ABSTRACT

Title of Dissertation: Weight Annealing Heuristics for Solving Bin Packing and other Combinatorial Optimization Problems: Concepts, Algorithms, and Computational Results

Kok-Hua Loh, Doctor of Philosophy, 2006

Dissertation directed by : Professor Bruce L. Golden
Department of Decision and Information Technologies

The application of weight annealing to combinatorial optimization problems is relatively new, compared to applications of well-known optimization techniques such as simulated annealing and tabu search. The weight annealing approach seeks to expand a neighborhood search by creating distortions in different parts of the search space. Distortion is controlled through weight assignment based on insights gained from one iteration of the search procedure to the next with a view towards focusing computational efforts on the poorly solved regions of the search space. The search for the global optimum should be accelerated and the solution quality should be improved with weight annealing.

In this dissertation, we present key ideas behind weight annealing and develop algorithms that solve combinatorial optimization problems. Our weight annealing-based heuristics solve the one-dimensional bin packing problem and the two-dimensional bin packing problem with and without guillotine cutting and item orientation constraints. We also solve the maximum cardinality bin packing problem and the multidimensional multiple knapsack problem with our heuristics.

WEIGHT ANNEALING HEURISTICS FOR SOLVING BIN PACKING AND OTHER COMBINATORIAL OPTIMIZATION PROBLEMS: CONCEPTS, ALGORITHMS, AND COMPUTATIONAL RESULTS

by

Kok-Hua Loh

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
2006

Advisory Committee:

Professor Bruce L. Golden, Chair
Professor Edward A. Wasil
Associate Professor Gilvan Souza
Assistant Professor Wolfgang Jank
Professor Paul M. Schonfeld

Acknowledgements

I will like to express my sincere gratitude to the chair of my dissertation committee, Professor Bruce L Golden, for his guidance and support in the course of my research. I am greatly indebted to Professor Edward A. Wasil for his insightful comments and editorial assistance in the preparation of this dissertation. I thank Professors Paul M. Schonfeld, Gilvan Souza and Wolfgang Jank, for gladly serving on my dissertation committee.

I will like to dedicate this dissertation to my wife Audrey and our children, Yueling and Song-Yang, for believing in me, right from the start.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The application of the weight annealing concept to combinatorial optimization problems is relatively new, compared to applications of well-known optimization techniques such as simulated annealing and tabu search. When applied to a combinatorial optimization problem such as the one-dimensional bin packing problem, the weight annealing approach seeks to expand a neighborhood search by creating distortions in different parts of the search space. Distortion is controlled through weight assignment based on insights gained from one iteration of the search procedure to the next with a view towards focusing computational efforts on the poorly solved regions of the search space. The search for the global optimum is accelerated and the solution quality is improved.

In this dissertation, we present key ideas behind weight annealing and develop algorithms that solve the one-dimensional bin packing problem and the two-dimensional bin packing problem. We also develop weight annealing algorithms that solve the maximum cardinality bin packing problem and the multidimensional multiple knapsack problem.

In Chapter 2, we apply the weight annealing concept to the one-dimensional bin packing problem. In this problem, we need to pack different size items into identical bins of equal capacity. Our objective is to minimize the number of bins without violating the capacity constraints. We also present an algorithm to solve the dual bin packing problem. In this problem, we seek to minimize the maximum space utilization of any bin, given a

fixed number of bins. We conduct extensive computational tests with benchmark problems and compare the results produced by our weight annealing algorithm to the results produced by heuristics such as tabu search and exact algorithms that are found in the bin packing literature.

In Chapters 3 and 4, we develop weight annealing algorithms that solve the two-dimensional bin packing problem. Given a set of rectangular items with different widths and heights, we need to allocate all items to a minimum number of identical rectangular bins without allowing the items to overlap. The items have to be packed with their edges parallel to the edges of a bin. There are four variants of the two-dimensional bin packing problem:

- All items have fixed orientations and are obtained through a sequence of edge-to-edge guillotine cuts that are parallel to the edges of a bin. The guillotine constraint originates from the technological requirements of automated cutting machines.
- The orientations of the items may be rotated through 90˚ and guillotine cuts are required.
- The orientations of the items are fixed and free cutting is allowed. The edge-to-edge cutting constraint does not apply.
- The orientations of the items may be rotated through 90˚ and free cutting applies.

In Chapter 3, we develop a weight annealing algorithm that solves the guillotine cutting versions of the two-dimensional bin packing problem. We solve a set of benchmark problems with our weight annealing algorithms and show that the results

produced by weight annealing are comparable in terms of accuracy and computational speed to the best results found in the literature.

