_ = "";
		Generate();
	}
}

ODMatrixGen::ODMatrixGen(std::string& fileName) : generated_(false)
{
	if(!fileName.empty())
	{
		fileName_ = fileName;
		Generate(true);
	}
	else
	{
		fileName_ = "";
		Generate();
	}
}



void ODMatrixGen::Generate()
{

	int xcoorm[] = {1, 3, 1, 3,   9, 11, 9, 11,   25, 27, 25, 27,   29, 31, 29, 31,    13, 15, 13, 15};
	int ycoorm[] = {31, 31, 29, 29,  29, 29,  31, 31, 3, 3, 1, 1,    3, 3, 1, 1, 3, 3, 1, 1};
	int size = sizeof(xcoorm) / sizeof(&xcoorm[0]);


	int value = 0;
	for (int i = 0; i < size; ++i)
	{
		IntVec vec;
		odMatrix_.push_back(vec);
```

```
                for (int j = 0; j < size; ++j)
                {
                        value = (xcoorm[i] - xcoorm[j]) * (xcoorm[i] - xcoorm[j]) -
                                  (ycoorm[i] - ycoorm[j]) * (ycoorm[i] - ycoorm[j]);
                        if(value > 0) value = sqrt((double)value);
                        odMatrix_[i].push_back(value);

                } // for
        } // for

        generated_ = true;

}


void ODMatrixGen::Generate(bool bFromFile)
{
        if(bFromFile)
        {
                std::ifstream fileStream(fileName_.c_str());
                if(!fileStream.is_open())
                {
                        fileStream.open(fileName_.c_str());
                }

                if(!fileStream.is_open())
                        throw 1;

//              fileStream.setmode(filebuf::text);

                int xcoord;
                int ycoord;

                IntVec xcoordVec;
                IntVec ycoordVec;

                while(!fileStream.eof())
                {
                        fileStream >> xcoord;
                        xcoordVec.push_back(xcoord);

                        fileStream >> ycoord;
                        ycoordVec.push_back(ycoord);
                }

                fileStream.close();

                if(xcoordVec.empty() || ycoordVec.empty()) throw 1;
                if(xcoordVec.size() != ycoordVec.size()) throw 1;

                //TDB

                int value = 0;
                int size = xcoordVec.size();
                for (int i = 0; i < size; ++i)
                {
```

```
            IntVec vec;
            odMatrix_.push_back(vec);
            for (int j = 0; j < size; ++j)
            {
                value = (xcoordVec[i] - xcoordVec[j]) * (xcoordVec[i] -
xcoordVec[j]) -
                    (ycoordVec[i] - ycoordVec[j]) * (ycoordVec[i] -
ycoordVec[j]);
                if(value > 0) value = sqrt((double)value);
                odMatrix_[i].push_back(value);

            } // for
        } // for

        generated_ = true;

    }
    else
    {
        Generate();
    }

}
```

```
#include "RandGenImpl.h"

#define MBIG 1000000000
#define MSEED 161803398
#define MZ 0
#define FAC (1.0/MBIG)


float RandGenImpl::ran(long *idum)
{
        static int inext,inextp;
        static long ma[56];
        static int iff=0;
        long mj,mk;
        int i,ii,k;

        if (*idum < 0 || iff == 0) {
                iff=1;
                mj=MSEED-(*idum < 0 ? -*idum : *idum);
                mj %= MBIG;
                ma[55]=mj;
                mk=1;
                for (i=1;i<=54;i++) {
                        ii=(21*i) % 55;
                        ma[ii]=mk;
                        mk=mj-mk;
                        if (mk < MZ) mk += MBIG;
                        mj=ma[ii];
                }
                for (k=1;k<=4;k++)
                        for (i=1;i<=55;i++) {
                                ma[i] -= ma[1+(i+30) % 55];
                                if (ma[i] < MZ) ma[i] += MBIG;
                        }
                inext=0;
                inextp=31;
                *idum=1;
        }
        if (++inext == 56) inext=1;
        if (++inextp == 56) inextp=1;
        mj=ma[inext]-ma[inextp];
        if (mj < MZ) mj += MBIG;
        ma[inext]=mj;
        return mj*FAC;
}  // RandGenImpl::ran

#undef MBIG
#undef MSEED
#undef MZ
#undef FAC
/* (C) Copr. 1986-92 Numerical Recipes Software $2'9M)!].)!-01a. */
```

```
#include "RestrictionsAllocator.h"


RestrictionsAllocator::~RestrictionsAllocator()
{
	if(!restrictors_.empty())
	{
		RESTRICTTORMAP::iterator iter = restrictors_.begin();
		RESTRICTTORMAP::iterator iterEnd = restrictors_.end();
		for(; iter != iterEnd; ++iter)
		{
			delete (*iter).second;
			(*iter).second = 0;
		}
	}
}


INTSET& RestrictionsAllocator::FindSet(int size)
{
	if( size > 64) throw 1; //only for lattice point up to 64

	if(restrictors_.empty()) return Allocate(size);

	RESTRICTTORMAP::iterator iter = restrictors_.find(size);
	if(iter == restrictors_.end())
	{
		return Allocate(size);
	}

	return *((*iter).second);
}


// filter
void RestrictionsAllocator::Filter(INTSET& object)
{
	INTSET::iterator iter = object.begin();
	INTSET::iterator iterEnd = object.end();
	IntVec buffer;
	for(; iter != iterEnd; ++iter)
	{
		if(*iter > maxLatticeValue_)
		{
			buffer.push_back(*iter);
		}
	}

	if(!buffer.empty())
	{
		int bufSize = buffer.size();
		for(int i = 0; i < bufSize; ++i)
		{
			object.erase(buffer[i]);
		}
	}
```

```
}


INTSET& RestrictionsAllocator::Allocate(int size)
{

        INTSET* value = new INTSET;
        if(!value) throw 1;

        value->insert(size); // make sure that the point is part of it

        switch(size)
        {
        case 1:
                {
                        value->insert(2);
                        value->insert(3);
                        value->insert(4);
                }
                break;
        case 2:
                {
                        value->insert(1);
                        value->insert(3);
                        value->insert(4);
                        value->insert(5);
                        value->insert(7);
                }
                break;
        case 3:
                {
                        value->insert(1);
                        value->insert(2);
                        value->insert(4);
                        value->insert(9);
                        value->insert(10);
                }
                break;
        case 4:
                {
                        value->insert(1);
                        value->insert(2);
                        value->insert(5);
                        value->insert(3);
                        value->insert(7);
                        value->insert(9);
                        value->insert(10);
                        value->insert(13);
                }
                break;
        case 5:
                {
                        value->insert(2);
                        value->insert(6);
                        value->insert(4);
                        value->insert(7);
                        value->insert(8);
                }
                break;
        case 6:
                {
                        value->insert(5);
                        value->insert(17);
                        value->insert(7);
                        value->insert(8);
                        value->insert(19);
                }
                break;
        case 7:
                {
                        value->insert(2);
                        value->insert(5);
                        value->insert(6);
                        value->insert(4);
                        value->insert(8);
                        value->insert(10);
                        value->insert(13);
                        value->insert(14);
                }
                break;
        case 8:
                {
                        value->insert(5);
                        value->insert(6);
                        value->insert(17);
                        value->insert(7);
                        value->insert(19);
                        value->insert(13);
                        value->insert(14);
                        value->insert(25);
                }
                break;
        case 9:
                {
                        value->insert(3);
                        value->insert(4);
                        value->insert(10);
                        value->insert(11);
                        value->insert(12);
                }
                break;
        case 10:
                {
                        value->insert(3);
                        value->insert(4);
                        value->insert(7);
                        value->insert(9);
                        value->insert(13);
                        value->insert(11);
                        value->insert(12);
                        value->insert(15);
                }
                break;
        case 11:
                {
                        value->insert(9);
                        value->insert(10);
                        value->insert(12);
                        value->insert(33);
                        value->insert(34);
                }
                break;
        case 12:
                {
                        value->insert(9);
                        value->insert(10);
                        value->insert(13);
                        value->insert(11);
                        value->insert(15);
                        value->insert(33);
                        value->insert(34);
                        value->insert(37);
                }
                break;
        case 13:
                {
                        value->insert(4);
                        value->insert(7);
                        value->insert(8);
                        value->insert(10);
                        value->insert(14);
                        value->insert(12);
                        value->insert(15);
                        value->insert(16);
                }
                break;
        case 14:
                {
                        value->insert(7);
                        value->insert(8);
                        value->insert(19);
                        value->insert(13);
                        value->insert(25);
                        value->insert(15);
                        value->insert(16);
                        value->insert(27);
                }
                break;
        case 15:
                {
                        value->insert(10);
                        value->insert(13);
                        value->insert(14);
                        value->insert(12);
                        value->insert(16);
                        value->insert(34);
                        value->insert(37);
                        value->insert(38);
                }
                break;
        case 16:
                {
                        value->insert(13);
                        value->insert(14);
                        value->insert(25);
                        value->insert(15);
                        value->insert(27);
                        value->insert(37);
                        value->insert(38);
                        value->insert(49);
                }
                break;
        case 17:
                {
                        value->insert(6);
                        value->insert(18);
                        value->insert(8);
                        value->insert(19);
                        value->insert(20);
                }
                break;
        case 18:
                {
                        value->insert(17);
                        value->insert(21);
                        value->insert(19);
                        value->insert(20);
                        value->insert(23);
                }
                break;
        case 19:
                {
                        value->insert(6);
                        value->insert(17);
                        value->insert(18);
                        value->insert(8);
                        value->insert(20);
                        value->insert(14);
                        value->insert(25);
                        value->insert(26);
                }
                break;
        case 20:
                {
                        value->insert(17);
                        value->insert(18);
                        value->insert(21);
                        value->insert(19);
                        value->insert(23);
                        value->insert(25);
                        value->insert(26);
                        value->insert(29);
                }
                break;
```

```
case 21:
        {
                value->insert(18);
                value->insert(22);
                value->insert(20);
                value->insert(23);
                value->insert(24);
        }
        break;
case 22:
        {
                value->insert(21);
                value->insert(23);
                value->insert(24);
        }
        break;
case 23:
        {
                value->insert(18);
                value->insert(21);
                value->insert(22);
                value->insert(20);
                value->insert(24);
                value->insert(26);
                value->insert(29);
                value->insert(30);
        }
        break;
case 24:
        {
                value->insert(21);
                value->insert(22);
                value->insert(23);
                value->insert(29);
                value->insert(30);
        }
        break;
case 25:
        {
                value->insert(8);
                value->insert(19);
                value->insert(20);
                value->insert(14);
                value->insert(26);
                value->insert(16);
                value->insert(27);
                value->insert(28);
        }
        break;
case 26:
        {
                value->insert(19);
                value->insert(20);
                value->insert(23);
                value->insert(25);
                value->insert(29);
```

```
				value->insert(27);
				value->insert(28);
				value->insert(31);
			}
			break;
		case 27:
			{
				value->insert(14);
				value->insert(25);
				value->insert(26);
				value->insert(16);
				value->insert(28);
				value->insert(38);
				value->insert(49);
				value->insert(50);
			}
			break;
		case 28:
			{
				value->insert(25);
				value->insert(26);
				value->insert(29);
				value->insert(27);
				value->insert(31);
				value->insert(49);
				value->insert(50);
				value->insert(53);
			}
			break;
		case 29:
			{
				value->insert(20);
				value->insert(23);
				value->insert(24);
				value->insert(26);
				value->insert(30);
				value->insert(28);
				value->insert(31);
				value->insert(32);
			}
			break;
		case 30:
			{
				value->insert(23);
				value->insert(24);
				value->insert(29);
				value->insert(31);
				value->insert(32);
			}
			break;
		case 31:
			{
				value->insert(26);
				value->insert(29);
				value->insert(30);
				value->insert(28);
			value->insert(32);
			value->insert(50);
			value->insert(53);
			value->insert(54);
		}
		break;
	case 32:
		{
			value->insert(29);
			value->insert(30);
			value->insert(31);
			value->insert(53);
			value->insert(54);
		}
		break;
	case 33:
		{
			value->insert(11);
			value->insert(12);
			value->insert(34);
			value->insert(35);
			value->insert(36);
		}
		break;
	case 34:
		{
			value->insert(11);
			value->insert(12);
			value->insert(15);
			value->insert(33);
			value->insert(37);
			value->insert(35);
			value->insert(36);
			value->insert(39);
		}
		break;
	case 35:
		{
			value->insert(33);
			value->insert(34);
			value->insert(36);
			value->insert(41);
			value->insert(42);
		}
		break;
	case 36:
		{
			value->insert(33);
			value->insert(34);
			value->insert(37);
			value->insert(35);
			value->insert(39);
			value->insert(41);
			value->insert(42);
			value->insert(45);
		}
```

```
        break;
case 37:
        {
                value->insert(12);
                value->insert(15);
                value->insert(16);
                value->insert(34);
                value->insert(38);
                value->insert(36);
                value->insert(39);
                value->insert(40);
        }
        break;
case 38:
        {
                value->insert(15);
                value->insert(16);
                value->insert(27);
                value->insert(37);
                value->insert(49);
                value->insert(39);
                value->insert(40);
                value->insert(51);
        }
        break;
case 39:
        {
                value->insert(34);
                value->insert(35);
                value->insert(38);
                value->insert(36);
                value->insert(40);
                value->insert(42);
                value->insert(45);
                value->insert(46);
        }
        break;
case 40:
        {
                value->insert(37);
                value->insert(38);
                value->insert(49);
                value->insert(39);
                value->insert(51);
                value->insert(45);
                value->insert(46);
                value->insert(57);
        }
        break;
case 41:
        {
                value->insert(35);
                value->insert(36);
                value->insert(42);
                value->insert(43);
                value->insert(44);
```

```
		}
		break;
	case 42:
		{
			value->insert(35);
			value->insert(36);
			value->insert(39);
			value->insert(41);
			value->insert(45);
			value->insert(43);
			value->insert(44);
			value->insert(47);
		}
		break;
	case 43:
		{
			value->insert(41);
			value->insert(42);
			value->insert(44);
		}
		break;
	case 44:
		{
			value->insert(41);
			value->insert(42);
			value->insert(45);
			value->insert(43);
			value->insert(47);
		}
		break;
	case 45:
		{
			value->insert(36);
			value->insert(39);
			value->insert(40);
			value->insert(42);
			value->insert(46);
			value->insert(44);
			value->insert(47);
			value->insert(48);
		}
		break;
	case 46:
		{
			value->insert(39);
			value->insert(40);
			value->insert(51);
			value->insert(45);
			value->insert(57);
			value->insert(47);
			value->insert(48);
			value->insert(59);
		}
		break;
	case 47:
		{
```

```
                value->insert(42);
                value->insert(45);
                value->insert(46);
                value->insert(44);
                value->insert(48);
        }
        break;
case 48:
        {
                value->insert(45);
                value->insert(46);
                value->insert(57);
                value->insert(47);
                value->insert(59);
        }
        break;
case 49:
        {
                value->insert(16);
                value->insert(27);
                value->insert(28);
                value->insert(38);
                value->insert(50);
                value->insert(40);
                value->insert(51);
                value->insert(52);
        }
        break;
case 50:
        {
                value->insert(27);
                value->insert(28);
                value->insert(31);
                value->insert(49);
                value->insert(53);
                value->insert(51);
                value->insert(52);
                value->insert(55);
        }
        break;
case 51:
        {
                value->insert(38);
                value->insert(49);
                value->insert(50);
                value->insert(40);
                value->insert(52);
                value->insert(46);
                value->insert(57);
                value->insert(58);
        }
        break;
case 52:
        {
                value->insert(49);
                value->insert(50);
                value->insert(53);
                value->insert(51);
                value->insert(55);
                value->insert(57);
                value->insert(58);
                value->insert(61);
        }
        break;
case 53:
        {
                value->insert(28);
                value->insert(31);
                value->insert(32);
                value->insert(50);
                value->insert(54);
                value->insert(52);
                value->insert(55);
                value->insert(56);
        }
        break;
case 54:
        {
                value->insert(31);
                value->insert(32);
                value->insert(53);
                value->insert(55);
                value->insert(56);
        }
        break;
case 55:
        {
                value->insert(50);
                value->insert(53);
                value->insert(54);
                value->insert(52);
                value->insert(56);
                value->insert(58);
                value->insert(61);
                value->insert(62);
        }
        break;
case 56:
        {
                value->insert(53);
                value->insert(54);
                value->insert(55);
                value->insert(61);
                value->insert(62);
        }
        break;
case 57:
        {
                value->insert(40);
                value->insert(51);
                value->insert(52);
                value->insert(46);
                value->insert(58);
                value->insert(48);
                value->insert(59);
                value->insert(60);
        }
        break;
case 58:
        {
                value->insert(51);
                value->insert(52);
                value->insert(55);
                value->insert(57);
                value->insert(61);
                value->insert(59);
                value->insert(60);
                value->insert(63);
        }
        break;
case 59:
        {
                value->insert(46);
                value->insert(57);
                value->insert(58);
                value->insert(48);
                value->insert(60);
        }
        break;
case 60:
        {
                value->insert(57);
                value->insert(58);
                value->insert(61);
                value->insert(59);
                value->insert(63);
        }
        break;
case 61:
        {
                value->insert(52);
                value->insert(55);
                value->insert(56);
                value->insert(58);
                value->insert(62);
                value->insert(60);
                value->insert(63);
                value->insert(64);
        }
        break;
case 62:
        {
                value->insert(55);
                value->insert(56);
                value->insert(61);
                value->insert(63);
                value->insert(64);
        }
        break;
    case 63:
        {
            value->insert(58);
            value->insert(61);
            value->insert(62);
            value->insert(60);
            value->insert(64);
        }
        break;
    case 64:
        {
            value->insert(61);
            value->insert(62);
            value->insert(63);
        }
        break;
    }

    // make sure that values do not exceed
    // the number of lattice
    Filter(*value);

    restrictors_.insert(RESTRICTTORMAP::value_type(size, value));
    return *value;
}
```

```
#if !defined(COMMAND_LINE_INTERPRETER__HH)
#define COMMAND_LINE_INTERPRETER__HH

#include <string>

class CommandLineInterpreter
{

public:
        CommandLineInterpreter() :
          lattice_(4), points_(10), sampleSize_(100), outputFileName_("out.log") {}
        void Parse(int argc, char** argv);

        int NoLatticePoints() { return lattice_; }
        int NoPoints() { return points_; }
        int SampleSize() { return sampleSize_; }
        std::string& OutputFileName() { return outputFileName_; }
        std::string& PointsMatrixFile() { return pointsMatrixFile_; }

private:
        int lattice_;
        int points_;
        int sampleSize_;
        std::string outputFileName_;
        std::string pointsMatrixFile_;

};

#endif
```

```
 #if !defined(MANSAH__CONST_KNUTSTRRAND_GEN__HH)
#define MANSAH__CONST_KNUTSTRRAND_GEN__HH

#include <string>
#include "RandGenImpl.h"
#include"RandomGenerator.h"

class ConstraintKnutStrRandomGenerator : public RandomGenerator<std::string>
{
public:
        explicit ConstraintKnutStrRandomGenerator(RandGenImpl& randGen)
                : RandomGenerator<std::string>(randGen) { }
        void generate(std::vector<std::string>& ranVec, int howMany, int size,
                                  int lowerValue, int upperValue);

};

#endif // ConstraintKnutStrRandomGenerator
```

```
#if !defined(MATRIX_CALCULATOR__HH)
#define MATRIX_CALCULATOR__HH

typedef std:vector<int> IntVector;
typedef std::vector<IntVector> IntMatrix;

class DistanceMatrixCalculaor
{
public:
        IntMatrix& CalculateODDistMatrix(int size)
        IntMatrix& CalculateODDistMatrix(int size);

private:
        IntMatrix odMatrix_;
        IntMatrix ndMatrix_;

};


#endif
```

```
#if !defined (KNUT_INT_RAND_GEN__H)
#define KNUT_INT_RAND_GEN__H

#include <string>
#include "RandGenImpl.h"
#include"RandomGenerator.h"

class KnutIntRandomGenerator : public RandomGenerator<int>
{
public:
        explicit KnutIntRandomGenerator(RandGenImpl& randGen)
                : RandomGenerator<int>(randGen) { }
        void generate(std::vector<int>& ranVec, int howMany, int size,
                                    int lowerValue, int upperValue);
};


#endif
```

```
#if !defined(MANSAH__KNUT_RAND_GEN__HH)
#define MANSAH__KNUT_RAND_GEN__HH

#include <string>
#include"RandGenImpl.h"

class KnutRandomGenerator : public RandGenImpl
{
public:
        KnutRandomGenerator(long seed) : RandGenImpl(seed) { }
        float ran() { return RandGenImpl::ran(); }

};

#endif
```

```
#if !defined(MANSAH__KNUTSTRRAND_GEN__HH)
#define MANSAH__KNUTSTRRAND_GEN__HH

#include <string>
#include "RandGenImpl.h"
#include"RandomGenerator.h"

class KnutStrRandomGenerator : public RandomGenerator<std::string>
{
public:
        explicit KnutStrRandomGenerator(RandGenImpl& randGen)
                : RandomGenerator<std::string>(randGen) { }
        void generate(std::vector<std::string>& ranVec, int howMany, int size,
                                        int lowerValue, int upperValue);

};

#endif
```

```
#if !defined (LOCAL_SEARCH_MODELER__H)
#define LOCAL_SEARCH_MODELER__H



class LocalSearchModeler
{
public :
        LocalSearchModeler() { }
        virtual ~LocalSearchModeler() { }
        void Model();

protected:
        virtual void GenerateInitialSolution() = 0;
        virtual void CalculateObjectiveValue() = 0;
        virtual void PerformLocalSearch() = 0;

private:
        LocalSearchModeler(LocalSearchModeler& other); // not defined to prevent copying
        LocalSearchModeler& operator=(LocalSearchModeler& other); // not defined to  assignment

};



#endif
```

```
#if !defined (LOCAL_SEARCH_MODELER__H)
#define LOCAL_SEARCH_MODELER__H



class LocalSearchModeler
{
public :
	LocalSearchModeler() { }
	virtual ~LocalSearchModeler() { }
	void Model();

protected:
	virtual void GenerateInitialSolution() = 0;
	virtual void CalculateObjectiveValue() = 0;
	virtual void PerformLocalSearch() = 0;

private:
	LocalSearchModeler(LocalSearchModeler& other); // not defined to prevent copying
	LocalSearchModeler& operator=(LocalSearchModeler& other); // not defined to  assignment

};



#endif
```

```
#if !defined(MATRIX__H)
#define MATRIX__H

#include "Utility.h"

template <typename T>
class Matrix
{
public:
	Matrix() { }
	virtual ~Matrix() { }

	virtual int columns() = 0;
	virtual int rows() = 0;
	virtual T& GetMatrix() = 0;

};

#endif
```

```
#if !defined(MODEL_FIVE__H)
#define MODEL_FIVE__H

#include "LocalSearchModeler.h"
#include "Matrix.h"
#include <vector>
#include <cassert>

#include "ModelOne.h"
#include "RestrictionsAllocator.h"

template <typename T1, typename T2, typename ObjectCalc>
class ModelFive : public LocalSearchModeler, private ObjectCalc
{
        typedef std::string MOD5STR;

public:
        ModelFive
                (
                  T1& ranGen, Matrix<T2>& odMat, const char* outFile = "out.log",
                        int lattice = 4, int points = 10,
                        int sample = 100, const char* inMatrixFile = 0
                ) :
                        randomGenerator_(ranGen), mPointSize_(points), nLatticeSize_(lattice),
                                sample_(sample), odMat_(odMat), outPutFile_(outFile ? outFile :
"out.log"),
                                matrixFile_(inMatrixFile ? inMatrixFile : "")
                { }

        ~ModelFive() { }

private:

        T1& randomGenerator_;
        int mPointSize_;
        int nLatticeSize_;
        int sample_;
        std::string outPutFile_;
        std::string matrixFile_;
        StrVec solutionVec_;
        Matrix<T2>& odMat_;

        // from LocalSearchModeler
        void GenerateInitialSolution();
        void CalculateObjectiveValue();
        void PerformLocalSearch();

        void RecreateIntString(MOD5STR& str, IntVec& intVec);
        void ExtractIntegersFromString(IntVec& intVec, const MOD5STR& str);

        void printResult(int mode = std::ios::out, char* msg = 0)
        {
                printContainerResult(solutionVec_, mode , msg);
        }
```

```
}; // MODEL_FIVE__H


template <typename T1, typename T2, typename ObjectCalc>
void ModelFive<T1, T2, ObjectCalc>::GenerateInitialSolution()
{

        // generating the initial solution using model one
        RecipeRandGenImpl gRanGen(-1L);
        ConstraintKnutStrRandomGenerator gKnutRanGen(gRanGen);
        ODMatrixGen odMatGen(matrixFile_.c_str());
        ModelOne<ConstraintKnutStrRandomGenerator, IntMatrix, ObjValueCalculator>
                        modelOne(gKnutRanGen, odMatGen, outPutFile_.c_str(),
                        nLatticeSize_, mPointSize_, sample_, matrixFile_.c_str());

        modelOne.Model();
        StrVec& initSol = modelOne.GetSolution();

        // now we have the initial solution from model one we have to proceed to do the rest
        solutionVec_.swap(initSol);

} /// GenerateInitialSolution


template <typename T1, typename T2, typename ObjectCalc>
void ModelFive<T1, T2, ObjectCalc>::CalculateObjectiveValue()
{
 // nothing to be done here
} // CalculateObjectiveValue


template <typename T1, typename T2, typename ObjectCalc>
void ModelFive<T1, T2, ObjectCalc>::PerformLocalSearch()
{

        NDMatrixGen ndMat;
        ndMat.GetMatrix(nLatticeSize_);
        IntVec strElemAsIntVec;
        RestrictionsAllocator resAlloc(nLatticeSize_);

        for(int i = 0; i < solutionVec_.size(); ++i)
        {
                ExtractIntegersFromString(strElemAsIntVec, solutionVec_[i]);
                for(int j = 1; j < strElemAsIntVec.size(); ++j)
                {

                        LNGINTPAIR objValuepair = CalculatObjectiveValue(j, solutionVec_[i],
                                                                odMat_.GetMatrix(),
ndMat.GetMatrix(),

        resAlloc.FindSet(strElemAsIntVec[j]));

                        // assign poisition value with result to change string
                        strElemAsIntVec[j] = objValuepair.second;
                }
                RecreateIntString(solutionVec_[i], strElemAsIntVec);
        }

        // print the local serach result

        printResult(std::ios::app, "MODEL V: Results After Local Search");


        // calculate new overall objective function.
        LNGVEC objValueVec;
        for(unsigned int cnt = 0; cnt < solutionVec_.size(); ++cnt)
        {
                objValueVec.push_back(CalculatObjectiveValue(solutionVec_[cnt],
                        odMat_.GetMatrix(), ndMat.GetMatrix()));
        }


        printContainerResult(objValueVec, std::ios::app, "MODEL V: Results of Objective Values");

} // PerformLocalSearch


template <typename T1, typename T2, typename ObjectCalc>
void ModelFive<T1, T2, ObjectCalc>::ExtractIntegersFromString(IntVec& intVec, const
MOD5STR& str)
{
        if(!intVec.empty()) intVec.clear();
        unsigned int strSize = str.size();
        MOD5STR buffer;
        bool hadAnElement = false;
        bool extractBuffer = false;
        for(unsigned int cnt = 0; cnt < strSize; ++cnt)
        {
                if(str[cnt] != 'x')
                {
                        buffer += str[cnt];
                        hadAnElement = true;