In Chapter 4, we develop a weight annealing algorithm that solves the free cutting versions of the two-dimensional bin packing problem. We present computational results that show our algorithm produces high-quality solutions quickly.

In Chapter 5, we consider the maximum cardinality bin packing problem. This problem is a special case of the multiple knapsack problem where all items have the same profit and all knapsacks (or bins) have the same capacity. The objective is to pack the maximum number of items into the available bins. We develop a weight annealing algorithm and compare the results produced by weight annealing with two other highly competitive algorithms found in the literature.

In Chapter 6, we consider the multidimensional knapsack problem. This is a standard knapsack problem with additional multiple resource (knapsack) constraints, or one constraint consisting of a multidimensional attribute. We develop a weight annealing algorithm and test it on a set of widely used benchmark problems found in the literature.

In Chapter 7, the weight annealing concept is applied to the multidimensional multiple knapsack problem. This is a generalization of the multiple knapsack problem in which every knapsack has multidimensional constraints. Since there are few benchmark problems in the open literature, we develop a set of instances that can be used to test the performance of solution procedures. We present computational results to show that our weight annealing algorithm produces high-quality solutions to the multidimensional multiple knapsack problem.

Chapter 2

The One-Dimensional Bin Packing Problem

2.1 Introduction

In the one-dimensional bin packing problem, we try to pack n items into identical bins. Each bin has a capacity C. Each item j in bin i has a space requirement t_{ij}. The objective is to minimize the number of bins without violating the capacity constraints.

The one-dimensional bin packing problem (1BP) is NP-hard (Garey and Johnson 1979). Over the years, many heuristics have been developed to find near-optimal solutions to the 1BP. A survey by Coffman, Garey, and Johnson (1997) gives a comprehensive review of approximation algorithms. First-fit decreasing (FFD) and best-fit decreasing (BFD) are two well-known approximation algorithms that can produce near-optimal solutions quickly.

The branch-and-bound procedure developed by Martello and Toth (1990) (denoted by MTP) is one of the earliest methods for solving the one-dimensional bin packing problem. MTP is the basic reference in many comparative studies. Scholl, Klein, and Jürgens (1997) developed a hybrid method (BISON) that combines a tabu search with a branch-and-bound procedure based on several bounds, and uses a new branching scheme. Schwerin and Wäscher (1999) devised a heuristic (MTPCS) that integrated into MTP a new lower bound L_{cs} that is based on the one-dimensional cutting stock problem. Fleszar and Hindi (2002) proposed a heuristic (PMBS′ + VNS) that was based on the minimum bin slack (MBS) heuristic of Gupta and Ho (1999) and the variable

neighborhood search (VNS) metaheuristic of Hansen and Mladenović (1999). Alvim, Ribeiro, Glover, and Alosie (2004) proposed a hybrid improvement heuristic (HI_BP) that is a sophisticated procedure involving the use of lower bound techniques, construction heuristics with reference to the dual min-max problem, load redistribution based on dominance, differencing, and unbalancing, and an improvement process using tabu search.

In this chapter, we propose a simple and fast heuristic based on the concept of weight annealing to solve the one-dimensional bin packing problem. In Section 2.2, we describe the concept of weight annealing in the context of a combinatorial optimization problem (the traveling salesman problem). In Section 2.3, we describe our local search procedures and apply weight annealing to the one-dimensional bin packing problem. We develop a version of the weight annealing algorithm for the one-dimensional bin packing problem (denoted by WA1BP) that minimizes the number of bins. In Section 2.4, we introduce a variation to WA1BP that solves the one-dimensional dual bin packing problem (1DBP). The objective in 1DBP is to minimize the maximum space utilization of any bin given a fixed number of bins. In Section 2.5, we conduct an extensive computational experiment with benchmark problems and compare the results of our weight annealing heuristic to results given in the bin packing literature.

2.2 The Concept of Weight Annealing

The application of weight annealing to combinatorial optimization problems is a recent development due to Elidan et al. (2002) and Ninio and Schneider (2005). Weight annealing shares many features with other metaheuristics such as simulated annealing

and deterministic annealing. In particular, simulated annealing and deterministic annealing start with an initial solution and perform a sequence of moves in the local search that include deteriorating moves. A deteriorating move may be allowed as long as the objective function value does not worsen beyond a specific threshold. Weight annealing not only considers the value of the objective function but also makes use of the information on how well every part of the search space is being solved at every stage of an optimization run. When applied to a combinatorial optimization problem like the one-dimensional bin packing problem, the weight annealing approach seeks to expand and speed up the neighborhood search by creating distortions in different parts of the search space. Distortion is controlled through weight assignments based on insights gained from one iteration to the next, with a view towards focusing computational efforts on the poorly solved regions. The search for the global optimum should be accelerated and the solution quality should be improved with weight annealing.