                }
                else if(hadAnElement)
                {
                        int value = atoi(buffer.c_str());
                        intVec.push_back(value);
                        hadAnElement =  false;
                        if(!buffer.empty()) buffer = "";
                }
        } // for

        // record the last value
        intVec.push_back(atoi(buffer.c_str()));

} // ExtractIntegersFromString


template <typename T1, typename T2, typename ObjectCalc>
void ModelFive<T1, T2, ObjectCalc>::RecreateIntString(MOD5STR& str, IntVec& intVec)
```

```
{
	IntVec::iterator iter = intVec.begin();
	IntVec::iterator iterEnd = intVec.end();

	const char BUFSIZE = 10;
	char buffer[BUFSIZE];
	MOD5STR newStr;

	for(; iter != iterEnd; ++iter)
	{
		memset(buffer, 0, sizeof(char) * BUFSIZE);
		if(*iter < 10)
			sprintf(buffer, "xx%d", *iter);
		else if(*iter < 100)
			sprintf(buffer, "x%d", *iter);
		else if(*iter < 1000)
			sprintf(buffer, "%d", *iter);
		else
			throw 1; // error

		assert(strlen(buffer) < 4); // we can take only up to 999
		if(strlen(buffer) > 3) throw 1;

		newStr += buffer;
	}

	str = newStr;

} // RecreateIntString

#endif
```

```
#if !defined(MODEL_ONE__H)
#define MODEL_ONE__H

#include "Utility.h"
#include "LocalSearchModeler.h"
#include "Matrix.h"
#include <vector>
#include <cassert>
#include "NDMatrixGen.h"

template <typename T1, typename T2, typename ObjectCalc>
class ModelOne : public LocalSearchModeler, private ObjectCalc
{
	typedef std::string MOD1STR;

public:
	ModelOne
		(
		  T1& ranGen, Matrix<T2>& odMat, const char* outFile = "out.log",
			int lattice = 4, int points = 10,
			int sample = 100, const char* inMatrixFile = 0
		) :
			randomGenerator_(ranGen), mPointSize_(points), nLatticeSize_(lattice),
				sample_(sample), odMat_(odMat), outPutFile_(outFile ? outFile :
"out.log"),
				matrixFile_(inMatrixFile ? inMatrixFile : "")
		{ }

	~ModelOne() { }
	StrVec& GetSolution() { return solutionVec_; }

private:

	T1& randomGenerator_;
	int mPointSize_;
	int nLatticeSize_;
	int sample_;
	std::string outPutFile_;
	std::string matrixFile_;
	StrVec solutionVec_;
	Matrix<T2>& odMat_;

	// for objective value
	DoubleVec objectiveVector_;

	void GenerateInitialSolution();
	void CalculateObjectiveValue();
	void PerformLocalSearch();

	void printResult(int mode = std::ios::out, char* msg = 0)
	{
		printContainerResult(solutionVec_, mode , msg);
	}


	void RecreateIntString(MOD1STR& str, IntVec& intVec);
	void ExtractIntegersFromString(IntVec& intVec, const MOD1STR& str);

};



template <typename T1, typename T2, typename ObjectCalc>
void ModelOne<T1, T2, ObjectCalc>::GenerateInitialSolution()
{
	randomGenerator_.generate(solutionVec_, 100, mPointSize_, 1, nLatticeSize_);

	char buffer[512];
	sprintf(buffer, "MODEL I: Results Initial Solution;\n\t # lattice points = %d"
		"\t# of points = %d\tSample size = %d", nLatticeSize_, mPointSize_, sample_);
	printResult(std::ios::out, buffer);
//	printResult(std::ios::out, "MODEL I: Initial Solution");
}


template <typename T1, typename T2, typename ObjectCalc>
void ModelOne<T1, T2, ObjectCalc>::CalculateObjectiveValue()
{

	/*
	if(!objectiveVector_.empty()) objectiveVector_.clear();
	char[4] buffer;

	StrVec::iterator iter = solutionVec_::begin();
	StrVec::iterator iterEnd = solutionVec_::end();
	std::string::iterator strIter;
	std::string::iterator strIterEnd;

	// extract values form string
	for(; iter != iterEnd; ++iter)
	{
		std::vector<int> latticeValues;
		char* ptr = (*iter)[0];
		for(; ptr != 0; ptr + 3) // 3 = # representing a position
		{
			memset(buffer, 0, 4 * sizeof(char));
			strncpy(buffer, ptr, 3);
			std::string digit;
			for( int i = 0; buffer[i] != 0; ++i)
			{
				if(isdigit(buffer[i])
				{
					digit += buffer[i];
				}
			} // for

			if(!digit.empty())
			{
				latticeValues.push_back(atoi(digit.c_str()));
			}

		}
```

```
        assert(latticeValues.size() == mPointSize_); // check point

        // now calculate the objective value
        for(int j = 0; j < mPointSize_; ++j)
        {

        }
    } // for


    /*
    ///////////////////////////////
    for(int i = 1; i < mPointSize_; ++i)
    {
        for (int j = i + 1; j < mPointSize_; ++j)
        {
            int l = j;
            int k = i;
            double value = od[i,j] - nd[k,l];
            value *= value;
        }

        objectiveVector_.push_back(value);
    }
*/


}

template <typename T1, typename T2, typename ObjectCalc>
void ModelOne<T1, T2, ObjectCalc>::ExtractIntegersFromString(IntVec& intVec, const
MOD1STR& str)
{
    if(!intVec.empty()) intVec.clear();
    unsigned int strSize = str.size();
    MOD1STR buffer;
    bool hadAnElement = false;
    bool extractBuffer = false;
    for(unsigned int cnt = 0; cnt < strSize; ++cnt)
    {
        if(str[cnt] != 'x')
        {
            buffer += str[cnt];
            hadAnElement = true;

        }
        else if(hadAnElement)
        {
            int value = atoi(buffer.c_str());
            intVec.push_back(value);
            hadAnElement =  false;
            if(!buffer.empty()) buffer = "";
        }
    } // for
```

```
		// record the last value
		intVec.push_back(atoi(buffer.c_str()));

} // ExtractIntegersFromString


template <typename T1, typename T2, typename ObjectCalc>
void ModelOne<T1, T2, ObjectCalc>::RecreateIntString(MOD1STR& str, IntVec& intVec)
{
		IntVec::iterator iter = intVec.begin();
		IntVec::iterator iterEnd = intVec.end();

		const char BUFSIZE = 10;
		char buffer[BUFSIZE];
		MOD1STR newStr;

		for(; iter != iterEnd; ++iter)
		{
			memset(buffer, 0, sizeof(char) * BUFSIZE);
			if(*iter < 10)
				sprintf(buffer, "xx%d", *iter);
			else if(*iter < 100)
				sprintf(buffer, "x%d", *iter);
			else if(*iter < 1000)
				sprintf(buffer, "%d", *iter);
			else
				throw 1; // error

			assert(strlen(buffer) < 4); // we can take only up to 999
			if(strlen(buffer) > 3) throw 1;

			newStr += buffer;
		}

		str = newStr;

} // RecreateIntString


template <typename T1, typename T2, typename ObjectCalc>
void ModelOne<T1, T2, ObjectCalc>::PerformLocalSearch()
{

		NDMatrixGen ndMat;
		ndMat.GetMatrix(nLatticeSize_);
		IntVec strElemAsIntVec;

		for(int i = 0; i < solutionVec_.size(); ++i)
		{
			ExtractIntegersFromString(strElemAsIntVec, solutionVec_[i]);
			for(int j = 1; j < strElemAsIntVec.size(); ++j)
			{

				LNGINTPAIR objValuepair = CalculatObjectiveValue(j, solutionVec_[i],
											odMat_.GetMatrix(),
ndMat.GetMatrix() );
```

```
            // assign poisition value with result to change string
            strElemAsIntVec[j] = objValuepair.second;
        }

        RecreateIntString(solutionVec_[i], strElemAsIntVec);
    }

    // print the local serach result
    char buffer[512];
    sprintf(buffer, "MODEL I: Results After Local Search;\n\t # lattice points = %d"
        "\t# of points = %d\tSample size = %d", nLatticeSize_, mPointSize_, sample_);
    printResult(std::ios::app, buffer);

//  printResult(std::ios::app, "MODEL I: Results After Local Search");


    // calculate new overall objective function.
    LNGVEC objValueVec;
    for(unsigned int cnt = 0; cnt < solutionVec_.size(); ++cnt)
    {
        objValueVec.push_back(CalculatObjectiveValue(solutionVec_[cnt],
            odMat_.GetMatrix(), ndMat.GetMatrix()));
    }

    printContainerResult(objValueVec, std::ios::app, "MODEL I: Results of Objective Values");

}


/*
template <typename T1, typename T2, typename ObjectCalc>
void ModelOne<T1, T2, ObjectCalc>::printResult(int mode, char* msg)
{



    std::ofstream outStream("out.log", mode);

    if(msg)
        outStream << msg << std::endl;

    StrVec::iterator iter = solutionVec_.begin();
 StrVec::iterator iterEnd = solutionVec_.end();
    for(int i = 1; iter != iterEnd; ++iter, ++i)
    {

        outStream << (*iter) << " ";
        if(i%5 == 0) outStream << std::endl;
    }
    outStream << std::endl;
    outStream << std::endl;



}
*/
```

```
#endif



#if !defined(ND_MATRIX_GEN__H)
#define ND_MATRIX_GEN__H

#include "matrix.h"
#include <map>


class NDMatrixGen : public Matrix<IntMatrix>
{
public:
	typedef std::pair<int*, int*> IntPointerPair;
	typedef std::map<int, IntPointerPair> IntPointerPairMap;
public:
	NDMatrixGen();
	~NDMatrixGen() { }

	int columns() { return ndMatrix_.empty() ? 0 : ndMatrix_[0].size(); }
	int rows() { return ndMatrix_.size(); }
	IntMatrix& GetMatrix() { return ndMatrix_; }
	IntMatrix& GetMatrix(int size) { Generate(size); return ndMatrix_; }

private:

	int type_;
	IntPointerPairMap matrixContainer_;
	IntMatrix ndMatrix_;



	void Generate(int size);

};

#endif
```

```
#if !defined(MANSAH__OBJ_VALUE_CALC__HH)
#define MANSAH__OBJ_VALUE_CALC__HH

#include "Utility.h"
#include <string>
#include <vector>
#include <set>
#include "Matrix.h"


class ObjValueCalculator
{
        typedef std::string OBJVString;
        typedef std::vector<int> OBJIntVec;
        typedef std::vector<LNGINTPAIR> LNINTPAIRVEC;

public:
        LNGINTPAIR CalculatObjectiveValue
                (
                        unsigned int pos,
                        const OBJVString& str,
                        IntMatrix& odMat,
                        IntMatrix& ndMat
                );


        LNGINTPAIR CalculatObjectiveValue
                (
                        unsigned int pos,
                        const OBJVString& str,
                        IntMatrix& odMat,
                        IntMatrix& ndMat,
                        INTSET& relLPts

                );


        long CalculatObjectiveValue
                (
                        const OBJVString& str,
                        IntMatrix& odMat,
                        IntMatrix& ndMat
                );

private:
        void ExtractValues(const OBJVString& inStr, OBJIntVec& outVec);

        void CalculatObjectiveValue
                (
                        unsigned int pos,
                        const OBJVString& str,
                        IntMatrix& odMat,
                        IntMatrix& ndMat,
                        LNINTPAIRVEC& objectiveFunctionList

                );
};

#endif
```

```
#if !defined(OD_MATRIX_GEN__H)
#define OD_MATRIX_GEN__H

#include "matrix.h"
#include <string>


class ODMatrixGen : public Matrix<IntMatrix>
{
public:
	~ODMatrixGen() { }
	explicit ODMatrixGen(const char* fileName);
	explicit ODMatrixGen(std::string& fileName);

	int columns() { if(!generated_) Generate();
	if(!odMatrix_.empty()) { return odMatrix_[0].size();} else return 0; }
	int rows() { if(!generated_) Generate(); return odMatrix_.size(); }
	IntMatrix& GetMatrix() { return odMatrix_; }

private:
	bool generated_;
	IntMatrix odMatrix_;
	std::string fileName_;
	void Generate();
	void Generate(bool fromFile);

};

#endif
```

```
#if !defined(MANSAH__RAND_GEN__IMP__HH)
#define MANSAH__RAND_GEN__IMP__HH

#include <string>
#include <ctime>

class RandGenImpl
{
public:
        explicit RandGenImpl(long idum) : idum_(idum) { } // use -ve value to initialize
        RandGenImpl() { idum_ = time(0); idum_ = -idum_;} // for true random
        virtual float ran() = 0 { return ran(&idum_); }
        virtual ~RandGenImpl() { }

private :
        long idum_;
        float RandGenImpl::ran(long *idum);

};

#endif
```

```
#if !defined(MANSAH__RAND_GEN__HH)
#define MANSAH__RAND_GEN__HH

#include <vector>

class RandGenImpl;
template < class T>
class RandomGenerator
{
public:
        virtual void generate( std::vector<T>& ranVec, int howMany, int size,
                                                                int lowerValue, int upperValue ) = 0;
        RandomGenerator(RandGenImpl& randGen) : randGen_(randGen) { }
        virtual ~RandomGenerator() { }

protected:
                float ran() { return randGen_.ran(); }
private:
        RandGenImpl& randGen_;
};

#endif
```

```
#if !defined(MANSAH__RECIPE_RAND_GEN__IMP__HH)
#define MANSAH__RECIPE_RAND_GEN__IMP__HH

#include "RandGenImpl.h"

class RecipeRandGenImpl : public RandGenImpl
{

public:
        explicit RecipeRandGenImpl(long idum) : RandGenImpl(idum) { }
        RecipeRandGenImpl() { }
        float ran() { return RandGenImpl::ran(); }

};

#endif
```

```
#if !defined(MANSAH_RESTICTIONS_ALLOCATOR__HH)
#define MANSAH_RESTICTIONS_ALLOCATOR__HH

#include "ObjectiveValueCalculator.h"
#include <map>

class RestrictionsAllocator
{
        typedef std::map<int, INTSET*, std::less<int> > RESTRICTTORMAP;

public:
        explicit RestrictionsAllocator(int maxLatticeValue) : maxLatticeValue_(maxLatticeValue) {
}
        RestrictionsAllocator() : maxLatticeValue_(4) { }
        ~RestrictionsAllocator();

        INTSET& FindSet(int size);

private:
        int maxLatticeValue_;
        RESTRICTTORMAP restrictors_;

        INTSET& Allocate(int size);
        void Filter(INTSET& object);


}; // MANSAH_RESTICTIONS_ALLOCATOR__HH

#endif
```

```
#if !defined(UTILITY__MANSA__HH)
#define UTILITY__MANSA__HH
#include <vector>
#include <set>
#include <fstream>

typedef std::vector<int> IntVec;
typedef std::vector<IntVec> IntMatrix;

typedef std::vector<std::string> StrVec;
typedef std::vector<double> DoubleVec;

typedef std::vector<double> DBLVEC;
typedef std::pair<long, int> LNGINTPAIR;
typedef std::vector<long> LNGVEC;
typedef std::set<int, std::less<int> > INTSET;

template <class T>
void printContainerResult(T& list, int mode = std::ios::out, char* msg = 0)
{

        std::ofstream outStream("out.log", mode);

        if(msg)
                outStream << msg << std::endl;

        T::iterator iter = list.begin();
        T::iterator iterEnd = list.end();
        for(int i = 1; iter != iterEnd; ++iter, ++i)
        {

                outStream << (*iter) << " ";
                if(i%5 == 0) outStream << std::endl;
        }
        outStream << std::endl;
        outStream << std::endl;

} // printContainerResult

#endif
```

Appendix B

Source code of IRNS

```
Model timer ("easy.mod"); // calculates running time

timer.solve();

float+ begintime := timer.getTime();

data "latticepoints256.dat"; // file containing latticepoints

data "SM4-1504-5.dat"; // file containing original points

enum points ... ; // original points

int+ xcoorm[points] := ... ; // x-coordinates of original points
int+ ycoorm[points] := ... ; // y-coordinates of original points
int+ zcoorm[points] := ... ;

float+ od[i in points, j in points] := sqrt((xcoorm[i] - xcoorm[j])*(xcoorm[i] - xcoorm[j]) + (ycoorm[i]
- ycoorm[j])*(ycoorm[i] - ycoorm[j]) + (zcoorm[i] - zcoorm[j])*(zcoorm[i] - zcoorm[j]));
// distance matrix for original points

enum latticepoints ... ; // latticepoints

int+ xcoorn[latticepoints] := ... ; // y-coordinates of latticepoints
int+ ycoorn[latticepoints] := ... ; // y-coordinates of latticepoints

setof(latticepoints) N1 := ... ; // subsets of latticepoints used at different steps of the model

setof(latticepoints) N21 := ... ;
setof(latticepoints) N22 := ... ;
setof(latticepoints) N23 := ... ;
setof(latticepoints) N24 := ... ;

setof(latticepoints) N31 := ... ;
setof(latticepoints) N32 := ... ;
setof(latticepoints) N33 := ... ;
setof(latticepoints) N34 := ... ;
setof(latticepoints) N35 := ... ;
setof(latticepoints) N36 := ... ;
setof(latticepoints) N37 := ... ;
setof(latticepoints) N38 := ... ;
setof(latticepoints) N39 := ... ;
setof(latticepoints) N310 := ... ;
setof(latticepoints) N311 := ... ;
setof(latticepoints) N312 := ... ;
setof(latticepoints) N313 := ... ;
setof(latticepoints) N314 := ... ;
setof(latticepoints) N315 := ... ;
setof(latticepoints) N316 := ... ;
setof(latticepoints) N41 := ... ;
setof(latticepoints) N42 := ... ;
setof(latticepoints) N43 := ... ;
setof(latticepoints) N44 := ... ;
setof(latticepoints) N45 := ... ;
setof(latticepoints) N46 := ... ;
setof(latticepoints) N47 := ... ;
setof(latticepoints) N48 := ... ;
setof(latticepoints) N49 := ... ;
setof(latticepoints) N410 := ... ;
setof(latticepoints) N411 := ... ;
setof(latticepoints) N412 := ... ;
setof(latticepoints) N413 := ... ;
setof(latticepoints) N414 := ... ;
setof(latticepoints) N415 := ... ;
setof(latticepoints) N416 := ... ;
setof(latticepoints) N417 := ... ;
setof(latticepoints) N418 := ... ;
setof(latticepoints) N419 := ... ;
setof(latticepoints) N420 := ... ;
setof(latticepoints) N421 := ... ;
setof(latticepoints) N422 := ... ;
setof(latticepoints) N423 := ... ;
setof(latticepoints) N424 := ... ;
setof(latticepoints) N425 := ... ;
setof(latticepoints) N426 := ... ;
setof(latticepoints) N427 := ... ;
setof(latticepoints) N428 := ... ;
setof(latticepoints) N429 := ... ;
setof(latticepoints) N430 := ... ;
setof(latticepoints) N431 := ... ;
setof(latticepoints) N432 := ... ;
setof(latticepoints) N433 := ... ;
setof(latticepoints) N434 := ... ;
setof(latticepoints) N435 := ... ;
setof(latticepoints) N436 := ... ;
setof(latticepoints) N437 := ... ;
setof(latticepoints) N438 := ... ;
setof(latticepoints) N439 := ... ;
setof(latticepoints) N440 := ... ;
setof(latticepoints) N441 := ... ;
setof(latticepoints) N442 := ... ;
setof(latticepoints) N443 := ... ;
setof(latticepoints) N444 := ... ;
setof(latticepoints) N445 := ... ;
setof(latticepoints) N446 := ... ;
setof(latticepoints) N447 := ... ;
setof(latticepoints) N448 := ... ;
setof(latticepoints) N449 := ... ;
setof(latticepoints) N450 := ... ;
setof(latticepoints) N451 := ... ;
setof(latticepoints) N452 := ... ;
setof(latticepoints) N453 := ... ;
setof(latticepoints) N454 := ... ;
setof(latticepoints) N455 := ... ;
setof(latticepoints) N456 := ... ;
setof(latticepoints) N457 := ... ;
setof(latticepoints) N458 := ... ;
setof(latticepoints) N459 := ... ;
setof(latticepoints) N460 := ... ;
setof(latticepoints) N461 := ... ;
setof(latticepoints) N462 := ... ;
setof(latticepoints) N463 := ... ;
setof(latticepoints) N464 := ... ;

float+ Nd[i in latticepoints, j in latticepoints] := sqrt(1.5*((xcoorn[i] - xcoorn[j])*(xcoorn[i] -
xcoorn[j]) + (ycoorn[i] - ycoorn[j])*(ycoorn[i] - ycoorn[j])));
// distance matrix for latticepoints

latticepoints Assignment[points]; // stores assignment for points

// The following sets are used to divide the set of points to be assigned into sets of 10

setof(points) Q; // contains the point with the smallest order
setof(points) S; // contains the 10 points to be assigned
setof(points) R; // contains points that are yet to be assigned
int minimum; // the order of the point with the least order
int counter := 0; // counter for deciding which model to use in step 1

float+ OB := 0; // objective value

float+ OB1 := 0; // objective value after step 1
float+ OB2 := 0; // step 1 stops when OB1 = OB2

setof(points) Opoints := {k | k in points}; // set of all original points
setof(points) M :=  { i | i in points}; // set of points to be assigned at a particular time
setof(latticepoints) N := {j | j in N1}; // latticepoints to be used at at a particular time

setof(points) Mb; // points kept constant in step 1
setof(latticepoints) Na; // subset of N to which points can be assigned
setof(points) Ma; // points kept constant in all other steps


setof(latticepoints) NN[i in points] := {}; // neighboring lattice points for point i


forall(i in points)

   Assignment[i] := latpt_one_one;

Model mathprogram1 ("step31.mod") editMode; // model for initial run of step 1

Model mathprogram1a ("step31a.mod") editMode; // model for all other runs of step 1

//step 1

Mb := {};

R := {i| i in points};

repeat {
   S := {};

   repeat {

     minimum := min (i in R) ord(i);

     Q := {i| i in R : ord(i) = minimum};

     S := S union Q;

     R := R diff Q;

   } until card(S) >= 10 \/ card(R) = 0;

   counter := counter + 1;

   M := {i| i in S};

   if counter = 1 then {

     mathprogram1.solve();

//  cout << "objective value after step 1' is : " << mathprogram1.objectiveValue() << endl;

//  cout << " Time for step 1 is : " << mathprogram1.getTime() << endl;

   forall(j in N){
     forall (i in M : mathprogram1.X[i,j] = 1)
       Assignment[i] := j;
   }

     mathprogram1.reset();
   }

   if counter > 1 then {

     mathprogram1a.solve();

//  cout << "objective value after step 1" is : " << mathprogram1a.objectiveValue() << endl;

//  cout << " Time for step 1 is : " << mathprogram1a.getTime() << endl;

   forall(j in N){
     forall (i in M : mathprogram1a.X[i,j] = 1)
       Assignment[i] := j;
   }
    mathprogram1a.reset();
   }

   Mb :=  Mb union M;

} until card(R) = 0;

OB2 := sum (ordered i,j in points)
     (od[i,j] - Nd[Assignment[i],Assignment[j]]) * (od[i,j] - Nd[Assignment[i],Assignment[j]]);
```

```
cout << "The objective function after step 1 is: " << OB2 << endl;

//step 1 is repeated until solution converges


repeat {

OB1 := OB2;

OB2 := 0;

counter := 0;


R := {i| i in points};

repeat {

  S := {};

  repeat {

    minimum := min (i in R) ord(i);

    Q := {i| i in R : ord(i) = minimum};

    S := S union Q;

    R := R diff Q;

  } until card(S) >= 10 \/ card(R) = 0;

  counter := counter + 1;

  M := {i| i in S};

  Mb := Opoints diff M;

  mathprogram1a.solve();

  forall(j in N){
    forall (i in M : mathprogram1a.X[i,j] = 1)
      Assignment[i] := j;
  }

  mathprogram1a.reset();

} until card(R) = 0;


OB2 := sum (ordered i,j in points)
    (od[i,j] - Nd[Assignment[i],Assignment[j]]) * (od[i,j] - Nd[Assignment[i],Assignment[j]]);

} until OB1 = OB2;
```

```
OB := OB1;

cout << "The objective function after step 1 is: " << OB << endl;


Model mathprogram2 ("step32a.mod") editMode;

// step 2

M := { i |i in points : Assignment[i] = latpt_one_one};
N := {j | j in N21};
Na := {k | k in N: ( 4 <= ord(k) < 8)};
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

//  cout << "objective value after step 2a is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 2a is : " << mathprogram2.getTime() << endl;

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_two_one};
N := {j | j in N22};
Na := {k | k in N: ( 8 <= ord(k) < 12)};
```

```
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

   repeat {

      S := {};

      repeat {

         minimum := min (i in R) ord(i);

         Q := {i| i in R : ord(i) = minimum};

         S := S union Q;

         R := R diff Q;

      } until card(S) >= 10 \/ card(R) = 0;

      M := {i| i in S};

      Ma := Opoints diff M;

      mathprogram2.solve();

      forall (j in Na) {
         forall (i in M : mathprogram2.X[i,j] = 1)
            Assignment[i] := j;
      }

//    cout << "objective value after step 2b is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 2b is : " << mathprogram2.getTime() << endl;

      mathprogram2.reset();

   } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_three_one};
N := {j | j in N23};
Na := {k | k in N: (12 <= ord(k) < 16)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

   repeat {

   S := {};

      repeat {
```

```
      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 2c is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 2c is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_four_one};
N := {j | j in N24};
Na := {k | k in N: (16 <= ord(k) < 20)};
Ma := Opoints diff M;


R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;
    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 2d is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 2d is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


OB2 := sum (ordered i,j in points)
    (od[i,j] - Nd[Assignment[i],Assignment[j]]) * (od[i,j] - Nd[Assignment[i],Assignment[j]]);

cout << "The objective function after step 2 is: " << OB2 << endl;


Model mathprogram3 ("step32b.mod") editMode;

// local search

repeat {


  OB1 := OB2;

  OB2 := 0;

  M := { i | i in points : Assignment[i] = latpt_one_two \/ Assignment[i] = latpt_two_two \/
Assignment[i] = latpt_three_two \/Assignment[i] = latpt_four_two};
  Ma := Opoints diff M;
  N := {j | j in N24};
  Na := {};
  R := { i | i in M};

  if card(M) >= 1 then {

    forall (i in M)
      NN[i] := {j|j in N : ((xcoorn[j] = xcoorn[Assignment[i]]) \/ (xcoorn[j] = xcoorn[Assignment[i]] -
8) \/ (xcoorn[j] =  xcoorn[Assignment[i]] + 8))
                & ((ycoorn[j] = ycoorn[Assignment[i]]) \/ (ycoorn[j] = ycoorn[Assignment[i]] - 8) \/
(ycoorn[j] =  ycoorn[Assignment[i]] + 8))};

    forall (i in M)
      Na := Na union NN[i];

    repeat {
      S := {};

      repeat {

        minimum := min (i in R) ord(i);

        Q := {i| i in R : ord(i) = minimum};

        S := S union Q;

        R := R diff Q;

      } until card(S) >= 20 \/ card(R) = 0;

      M := {i| i in S};

      Ma := Opoints diff M;

      mathprogram3.solve();

      forall(j in Na){
        forall (i in M : mathprogram3.X[i,j] = 1)
          Assignment[i] := j;
      }

      mathprogram3.reset();