We illustrate the weight annealing approach by applying it to a traveling salesman problem (TSP). Let $G = (N,E)$ be an undirected graph, where N is the set of all nodes (cities) and E is the set of edges connecting these cities. Each edge $(i, j) \in E$ has a length d_{ij}. Given a fixed number of cities, the salesman must visit every city exactly once before returning back to the origin. In trying to solve the TSP, one can try a greedy approach, by successively picking the next city to be the one closest to the current city. However, towards the end of the tour, the salesman is likely to be penalized for being myopic and compelled to take a long trip to visit all remaining cities before returning back to the origin. With a greedy approach, the salesman may get trapped in a poor local optimum.

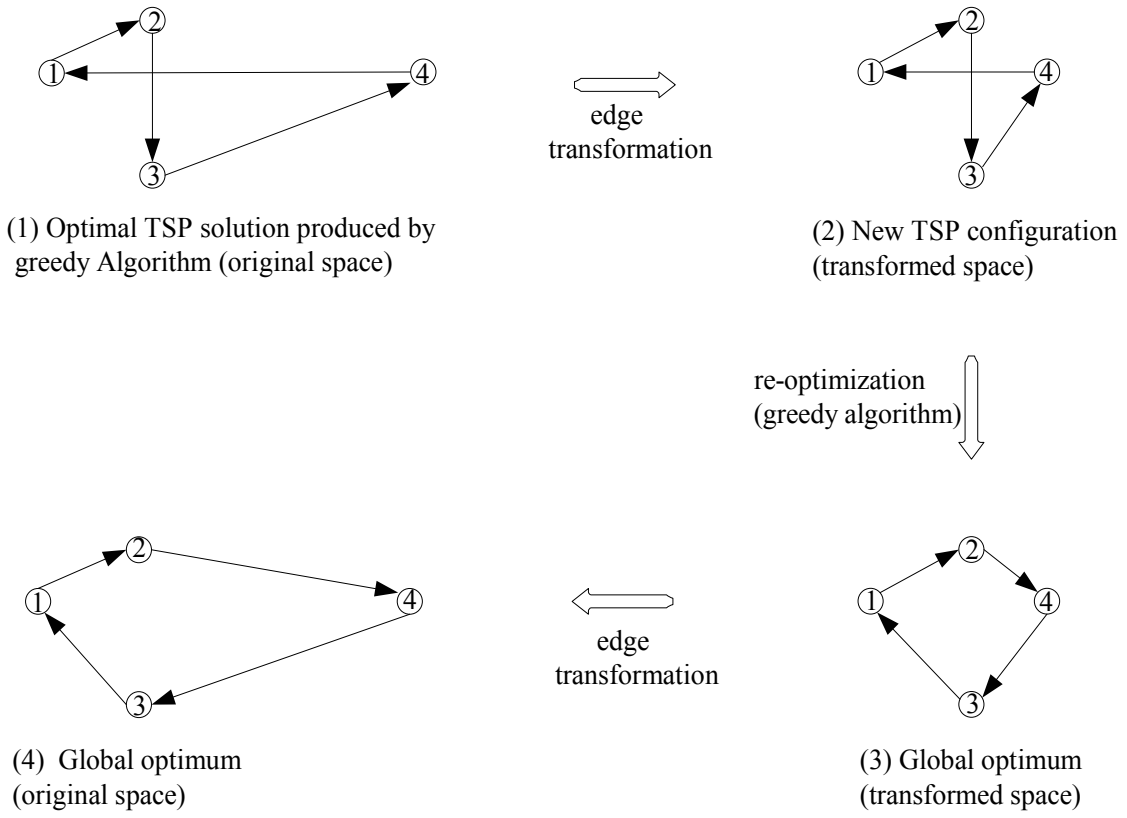


Figure 2.1 Solving a TSP with a greedy approach using weight annealing.