    } until card(R) = 0;
  }


  M := { i | i in points : Assignment[i] = latpt_five_two \/ Assignment[i] = latpt_six_two \/
Assignment[i] = latpt_seven_two \/Assignment[i] = latpt_eight_two};
  Ma := Opoints diff M;
  N := {j | j in N24};
  Na := {};
  R := { i | i in M};

  if card (M) >= 1 then {

    forall (i in M)

      NN[i] := {j|j in N : ((xcoorn[j] = xcoorn[Assignment[i]]) \/ (xcoorn[j] = xcoorn[Assignment[i]] -
8) \/ (xcoorn[j] =  xcoorn[Assignment[i]] + 8))
                & ((ycoorn[j] = ycoorn[Assignment[i]]) \/ (ycoorn[j] = ycoorn[Assignment[i]] - 8) \/
(ycoorn[j] =  ycoorn[Assignment[i]] + 8))};

    forall (i in M)
      Na := Na union NN[i];

    repeat {

      S := {};

      repeat {
        minimum := min (i in R) ord(i);

        Q := {i| i in R : ord(i) = minimum};

        S := S union Q;

        R := R diff Q;

      } until card(S) >= 20 ∨ card(R) = 0;

      M := {i| i in S};

      Ma := Opoints diff M;

      mathprogram3.solve();

      forall(j in Na){
        forall (i in M : mathprogram3.X[i,j] = 1)
          Assignment[i] := j;
      }

      mathprogram3.reset();
    } until card(R) = 0;
  }


  M := { i | i in points : Assignment[i] = latpt_nine_two ∨ Assignment[i] = latpt_ten_two ∨
Assignment[i] = latpt_eleven_two ∨Assignment[i] = latpt_twelve_two};
  Ma := Opoints diff M;
  N := {j | j in N24};
  Na := {};
  R := { i | i in M};

  if card(M) >= 1 then {

    forall (i in M)
      NN[i] := {j|j in N : ((xcoorn[j] = xcoorn[Assignment[i]]) ∨ (xcoorn[j] = xcoorn[Assignment[i]] -
8) ∨ (xcoorn[j] =  xcoorn[Assignment[i]] + 8))
                & ((ycoorn[j] = ycoorn[Assignment[i]]) ∨ (ycoorn[j] = ycoorn[Assignment[i]] - 8) ∨
(ycoorn[j] =  ycoorn[Assignment[i]] + 8))};

    forall (i in M)
      Na := Na union NN[i];

    repeat {

      S := {};

      repeat {

        minimum := min (i in R) ord(i);

        Q := {i| i in R : ord(i) = minimum};

        S := S union Q;
        R := R diff Q;

      } until card(S) >= 20 ∨ card(R) = 0;

      M := {i| i in S};

      Ma := Opoints diff M;

      mathprogram3.solve();

      forall(j in Na){
        forall (i in M : mathprogram3.X[i,j] = 1)
          Assignment[i] := j;
      }

      mathprogram3.reset();
    } until card(R) = 0;
  }


  M := { i | i in points : Assignment[i] = latpt_thirteen_two ∨ Assignment[i] = latpt_fourteen_two ∨
Assignment[i] = latpt_fifteen_two ∨Assignment[i] = latpt_sixteen_two};
  Ma := Opoints diff M;
  N := {j | j in N24};
  Na := {};
  R := { i | i in M};

  if card(M) >= 1 then {

  forall (i in M)
    NN[i] := {j|j in N : ((xcoorn[j] = xcoorn[Assignment[i]]) ∨ (xcoorn[j] = xcoorn[Assignment[i]] - 8)
∨ (xcoorn[j] =  xcoorn[Assignment[i]] + 8))
                & ((ycoorn[j] = ycoorn[Assignment[i]]) ∨ (ycoorn[j] = ycoorn[Assignment[i]] - 8) ∨
(ycoorn[j] =  ycoorn[Assignment[i]] + 8))};

  forall (i in M)
    Na := Na union NN[i];

  repeat {

      S := {};

      repeat {

        minimum := min (i in R) ord(i);

        Q := {i| i in R : ord(i) = minimum};

        S := S union Q;

        R := R diff Q;

      } until card(S) >= 20 ∨ card(R) = 0;

      M := {i| i in S};
      Ma := Opoints diff M;

      mathprogram3.solve();

      forall(j in Na){
        forall (i in M : mathprogram3.X[i,j] = 1)
          Assignment[i] := j;
      }

      mathprogram3.reset();
    } until card(R) = 0;
  }


OB2 := sum (ordered i,j in points)
    (od[i,j] - Nd[Assignment[i],Assignment[j]]) * (od[i,j] - Nd[Assignment[i],Assignment[j]]);

} until OB1 = OB2;

cout << "The objective function after step 2 is: " << OB2 << endl;


// step 3

M := { i |i in points : Assignment[i] = latpt_one_two};
N := {j | j in N31};
Na := {k | k in N: ( 20 <= ord(k) < 24)};
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

//  cout << "objective value after step 3a is : " << mathprogram2.objectiveValue() << endl;
//    cout << " Time for step 3a is : " << mathprogram2.getTime() << endl;

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_two_two};
N := {j | j in N32};
Na := {k | k in N: ( 24 <= ord(k) < 28)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 3b is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 3b is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();

  } until card(R) = 0;
}
```

```
M := { i |i in points : Assignment[i] = latpt_three_two};
N := {j | j in N33};
Na := {k | k in N: (28 <= ord(k) < 32)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

  S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 3c is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 3c is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_four_two};
N := {j | j in N34};
Na := {k | k in N: (32 <= ord(k) < 36)};
Ma := Opoints diff M;


R := {i| i in M};

if card(M) >= 1 then {
  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//  cout << "objective value after step 3d is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 3d is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_five_two};
N := {j | j in N35};
Na := {k | k in N: ( 36 <= ord(k) < 40)};
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;
    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

//  cout << "objective value after step 3e is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 3e is : " << mathprogram2.getTime() << endl;

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_six_two};
N := {j | j in N36};
Na := {k | k in N: ( 40 <= ord(k) < 44)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
```

```
        Assignment[i] := j;
    }

//  cout << "objective value after step 3f is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 3f is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_seven_two};
N := {j | j in N37};
Na := {k | k in N: (44 <= ord(k) < 48)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

  S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 ∨ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//  cout << "objective value after step 3g is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 3g is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}
```

```
M := { i |i in points : Assignment[i] = latpt_eight_two};
N := {j | j in N38};
Na := {k | k in N: (48 <= ord(k) < 52)};
Ma := Opoints diff M;


R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 3h is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 3h is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_nine_two};
N := {j | j in N39};
Na := {k | k in N: ( 52 <= ord(k) < 56)};
R := {i| i in M};

if card(M) >= 1 then {

  repeat {
    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

//    cout << "objective value after step 3i is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 3i is : " << mathprogram2.getTime() << endl;

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_ten_two};
N := {j | j in N310};
Na := {k | k in N: ( 56 <= ord(k) < 60)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;
    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 3j is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 3j is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_eleven_two};
N := {j | j in N311};
Na := {k | k in N: (60 <= ord(k) < 64)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

  S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
```

```
        Assignment[i] := j;
    }

//  cout << "objective value after step 3k is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 3k is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_twelve_two};
N := {j | j in N312};
Na := {k | k in N: (64 <= ord(k) < 68)};
Ma := Opoints diff M;


R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//  cout << "objective value after step 3l is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 3l is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}
```

```
M := { i |i in points : Assignment[i] = latpt_thirteen_two};
N := {j | j in N313};
Na := {k | k in N: ( 68 <= ord(k) < 72)};
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

//  cout << "objective value after step 3m is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 3m is : " << mathprogram2.getTime() << endl;

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_fourteen_two};
N := {j | j in N314};
Na := {k | k in N: ( 72 <= ord(k) < 76)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};
    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 3n is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 3n is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_fifteen_two};
N := {j | j in N315};
Na := {k | k in N: (76 <= ord(k) < 80)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

  S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;
    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//  cout << "objective value after step 3o is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 3o is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_sixteen_two};
N := {j | j in N316};
Na := {k | k in N: (80 <= ord(k) < 84)};
Ma := Opoints diff M;


R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
```

```
          Assignment[i] := j;
      }

//    cout << "objective value after step 3p is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 3p is : " << mathprogram2.getTime() << endl;

      mathprogram2.reset();
   } until card(R) = 0;
}

OB2 := sum (ordered i,j in points)
      (od[i,j] - Nd[Assignment[i],Assignment[j]]) * (od[i,j] - Nd[Assignment[i],Assignment[j]]);

cout << "The objective function after step 3 is: " << OB2 << endl;


// local search

repeat {

   OB1 := OB2;

   OB2 := 0;

   M := { i | i in points : Assignment[i] = latpt_one_three \/ Assignment[i] = latpt_two_three \/
Assignment[i] = latpt_three_three \/ Assignment[i] = latpt_four_three};
   Ma := Opoints diff M;
   N := {j | j in N316};
   Na := {};

   if card(M) >= 1 then {

      forall (i in M)
         NN[i] := {j|j in N : ((xcoorn[j] = xcoorn[Assignment[i]]) \/ (xcoorn[j] = xcoorn[Assignment[i]] -
4) \/ (xcoorn[j] =  xcoorn[Assignment[i]] + 4))
                     & ((ycoorn[j] = ycoorn[Assignment[i]]) \/ (ycoorn[j] = ycoorn[Assignment[i]] - 4) \/
(ycoorn[j] =  ycoorn[Assignment[i]] + 4))};

      forall (i in M)
         Na := Na union NN[i];

      mathprogram3.solve();

      forall(j in Na){
         forall (i in M : mathprogram3.X[i,j] = 1)
            Assignment[i] := j;
      }

      mathprogram3.reset();
   }


   M := { i | i in points : Assignment[i] = latpt_five_three \/ Assignment[i] = latpt_six_three \/
Assignment[i] = latpt_seven_three \/Assignment[i] = latpt_eight_three};
   Ma := Opoints diff M;
```

```
  N := {j | j in N316};
  Na := {};

  if card (M) >= 1 then {

    forall (i in M)

      NN[i] := {j|j in N : ((xcoorn[j] = xcoorn[Assignment[i]]) \/ (xcoorn[j] = xcoorn[Assignment[i]] -
4) \/ (xcoorn[j] =  xcoorn[Assignment[i]] + 4))
                & ((ycoorn[j] = ycoorn[Assignment[i]]) \/ (ycoorn[j] = ycoorn[Assignment[i]] - 4) \/
(ycoorn[j] =  ycoorn[Assignment[i]] + 4))};

    forall (i in M)
      Na := Na union NN[i];

    mathprogram3.solve();

    forall(j in Na){
      forall (i in M : mathprogram3.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram3.reset();
  }


  M := { i | i in points : Assignment[i] = latpt_nine_three \/ Assignment[i] = latpt_ten_three \/
Assignment[i] = latpt_eleven_three \/Assignment[i] = latpt_twelve_three};
  Ma := Opoints diff M;
  N := {j | j in N316};
  Na := {};

  if card(M) >= 1 then {

    forall (i in M)
      NN[i] := {j|j in N : ((xcoorn[j] = xcoorn[Assignment[i]]) \/ (xcoorn[j] = xcoorn[Assignment[i]] -
4) \/ (xcoorn[j] =  xcoorn[Assignment[i]] + 4))
                & ((ycoorn[j] = ycoorn[Assignment[i]]) \/ (ycoorn[j] = ycoorn[Assignment[i]] - 4) \/
(ycoorn[j] =  ycoorn[Assignment[i]] + 4))};

    forall (i in M)
      Na := Na union NN[i];

    mathprogram3.solve();

    forall(j in Na){
      forall (i in M : mathprogram3.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram3.reset();
  }


  M := { i | i in points : Assignment[i] = latpt_thirteen_three \/ Assignment[i] = latpt_fourteen_three \/
Assignment[i] = latpt_fifteen_three \/ Assignment[i] = latpt_sixteen_three};
```

```
  Ma := Opoints diff M;
  N := {j | j in N316};
  Na := {};

  if card(M) >= 1 then {

    forall (i in M)
      NN[i] := {j|j in N : ((xcoorn[j] = xcoorn[Assignment[i]]) \/ (xcoorn[j] = xcoorn[Assignment[i]] -
4) \/ (xcoorn[j] =  xcoorn[Assignment[i]] + 4))
                  & ((ycoorn[j] = ycoorn[Assignment[i]]) \/ (ycoorn[j] = ycoorn[Assignment[i]] - 4) \/
(ycoorn[j] =  ycoorn[Assignment[i]] + 4))};

    forall (i in M)
      Na := Na union NN[i];

    mathprogram3.solve();

    forall(j in Na){
      forall (i in M : mathprogram3.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram3.reset();

  }

  M := { i | i in points : Assignment[i] = latpt_seventeen_three \/ Assignment[i] = latpt_eighteen_three
\/ Assignment[i] = latpt_nineteen_three \/Assignment[i] = latpt_twenty_three};
  Ma := Opoints diff M;
  N := {j | j in N316};
  Na := {};

  if card(M) >= 1 then {

    forall (i in M)
      NN[i] := {j|j in N : ((xcoorn[j] = xcoorn[Assignment[i]]) \/ (xcoorn[j] = xcoorn[Assignment[i]] -
4) \/ (xcoorn[j] =  xcoorn[Assignment[i]] + 4))
                  & ((ycoorn[j] = ycoorn[Assignment[i]]) \/ (ycoorn[j] = ycoorn[Assignment[i]] - 4) \/
(ycoorn[j] =  ycoorn[Assignment[i]] + 4))};

    forall (i in M)
      Na := Na union NN[i];

    mathprogram3.solve();

    forall(j in Na){
      forall (i in M : mathprogram3.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram3.reset();