Consider the TSP shown in Figure 2.1. The solution produced by the greedy algorithm is not the global optimum. To facilitate an escape from a poor local minimum, weight annealing would significantly reduce the distance of the long return trip by appropriate transformations to the lengths of the edges. Cities on the return leg would be visited early due to the greedy nature of the algorithm. In Figure 2.1, node 4 can be visited early in the tour if we transform (reduce) the lengths of d_{24}, d_{14}, and d_{34}. Thus, appropriate weight assignments to the edges can alter the sequence of visits. In this example, with weight annealing, we are able to generate the global optimum when using the greedy algorithm.

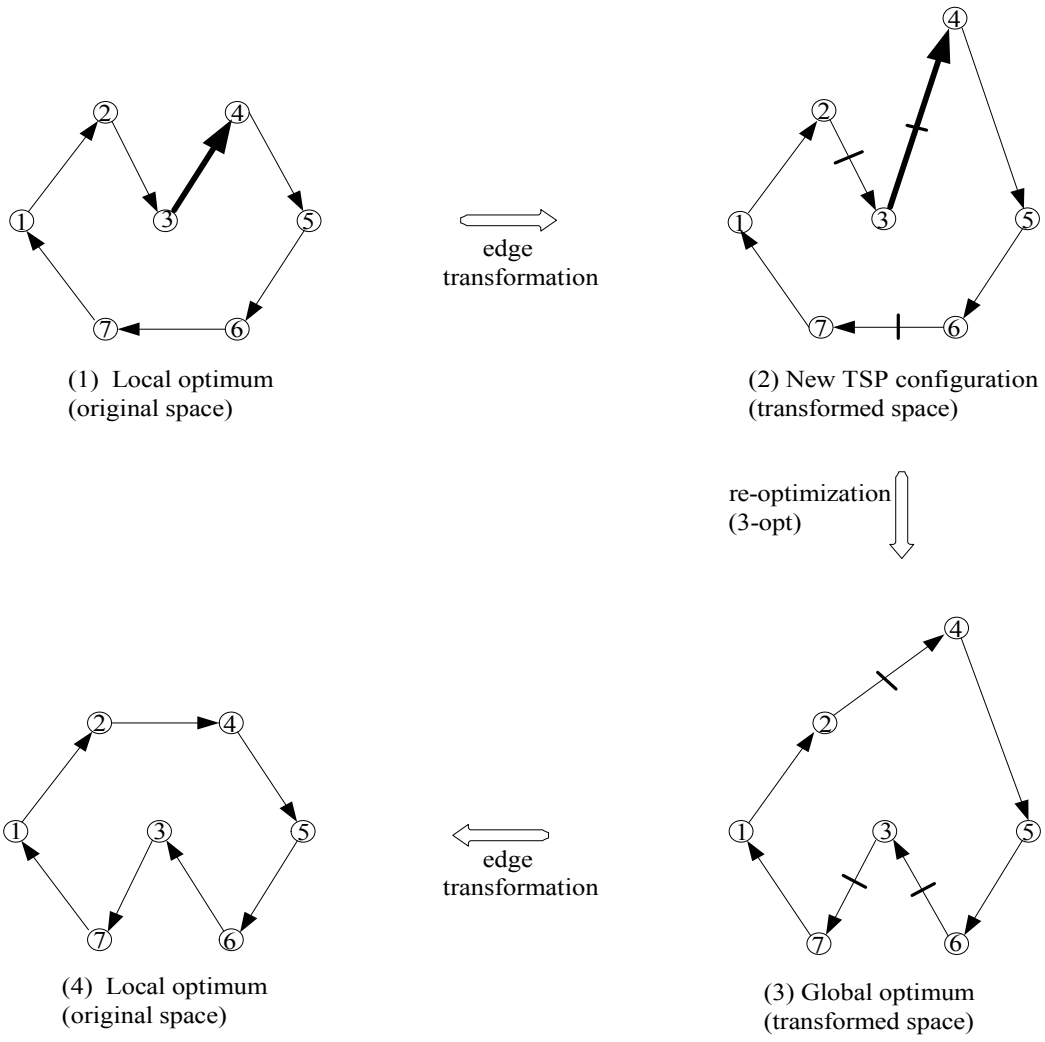


Figure 2.2 The use of weight annealing and 3-opt to prevent a targeted edge from appearing in an optimal solution.