  }
  M := { i | i in points : Assignment[i] = latpt_twentyone_three \/ Assignment[i] =
latpt_twentytwo_three \/ Assignment[i] = latpt_twentythree_three \/Assignment[i] =
latpt_twentyfour_three};
  Ma := Opoints diff M;
  N := {j | j in N316};
  Na := {};

  if card (M) >= 1 then {

    forall (i in M)

      NN[i] := {j|j in N : ((xcoorn[j] = xcoorn[Assignment[i]]) \/ (xcoorn[j] = xcoorn[Assignment[i]] -
4) \/ (xcoorn[j] =  xcoorn[Assignment[i]] + 4))
                & ((ycoorn[j] = ycoorn[Assignment[i]]) \/ (ycoorn[j] = ycoorn[Assignment[i]] - 4) \/
(ycoorn[j] =  ycoorn[Assignment[i]] + 4))};

    forall (i in M)
      Na := Na union NN[i];

    mathprogram3.solve();

    forall(j in Na){
      forall (i in M : mathprogram3.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram3.reset();
  }


  M := { i | i in points : Assignment[i] = latpt_twentyfive_three \/ Assignment[i] =
latpt_twentysix_three \/ Assignment[i] = latpt_twentyseven_three \/Assignment[i] =
latpt_twentyeight_three};
  Ma := Opoints diff M;
  N := {j | j in N316};
  Na := {};

  if card(M) >= 1 then {

    forall (i in M)
      NN[i] := {j|j in N : ((xcoorn[j] = xcoorn[Assignment[i]]) \/ (xcoorn[j] = xcoorn[Assignment[i]] -
4) \/ (xcoorn[j] =  xcoorn[Assignment[i]] + 4))
                & ((ycoorn[j] = ycoorn[Assignment[i]]) \/ (ycoorn[j] = ycoorn[Assignment[i]] - 4) \/
(ycoorn[j] =  ycoorn[Assignment[i]] + 4))};

    forall (i in M)
      Na := Na union NN[i];

    mathprogram3.solve();

    forall(j in Na){
      forall (i in M : mathprogram3.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram3.reset();
  }


  M := { i | i in points : Assignment[i] = latpt_twentynine_three \/ Assignment[i] = latpt_thirty_three \/
Assignment[i] = latpt_thirtyone_three \/Assignment[i] = latpt_thirtytwo_three};
  Ma := Opoints diff M;
  N := {j | j in N316};
  Na := {};

  if card(M) >= 1 then {

  forall (i in M)
    NN[i] := {j|j in N : ((xcoorn[j] = xcoorn[Assignment[i]]) \/ (xcoorn[j] = xcoorn[Assignment[i]] - 4)
\/ (xcoorn[j] =  xcoorn[Assignment[i]] + 4))
                & ((ycoorn[j] = ycoorn[Assignment[i]]) \/ (ycoorn[j] = ycoorn[Assignment[i]] - 4) \/
(ycoorn[j] =  ycoorn[Assignment[i]] + 4))};

  forall (i in M)
    Na := Na union NN[i];

  mathprogram3.solve();

  forall(j in Na){
    forall (i in M : mathprogram3.X[i,j] = 1)
      Assignment[i] := j;
  }

  mathprogram3.reset();
  }

  M := { i | i in points : Assignment[i] = latpt_thirtythree_three \/ Assignment[i] =
latpt_thirtyfour_three \/ Assignment[i] = latpt_thirtyfive_three \/Assignment[i] =
latpt_thirtysix_three};
  Ma := Opoints diff M;
  N := {j | j in N316};
  Na := {};

  if card(M) >= 1 then {

    forall (i in M)
      NN[i] := {j|j in N : ((xcoorn[j] = xcoorn[Assignment[i]]) \/ (xcoorn[j] = xcoorn[Assignment[i]] -
4) \/ (xcoorn[j] =  xcoorn[Assignment[i]] + 4))
                & ((ycoorn[j] = ycoorn[Assignment[i]]) \/ (ycoorn[j] = ycoorn[Assignment[i]] - 4) \/
(ycoorn[j] =  ycoorn[Assignment[i]] + 4))};

    forall (i in M)
      Na := Na union NN[i];

    mathprogram3.solve();

    forall(j in Na){
      forall (i in M : mathprogram3.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram3.reset();
  }


  M := { i | i in points : Assignment[i] = latpt_thirtyseven_three \/ Assignment[i] =
latpt_thirtyeight_three \/ Assignment[i] = latpt_thirtynine_three \/Assignment[i] = latpt_forty_three};
  Ma := Opoints diff M;
  N := {j | j in N316};
  Na := {};

  if card (M) >= 1 then {

    forall (i in M)

      NN[i] := {j|j in N : ((xcoorn[j] = xcoorn[Assignment[i]]) \/ (xcoorn[j] = xcoorn[Assignment[i]] -
4) \/ (xcoorn[j] =  xcoorn[Assignment[i]] + 4))
                & ((ycoorn[j] = ycoorn[Assignment[i]]) \/ (ycoorn[j] = ycoorn[Assignment[i]] - 4) \/
(ycoorn[j] =  ycoorn[Assignment[i]] + 4))};

    forall (i in M)
      Na := Na union NN[i];

    mathprogram3.solve();

    forall(j in Na){
      forall (i in M : mathprogram3.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram3.reset();
  }


  M := { i | i in points : Assignment[i] = latpt_fortyone_three \/ Assignment[i] = latpt_fortytwo_three \/
Assignment[i] = latpt_fortythree_three \/Assignment[i] = latpt_fortyfour_three};
  Ma := Opoints diff M;
  N := {j | j in N316};
  Na := {};

  if card(M) >= 1 then {

    forall (i in M)
      NN[i] := {j|j in N : ((xcoorn[j] = xcoorn[Assignment[i]]) \/ (xcoorn[j] = xcoorn[Assignment[i]] -
4) \/ (xcoorn[j] =  xcoorn[Assignment[i]] + 4))
                & ((ycoorn[j] = ycoorn[Assignment[i]]) \/ (ycoorn[j] = ycoorn[Assignment[i]] - 4) \/
(ycoorn[j] =  ycoorn[Assignment[i]] + 4))};

    forall (i in M)
      Na := Na union NN[i];

    mathprogram3.solve();

    forall(j in Na){
      forall (i in M : mathprogram3.X[i,j] = 1)
        Assignment[i] := j;
    }
    mathprogram3.reset();
  }


  M := { i | i in points : Assignment[i] = latpt_fortyfive_three \/ Assignment[i] = latpt_fortysix_three \/
Assignment[i] = latpt_fortyseven_three \/Assignment[i] = latpt_fortyeight_three};
  Ma := Opoints diff M;
  N := {j | j in N316};
  Na := {};

  if card(M) >= 1 then {

  forall (i in M)
    NN[i] := {j|j in N : ((xcoorn[j] = xcoorn[Assignment[i]]) \/ (xcoorn[j] = xcoorn[Assignment[i]] - 4)
\/ (xcoorn[j] =  xcoorn[Assignment[i]] + 4))
                & ((ycoorn[j] = ycoorn[Assignment[i]]) \/ (ycoorn[j] = ycoorn[Assignment[i]] - 4) \/
(ycoorn[j] =  ycoorn[Assignment[i]] + 4))};

  forall (i in M)
    Na := Na union NN[i];

  mathprogram3.solve();

  forall(j in Na){
    forall (i in M : mathprogram3.X[i,j] = 1)
      Assignment[i] := j;
  }

  mathprogram3.reset();
  }


  M := { i | i in points : Assignment[i] = latpt_fortynine_three \/ Assignment[i] = latpt_fifty_three \/
Assignment[i] = latpt_fiftyone_three \/Assignment[i] = latpt_fiftytwo_three};
  Ma := Opoints diff M;
  N := {j | j in N316};
  Na := {};

  if card(M) >= 1 then {

    forall (i in M)
      NN[i] := {j|j in N : ((xcoorn[j] = xcoorn[Assignment[i]]) \/ (xcoorn[j] = xcoorn[Assignment[i]] -
4) \/ (xcoorn[j] =  xcoorn[Assignment[i]] + 4))
                & ((ycoorn[j] = ycoorn[Assignment[i]]) \/ (ycoorn[j] = ycoorn[Assignment[i]] - 4) \/
(ycoorn[j] =  ycoorn[Assignment[i]] + 4))};

    forall (i in M)
      Na := Na union NN[i];

    mathprogram3.solve();

    forall(j in Na){
      forall (i in M : mathprogram3.X[i,j] = 1)
        Assignment[i] := j;
    }
    mathprogram3.reset();
  }


  M := { i | i in points : Assignment[i] = latpt_fiftythree_three \/ Assignment[i] = latpt_fiftyfour_three
\/ Assignment[i] = latpt_fiftyfive_three \/Assignment[i] = latpt_fiftysix_three};
  Ma := Opoints diff M;
  N := {j | j in N316};
  Na := {};

  if card (M) >= 1 then {

    forall (i in M)

      NN[i] := {j|j in N : ((xcoorn[j] = xcoorn[Assignment[i]]) \/ (xcoorn[j] = xcoorn[Assignment[i]] -
4) \/ (xcoorn[j] =  xcoorn[Assignment[i]] + 4))
                & ((ycoorn[j] = ycoorn[Assignment[i]]) \/ (ycoorn[j] = ycoorn[Assignment[i]] - 4) \/
(ycoorn[j] =  ycoorn[Assignment[i]] + 4))};

    forall (i in M)
      Na := Na union NN[i];

    mathprogram3.solve();

    forall(j in Na){
      forall (i in M : mathprogram3.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram3.reset();
  }


  M := { i | i in points : Assignment[i] = latpt_fiftyseven_three \/ Assignment[i] = latpt_fiftyeight_three
\/ Assignment[i] = latpt_fiftynine_three \/Assignment[i] = latpt_sixty_three};
  Ma := Opoints diff M;
  N := {j | j in N316};
  Na := {};

  if card(M) >= 1 then {

    forall (i in M)
      NN[i] := {j|j in N : ((xcoorn[j] = xcoorn[Assignment[i]]) \/ (xcoorn[j] = xcoorn[Assignment[i]] -
4) \/ (xcoorn[j] =  xcoorn[Assignment[i]] + 4))
                & ((ycoorn[j] = ycoorn[Assignment[i]]) \/ (ycoorn[j] = ycoorn[Assignment[i]] - 4) \/
(ycoorn[j] =  ycoorn[Assignment[i]] + 4))};

    forall (i in M)
      Na := Na union NN[i];

    mathprogram3.solve();

    forall(j in Na){
      forall (i in M : mathprogram3.X[i,j] = 1)
        Assignment[i] := j;
    }
    mathprogram3.reset();
  }


  M := { i | i in points : Assignment[i] = latpt_sixtyone_three \/ Assignment[i] = latpt_sixtytwo_three \/
Assignment[i] = latpt_sixtythree_three \/Assignment[i] = latpt_sixtyfour_three};
  Ma := Opoints diff M;
  N := {j | j in N316};
  Na := {};

  if card(M) >= 1 then {

  forall (i in M)
    NN[i] := {j|j in N : ((xcoorn[j] = xcoorn[Assignment[i]]) \/ (xcoorn[j] = xcoorn[Assignment[i]] - 4)
\/ (xcoorn[j] =  xcoorn[Assignment[i]] + 4))
                  & ((ycoorn[j] = ycoorn[Assignment[i]]) \/ (ycoorn[j] = ycoorn[Assignment[i]] - 4) \/
(ycoorn[j] =  ycoorn[Assignment[i]] + 4))};

  forall (i in M)
    Na := Na union NN[i];

  mathprogram3.solve();

  forall(j in Na){
    forall (i in M : mathprogram3.X[i,j] = 1)
      Assignment[i] := j;
  }

  mathprogram3.reset();
  }


OB2 := sum (ordered i,j in points)
    (od[i,j] - Nd[Assignment[i],Assignment[j]]) * (od[i,j] - Nd[Assignment[i],Assignment[j]]);

} until OB1 = OB2;

cout << "The objective function after step 3 is: " << OB2 << endl;


// step 4


M := { i |i in points : Assignment[i] = latpt_one_three};
N := {j | j in N41};
Na := {k | k in N: ( 84 <= ord(k) < 88)};
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {
      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

//  cout << "objective value after step 4a is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4a is : " << mathprogram2.getTime() << endl;

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_two_three};
N := {j | j in N42};
Na := {k | k in N: ( 88 <= ord(k) < 92)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;
    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//  cout << "objective value after step 4b is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4b is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_three_three};
N := {j | j in N43};
Na := {k | k in N: (92 <= ord(k) < 96)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

  S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }
```

```
//    cout << "objective value after step 4c is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4c is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_four_three};
N := {j | j in N44};
Na := {k | k in N: (96 <= ord(k) < 100)};
Ma := Opoints diff M;


R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 4d is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4d is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_five_three};
```

```
N := {j | j in N45};
Na := {k | k in N: ( 100 <= ord(k) < 104)};
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

//    cout << "objective value after step 4e is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4e is : " << mathprogram2.getTime() << endl;

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_six_three};
N := {j | j in N46};
Na := {k | k in N: ( 104 <= ord(k) < 108)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {
      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//  cout << "objective value after step 4f is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4f is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_seven_three};
N := {j | j in N47};
Na := {k | k in N: (108 <= ord(k) < 112)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

  S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;
    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//  cout << "objective value after step 4g is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4g is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_eight_three};
N := {j | j in N48};
Na := {k | k in N: (112 <= ord(k) < 116)};
Ma := Opoints diff M;


R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }
```

```
//  cout << "objective value after step 4h is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4h is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_nine_three};
N := {j | j in N49};
Na := {k | k in N: ( 116 <= ord(k) < 120)};
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

//  cout << "objective value after step 4i is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4i is : " << mathprogram2.getTime() << endl;

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_ten_three};
N := {j | j in N410};
```

```
Na := {k | k in N: ( 120 <= ord(k) < 124)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//  cout << "objective value after step 4j is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4j is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_eleven_three};
N := {j | j in N411};
Na := {k | k in N: (124 <= ord(k) < 128)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

  S := {};

    repeat {
      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 4k is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4k is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_twelve_three};
N := {j | j in N412};
Na := {k | k in N: (128 <= ord(k) < 132)};
Ma := Opoints diff M;


R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;
    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//  cout << "objective value after step 4l is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4l is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}

M := { i |i in points : Assignment[i] = latpt_thirteen_three};
N := {j | j in N413};
Na := {k | k in N: ( 132 <= ord(k) < 136)};
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

//  cout << "objective value after step 4m is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4m is : " << mathprogram2.getTime() << endl;

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_fourteen_three};
N := {j | j in N414};
Na := {k | k in N: ( 136 <= ord(k) < 140)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 4n is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4n is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_fifteen_three};
N := {j | j in N415};
Na := {k | k in N: (140 <= ord(k) < 144)};
```

```
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

  S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 4o is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4o is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_sixteen_three};
N := {j | j in N416};
Na := {k | k in N: (144 <= ord(k) < 148)};
Ma := Opoints diff M;


R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {
      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 4p is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4p is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_seventeen_three};
N := {j | j in N417};
Na := {k | k in N: ( 148 <= ord(k) < 152)};
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};
    Ma := Opoints diff M;

    mathprogram2.solve();

//  cout << "objective value after step 4a1 is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4a1 is : " << mathprogram2.getTime() << endl;

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_eighteen_three};
N := {j | j in N418};
Na := {k | k in N: ( 152 <= ord(k) < 156)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//  cout << "objective value after step 4b1 is : " << mathprogram2.objectiveValue() << endl;
//    cout << " Time for step 4b1 is : " << mathprogram2.getTime() << endl;

      mathprogram2.reset();

   } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_nineteen_three};
N := {j | j in N419};
Na := {k | k in N: (156 <= ord(k) < 160)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

   repeat {

   S := {};

      repeat {

         minimum := min (i in R) ord(i);

         Q := {i| i in R : ord(i) = minimum};

         S := S union Q;

         R := R diff Q;

      } until card(S) >= 10 \/ card(R) = 0;

      M := {i| i in S};

      Ma := Opoints diff M;

      mathprogram2.solve();

   forall (j in Na) {
         forall (i in M : mathprogram2.X[i,j] = 1)
            Assignment[i] := j;
      }

//    cout << "objective value after step 4c1 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4c1 is : " << mathprogram2.getTime() << endl;

      mathprogram2.reset();
   } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_twenty_three};
N := {j | j in N420};
Na := {k | k in N: (160 <= ord(k) < 164)};
Ma := Opoints diff M;
```

```
R := {i| i in M};

if card(M) >= 1 then {

   repeat {

      S := {};

      repeat {

         minimum := min (i in R) ord(i);

         Q := {i| i in R : ord(i) = minimum};

         S := S union Q;

         R := R diff Q;

      } until card(S) >= 10 \/ card(R) = 0;

      M := {i| i in S};

      Ma := Opoints diff M;

      mathprogram2.solve();

   forall (j in Na) {
         forall (i in M : mathprogram2.X[i,j] = 1)
            Assignment[i] := j;
      }

//    cout << "objective value after step 4d1 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4d1 is : " << mathprogram2.getTime() << endl;

      mathprogram2.reset();
   } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_twentyone_three};
N := {j | j in N421};
Na := {k | k in N: ( 164 <= ord(k) < 168)};
R := {i| i in M};

if card(M) >= 1 then {

   repeat {

      S := {};

      repeat {

         minimum := min (i in R) ord(i);
      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

//  cout << "objective value after step 4e1 is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4e1 is : " << mathprogram2.getTime() << endl;

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_twentytwo_three};
N := {j | j in N422};
Na := {k | k in N: ( 168 <= ord(k) < 172)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};
    Ma := Opoints diff M;

    mathprogram2.solve();

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//  cout << "objective value after step 4f1 is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4f1 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_twentythree_three};
N := {j | j in N423};
Na := {k | k in N: (172 <= ord(k) < 176)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

  S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//  cout << "objective value after step 4g1 is : " << mathprogram2.objectiveValue() << endl;
//    cout << " Time for step 4g1 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_twentyfour_three};
N := {j | j in N424};
Na := {k | k in N: (176 <= ord(k) < 180)};
Ma := Opoints diff M;


R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 4h1 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4h1 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_twentyfive_three};
N := {j | j in N425};
Na := {k | k in N: ( 180 <= ord(k) < 184)};
```

```
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

//    cout << "objective value after step 4i1 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4i1 is : " << mathprogram2.getTime() << endl;

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_twentysix_three};
N := {j | j in N426};
Na := {k | k in N: ( 184 <= ord(k) < 188)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);
      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//  cout << "objective value after step 4j1 is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4j1 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_twentyseven_three};
N := {j | j in N427};
Na := {k | k in N: (188 <= ord(k) < 192)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

  S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};
    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 4k1 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4k1 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_twentyeight_three};
N := {j | j in N428};
Na := {k | k in N: (192 <= ord(k) < 196)};
Ma := Opoints diff M;


R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 4l1 is : " << mathprogram2.objectiveValue() << endl;
//    cout << " Time for step 4l1 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}

M := { i |i in points : Assignment[i] = latpt_twentynine_three};
N := {j | j in N429};
Na := {k | k in N: ( 196 <= ord(k) < 200)};
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

//    cout << "objective value after step 4m1 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4m1 is : " << mathprogram2.getTime() << endl;

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_thirty_three};
N := {j | j in N430};
Na := {k | k in N: ( 200 <= ord(k) < 204)};
Ma := Opoints diff M;
R := {i| i in M};
```

```
if card(M) >= 1 then {

   repeat {

      S := {};

      repeat {

         minimum := min (i in R) ord(i);

         Q := {i| i in R : ord(i) = minimum};

         S := S union Q;

         R := R diff Q;

      } until card(S) >= 10 \/ card(R) = 0;

      M := {i| i in S};

      Ma := Opoints diff M;

      mathprogram2.solve();

      forall (j in Na) {
         forall (i in M : mathprogram2.X[i,j] = 1)
            Assignment[i] := j;
      }

//    cout << "objective value after step 4n1 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4n1 is : " << mathprogram2.getTime() << endl;

      mathprogram2.reset();

   } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_thirtyone_three};
N := {j | j in N431};
Na := {k | k in N: (204 <= ord(k) < 208)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

   repeat {

   S := {};

      repeat {

         minimum := min (i in R) ord(i);
      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 ∨ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 4o1 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4o1 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_thirtytwo_three};
N := {j | j in N432};
Na := {k | k in N: (208 <= ord(k) < 212)};
Ma := Opoints diff M;


R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 ∨ card(R) = 0;

    M := {i| i in S};
    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 4p1 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4p1 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}

M := { i |i in points : Assignment[i] = latpt_thirtythree_three};
N := {j | j in N433};
Na := {k | k in N: ( 212 <= ord(k) < 216)};
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

//    cout << "objective value after step 4a2 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4a2 is : " << mathprogram2.getTime() << endl;

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_thirtyfour_three};
N := {j | j in N434};
Na := {k | k in N: ( 216 <= ord(k) < 220)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//  cout << "objective value after step 4b2 is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4b2 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_thirtyfive_three};
N := {j | j in N435};
Na := {k | k in N: (220 <= ord(k) < 224)};
Ma := Opoints diff M;
R := {i| i in M};
```

```
if card(M) >= 1 then {

  repeat {

  S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 4c2 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4c2 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_thirtysix_three};
N := {j | j in N436};
Na := {k | k in N: (224 <= ord(k) < 228)};
Ma := Opoints diff M;


R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);
      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 ∨ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 4d2 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4d2 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_thirtyseven_three};
N := {j | j in N437};
Na := {k | k in N: ( 228 <= ord(k) < 232)};
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 ∨ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();
```

```
//  cout << "objective value after step 4e2 is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4e2 is : " << mathprogram2.getTime() << endl;

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_thirtyeight_three};
N := {j | j in N438};
Na := {k | k in N: ( 232 <= ord(k) < 236)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//  cout << "objective value after step 4f2 is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4f2 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
```

```
   } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_thirtynine_three};
N := {j | j in N439};
Na := {k | k in N: (236 <= ord(k) < 240)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

   repeat {

   S := {};

      repeat {

         minimum := min (i in R) ord(i);

         Q := {i| i in R : ord(i) = minimum};

         S := S union Q;

         R := R diff Q;

      } until card(S) >= 10 \/ card(R) = 0;

      M := {i| i in S};

      Ma := Opoints diff M;

      mathprogram2.solve();

   forall (j in Na) {
         forall (i in M : mathprogram2.X[i,j] = 1)
            Assignment[i] := j;
      }

//    cout << "objective value after step 4g2 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4g2 is : " << mathprogram2.getTime() << endl;

      mathprogram2.reset();
   } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_forty_three};
N := {j | j in N440};
Na := {k | k in N: (240 <= ord(k) < 244)};
Ma := Opoints diff M;


R := {i| i in M};
```

```
if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 4h2 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4h2 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_fortyone_three};
N := {j | j in N441};
Na := {k | k in N: ( 244 <= ord(k) < 248)};
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};
      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

//  cout << "objective value after step 4i2 is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4i2 is : " << mathprogram2.getTime() << endl;

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_fortytwo_three};
N := {j | j in N442};
Na := {k | k in N: ( 248 <= ord(k) < 252)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();
```

```
      forall (j in Na) {
         forall (i in M : mathprogram2.X[i,j] = 1)
            Assignment[i] := j;
      }

//    cout << "objective value after step 4j2 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4j2 is : " << mathprogram2.getTime() << endl;

      mathprogram2.reset();

   } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_fortythree_three};
N := {j | j in N443};
Na := {k | k in N: (252 <= ord(k) < 256)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

   repeat {

   S := {};

      repeat {

         minimum := min (i in R) ord(i);

         Q := {i| i in R : ord(i) = minimum};

         S := S union Q;

         R := R diff Q;

      } until card(S) >= 10 \/ card(R) = 0;

      M := {i| i in S};

      Ma := Opoints diff M;

      mathprogram2.solve();

   forall (j in Na) {
         forall (i in M : mathprogram2.X[i,j] = 1)
            Assignment[i] := j;
      }

//    cout << "objective value after step 4k2 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4k2 is : " << mathprogram2.getTime() << endl;

      mathprogram2.reset();
   } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_fortyfour_three};
N := {j | j in N444};
Na := {k | k in N: (256 <= ord(k) < 260)};
Ma := Opoints diff M;


R := {i| i in M};

if card(M) >= 1 then {

   repeat {

      S := {};

      repeat {

         minimum := min (i in R) ord(i);

         Q := {i| i in R : ord(i) = minimum};

         S := S union Q;

         R := R diff Q;

      } until card(S) >= 10 ∨ card(R) = 0;

      M := {i| i in S};

      Ma := Opoints diff M;

      mathprogram2.solve();

   forall (j in Na) {
         forall (i in M : mathprogram2.X[i,j] = 1)
            Assignment[i] := j;
      }

//    cout << "objective value after step 4l2 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4l2 is : " << mathprogram2.getTime() << endl;

      mathprogram2.reset();
   } until card(R) = 0;
}

M := { i |i in points : Assignment[i] = latpt_fortyfive_three};
N := {j | j in N445};
Na := {k | k in N: ( 260 <= ord(k) < 264)};
R := {i| i in M};

if card(M) >= 1 then {
  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

//    cout << "objective value after step 4m2 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4m2 is : " << mathprogram2.getTime() << endl;

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_fortysix_three};
N := {j | j in N446};
Na := {k | k in N: ( 264 <= ord(k) < 268)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;
      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//  cout << "objective value after step 4n2 is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4n2 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_fortyseven_three};
N := {j | j in N447};
Na := {k | k in N: (268 <= ord(k) < 272)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

  S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();
  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 4o2 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4o2 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_fortyeight_three};
N := {j | j in N448};
Na := {k | k in N: (272 <= ord(k) < 276)};
Ma := Opoints diff M;


R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 4p2 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4p2 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}

M := { i |i in points : Assignment[i] = latpt_fortynine_three};
N := {j | j in N449};
Na := {k | k in N: ( 276 <= ord(k) < 280)};
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

//    cout << "objective value after step 4a3 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4a3 is : " << mathprogram2.getTime() << endl;

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_fifty_three};
N := {j | j in N450};
Na := {k | k in N: ( 280 <= ord(k) < 284)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {
    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//  cout << "objective value after step 4b2 is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4b2 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_fiftyone_three};
N := {j | j in N451};
Na := {k | k in N: (284 <= ord(k) < 288)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

  S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;
```

```
          R := R diff Q;

        } until card(S) >= 10 \/ card(R) = 0;

        M := {i| i in S};

        Ma := Opoints diff M;

        mathprogram2.solve();

    forall (j in Na) {
          forall (i in M : mathprogram2.X[i,j] = 1)
             Assignment[i] := j;
        }

//    cout << "objective value after step 4c2 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4c2 is : " << mathprogram2.getTime() << endl;

        mathprogram2.reset();
    } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_fiftytwo_three};
N := {j | j in N452};
Na := {k | k in N: (288 <= ord(k) < 292)};
Ma := Opoints diff M;


R := {i| i in M};

if card(M) >= 1 then {

    repeat {

        S := {};

        repeat {

          minimum := min (i in R) ord(i);

          Q := {i| i in R : ord(i) = minimum};

          S := S union Q;

          R := R diff Q;

        } until card(S) >= 10 \/ card(R) = 0;

        M := {i| i in S};

        Ma := Opoints diff M;

        mathprogram2.solve();
```

```
  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//  cout << "objective value after step 4d2 is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4d2 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_fiftythree_three};
N := {j | j in N453};
Na := {k | k in N: ( 292 <= ord(k) < 296)};
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

//  cout << "objective value after step 4e2 is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4e2 is : " << mathprogram2.getTime() << endl;

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram2.reset();

  } until card(R) = 0;
}
```

```
M := { i |i in points : Assignment[i] = latpt_fiftyfour_three};
N := {j | j in N454};
Na := {k | k in N: ( 296 <= ord(k) < 300)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 4f2 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4f2 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_fiftyfive_three};
N := {j | j in N455};
Na := {k | k in N: (300 <= ord(k) < 304)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {
```

```
  S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 4g2 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4g2 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_fiftysix_three};
N := {j | j in N456};
Na := {k | k in N: (304 <= ord(k) < 308)};
Ma := Opoints diff M;


R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;
      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 4h2 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4h2 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_fiftyseven_three};
N := {j | j in N457};
Na := {k | k in N: ( 308 <= ord(k) < 312)};
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

//    cout << "objective value after step 4i2 is : " << mathprogram2.objectiveValue() << endl;
//  cout << " Time for step 4i2 is : " << mathprogram2.getTime() << endl;

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_fiftyeight_three};
N := {j | j in N458};
Na := {k | k in N: ( 312 <= ord(k) < 316)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//  cout << "objective value after step 4j2 is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4j2 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();

  } until card(R) = 0;
}
```

```
M := { i |i in points : Assignment[i] = latpt_fiftynine_three};
N := {j | j in N459};
Na := {k | k in N: (316 <= ord(k) < 320)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

  S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 4k2 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4k2 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_sixty_three};
N := {j | j in N460};
Na := {k | k in N: (320 <= ord(k) < 324)};
Ma := Opoints diff M;


R := {i| i in M};

if card(M) >= 1 then {
  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 4l2 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4l2 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}

M := { i |i in points : Assignment[i] = latpt_sixtyone_three};
N := {j | j in N461};
Na := {k | k in N: ( 324 <= ord(k) < 328)};
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;
    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

//  cout << "objective value after step 4m2 is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4m2 is : " << mathprogram2.getTime() << endl;

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_sixtytwo_three};
N := {j | j in N462};
Na := {k | k in N: ( 328 <= ord(k) < 332)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

    forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//  cout << "objective value after step 4n2 is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4n2 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();

  } until card(R) = 0;
}


M := { i |i in points : Assignment[i] = latpt_sixtythree_three};
N := {j | j in N463};
Na := {k | k in N: (332 <= ord(k) < 336)};
Ma := Opoints diff M;
R := {i| i in M};

if card(M) >= 1 then {

  repeat {

  S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//  cout << "objective value after step 4o2 is : " << mathprogram2.objectiveValue() << endl;

//  cout << " Time for step 4o2 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}
```

```
M := { i |i in points : Assignment[i] = latpt_sixtyfour_three};
N := {j | j in N464};
Na := {k | k in N: (336 <= ord(k) < 340)};
Ma := Opoints diff M;


R := {i| i in M};

if card(M) >= 1 then {

  repeat {

    S := {};

    repeat {

      minimum := min (i in R) ord(i);

      Q := {i| i in R : ord(i) = minimum};

      S := S union Q;

      R := R diff Q;

    } until card(S) >= 10 \/ card(R) = 0;

    M := {i| i in S};

    Ma := Opoints diff M;

    mathprogram2.solve();

  forall (j in Na) {
      forall (i in M : mathprogram2.X[i,j] = 1)
        Assignment[i] := j;
    }

//    cout << "objective value after step 4p2 is : " << mathprogram2.objectiveValue() << endl;

//    cout << " Time for step 4p2 is : " << mathprogram2.getTime() << endl;

    mathprogram2.reset();
  } until card(R) = 0;
}


OB2 := sum (ordered i,j in points)
    (od[i,j] - Nd[Assignment[i],Assignment[j]]) * (od[i,j] - Nd[Assignment[i],Assignment[j]]);

cout << "The objective function after step 4 is: " << OB2 << endl;



// local search

repeat {
```

```
OB1 := OB2;

OB2 := 0;

counter := 0;


R := {i| i in points};

repeat {

  S := {};

  Na := {};

  repeat {

    minimum := min (i in R) ord(i);

    Q := {i| i in R : ord(i) = minimum};

    S := S union Q;

    R := R diff Q;

  } until card(S) >= 10 \/ card(R) = 0;

  M := {i| i in S};

  Ma := Opoints diff M;

  forall (i in M)
      NN[i] := {j|j in N : ((xcoorn[j] = xcoorn[Assignment[i]]) \/ (xcoorn[j] = xcoorn[Assignment[i]] -
2) \/ (xcoorn[j] =  xcoorn[Assignment[i]] + 2))
                 & ((ycoorn[j] = ycoorn[Assignment[i]]) \/ (ycoorn[j] = ycoorn[Assignment[i]] - 2) \/
(ycoorn[j] =  ycoorn[Assignment[i]] + 2))};

  forall (i in M)
    Na := Na union NN[i];

//   cout << "M = {";

//   forall (i in M ) {

//      cout << i << " " ;
//   }

//   cout << "}" << endl;

//   cout << " Na = { ";

//   forall (j in Na) {

//      cout << j << " " ;
//   }
//   cout << endl;

   mathprogram3.solve();

//   cout << " Time for step 1 is : " << mathprogram3.getTime() << endl;

   forall(j in Na){
      forall (i in M : mathprogram3.X[i,j] = 1)
         Assignment[i] := j;
   }

   mathprogram3.reset();

} until card(R) = 0;


OB2 := sum (ordered i,j in points)
      (od[i,j] - Nd[Assignment[i],Assignment[j]]) * (od[i,j] - Nd[Assignment[i],Assignment[j]]);

} until OB1 = OB2;

cout << "The final objective function is: " << OB2 << endl;


cout <<" The running time is: " << timer.getTime() - begintime << endl;

forall (i in points)

   cout << sqrt(1.5) * xcoorn[Assignment[i]] << "    " << sqrt(1.5) * ycoorn[Assignment[i]] << endl;

cout << endl;

forall (j in points)

   cout << sqrt(1.5) * xcoorn[Assignment[j]]  << endl;

cout << endl;

forall (k in points)

   cout << sqrt(1.5) * ycoorn[Assignment[k]] << endl;
```

Bibliography

E. Aarts and J. K. Lenstra (editors), *Local Search in Combinatorial Optimization*, Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley & Sons, West Sussex, England, 1997

W. P. Adams and T. Johnson. Improved linear programming based lower bounds for the quadratic assignment problem, in Quadratic Assignment and Related Problems, P. Pardalos and H. Wolkowicz (editors), *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 16, 43 – 75, 1994.

R. K. Ahuja, J. B. Orlin, and A. Tiwari, A greedy genetic algorithm for the quadratic assignment problem, *Computers & Operations Research* 27**,** 917 – 934, 2000.

M. Berry and G. Linoff, *Mastering Data Mining*, John Wiley & Sons, New York, 2000.

M. Berry and G. Linoff, *Data Mining Techniques: For Marketing, Sales, and Customer Relationship Management*, John Wiley & Sons, New York, 2004.

M. S. Bazaraa and H. D. Sherali, Benders' partitioning scheme applied to a new formulation of the quadratic assignment problem, *Naval Research Logistics Quarterly* 27, 29 – 41,1980.

I. Borg and P. Groenen, *Modern Multidimensional Scaling: Theory and Applications*, Springer, New York, 1997.

R.E. Burkard and F. Rendl, A thermodynamically motivated simulation procedure for combinatorial optimization problems, *European Journal of Operational Research* 17, 169 – 174, 1984.

R.E. Burkard, *Quadratic assignment problems*, European Journal of Operational Research 15, 283 – 289, 1984.

P. Cabena, P. Hadjinian, R. Stadler, J. Verhees, and A. Zanasi, *Discovering Data Mining: From Concept to Implementation*, Prentice Hall, Upper Saddle River, New Jersey, 1997.

E. Cela, *The Quadratic Assignment Problem: Theory and Algorithms*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1998.

N. Christofides and E. Benavent, An exact algorithm for the quadratic assignment problem, *Oper. Res.* 37, 760 – 768, 1989.

E. Condon, B. Golden, S. Lele, S. Raghavan, and E. Wasil, “A visualization method based on adjacency data,” *Decision Support Systems* 33, 349 – 362, 2002.

E. Condon, B. Golden, and E. Wasil, “Visualizing group decisions in the analytic hierarchy process,” *Computers & Operations Research* 30, 1435 – 1445, 2003.

K.C. Cox, S. G. Eick, G. J. Wills, and R. J. Brachman, “Visual data mining: recognizing telephone calling fraud,” *Data Mining and Knowledge Discovery* 1, 225 – 231, 1997.

L. Davis, *Handbook of Genetic Algorithm*, Van Nostrand Rienhold, New York, 1991.

G. Deboeck and T. Kohonen (editors), *Visual Explorations in Finance*, Springer, London, 1998.

U. Dorndorf and E. Pesch. Data Warehouses, in Handbook on Data Management in Information Systems, J. Blazewicz, W. Kubiak, T. Morzy, and M. Rusinkiewicz (editors),
International Handbooks on Information Systems, 387 – 430, Springer, New York, 2003.

J. Elder and D. Pregibon, A statistical perspective on knowledge discovery in databases, in *Advances in Knowledge Discovery and Data Mining*, U.M. Fayyad et al. (editors),
83 – 116, AAAI/MIT Press, California, 1996.

G. Finke, R.E. Burkard, and F. Rendl, Quadratic assignment problems, *Annals of Discrete Mathematics* 31, 61 – 82, 1987.

C. Fleurent and J.A. Ferland, Genetic hybrids for the quadratic assignment problem, in Quadratic Assignment and Related Problems, P. Pardalos and H. Wolkowicz (editors), *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 16,
173 – 187, 1994.

C. Fleurent and J.A. Ferland, Genetic and hybrid algorithms for graph coloring, *Annals of Operations Research* 63, 437 – 461, 1996.

J.H. Friedman, On bias, variance, 0/1-loss, and the curse-of-dimensionality, *Data Mining and Knowledge Discovery* 1, 55 – 78, 1997.

A. M. Frieze and J. Yadegar, On the quadratic assignment problem, *Discrete Applied Mathematics* 5, 89 – 98, 1983.

M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Co., San Francisco, 1979.

F. Glover and M. Laguna, *Tabu Search*, Kluwer Academic Publishers, Boston, Massachusetts, 1997.

P. V. Hentenryck, *The OPL Optimization Programming Language*, MIT Press, Cambridge, Massachusetts, 1999.

J.H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, 1975.

P. J. Huber, From large to huge: a statistician's reactions to KDD & DM. Proc. *Third International Conference on Knowledge Discovery and Data Mining*, 304 – 308, AAAI Press, California, 1997.

L. Kaufman and F. Broeckx, An algorithm for the quadratic assignment problem using Benders' decomposition, *European Journal of Operational Research* 2, 209 – 211, 1978.

T. C. Koopmans and M. J. Beckmann, Assignment problems and the location of economic activities, *Econometrica* 25, 53 – 76, 1957

J. B. Kruskal and M. Wish, *Multidimensional Scaling*, Sage, Beverly Hills, California, 1978.

E. L. Lawler, The quadratic assignment problem. *Manag. Sci*. 9, 586 – 599, 1963.

Y. Li, P. M. Pardalos, and M. G. C. Resende, A Greedy Randomized Adaptive Search Procedure for the Quadratic Assignment Problem, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 16, 237 – 261, 1994.

M. J. Mackinnon and N. Glick, Data Mining and Knowledge Discovery in Databases-An overview, *Austral. & New Zealand J. Statist*. 41, 255 – 275, 1999.

Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer, Berlin, 1996.

T. Morzy and M. Zakrzewicz, Data Mining, in *Handbook on Data Management in Information Systems*, J. Blazewicz, W. Kubiak, T. Morzy, and M. Rusinkiewicz (editors), *International Handbooks on Information Systems*, 487 – 566, Springer, New York, 2003.

M. W. Padberg and M. P. Rijal, *Location, Scheduling, Design and Integer Programming*, Kluwer Academic Publishers, Boston, 1996.

G. Rawlins (editor), *Foundations of Genetic Algorithms*, First Workshop on the Foundations of Genetic Algorithms and Classifier Systems, Morgan Kaufmann Publishers, San Mateo, California, 1991.

J. W. Sammon, “A nonlinear mapping for data structure analysis,” *IEEE Transactions on Computers* 18, 401 – 409, 1969.

S. S. Schiffman, M. L. Reynolds, and F. W. Young, *Introduction to Multidimensional Scaling; Theory, Methods, and Applications*, Academic Press, Orlando, Florida, 1981.

H. P. Schwefel, *Numerical Optimization for Computer Models*, John Wiley, Chichester,UK, 1981.

A. Shoshani, Multidimensionality in Statistical, OLAP, and Scientific Databases, in *Multidimensional Databases: Problems and Solutions*, M. Rafanelli(editor), 46 – 68, Idea Group Publishing, Hershey, Pennsylvania, 2003.

J. Skorin-Kapov, Tabu search applied to the quadratic assignment problem, *ORSA Journal on Computing* 2, 33 – 45, 1990.

D. M. Tate and A. E. Smith, A genetic approach to the quadratic assignment problem, *Computers & Operations Research* 22, 73 – 83, 1995.

E. Wegman, Huge data sets and the frontiers of computational feasibility, *J. Comput. Graphical Statist*. 4, 281 – 295, 1995.

W. S. Torgerson, *Theory and Methods of Scaling*, John Wiley, New York, 1958.

M.R. Wilhelm and T.L. Ward, Solving quadratic assignment problems by simulated annealing, *IEEE Trans*. 19, 107 – 119, 1987.

F. W. Young, *Multidimensional Scaling: History, Theory and Applications*, Lawrence Erlbaum Associates, London, England, 1987.