Weight annealing allows deteriorating moves by sufficiently reducing the lengths of long edges through an appropriate transformation. These transformed edges are used in the current solution after re-optimization. Weight annealing can also prevent a specific edge from entering a solution. In Figure 2.2, the original tour (1,2,3,4,5,6,7,1) is a local minimum. With the 3-opt algorithm, we target edge (3,4) to be excluded from the next solution when using weight annealing. This is done by increasing the lengths of

d_{34} and d_{45}. After the length transformation, 3-opt replaces edges (3,4), (4,5) and (7,6) with shorter edges and generates the global minimum in the transformed space, which is a new local minimum in the original space. This example shows that we are able to attain two objectives: (1) Escaping from a poor local minimum and (2) Targeting edges to be excluded from a solution. With weight annealing, we can guide an algorithm to perform edge exchanges in certain regions of interest in the search space.

The variations in weights on the edges can be made large initially and then reduced according to a cooling schedule controlled by a temperature parameter *T*, as in simulated annealing. When the temperature is high, the re-weighting step has enough freedom to change the current configuration and increase the chances of escaping from a poor local minimum. As the temperature is lowered, the change of weights would be small, and the algorithm should converge to a good local optimum.

The earliest work on weight annealing is due to Ninio and Schneider (2005) and Elidan et al. (2002). Ninio and Schneider (2005) used weight annealing to solve the traveling salesman problem and the Sherrington-Kirkpartrick-model for spin glasses. They reported that weight annealing led to mostly better results than simulated annealing for the TSP (they solved five benchmark problems), but was inferior to simulated annealing in solving the Sherrington-Kirkpartrick instances. The authors demonstrated that weight annealing can be used to perturb the data to escape from local optimums in combinatorial optimization problems. The idea of weight annealing is similar to techniques such as search space smoothing (Coy, Golden, and Wasil 2000) and noising (Charon and Hudry 1993).

2.3 Outline of Weight Annealing Algorithm for Solving the One-Dimensional Bin Packing Problem

In this section, we provide the basic structure of a weight annealing algorithm for solving the one-dimensional bin packing problem. We denoted the algorithm by WA1BP. WA1BP has five steps.

Step 1. Start by constructing an initial solution using FFD algorithm.

Step 2. Based on the initial solution, compute and assign a set of weights to the bins and items to distort the item sizes according to residual capacities of their respective bins.

Step 3. Perform local search by swapping items between all pairs of bins, with an objective function that maximizes the sum of squares of bin loads. A bin load is defined as the sum of all item sizes in the bin.

Step 4. Perform re-weighting by determining a new set of weights based on the result of the previous optimization run.

Step 5. Return to Step 3 until some stopping criterion is reached.

2.3.1 Initial Solution

We use FFD to generate an initial solution. Given a list of items and a bin size *C,* we generate an initial solution in the following way:

- Sort the items in non-increasing order according to the item sizes.
- Place the first item on the list into the lowest numbered bin with sufficient residual capacity.
- If the item will not fit into any bin, open a new one for it.

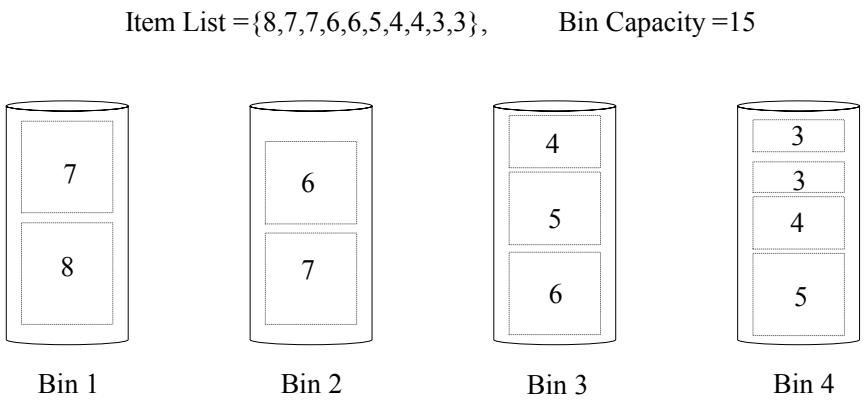


Figure 2.3 An example of applying the first-fit decreasing algorithm.

An approximation algorithm like FFD produces a solution that may result in unbalanced bin loads with the lower numbered bins having a larger proportion of large items, compared to the higher numbered bins. We show this type of situation in Figure 2.3.

2.3.2 Neighborhood Search for the One-Dimensional Bin Packing Problem

2.3.2.1 Objective Function

We consider an objective function that minimizes the number of bins of capacity C that we need to pack all the items without violating the capacity constraints. Our neighborhood search procedure is based on a series of moves (swapping items between bins) to improve the objective function value, by filling as many full bins as possible. Stated formally, the objective function maximizes the sum of squares of bin loads, where bin load l_i is defined as the sum of sizes of items in bin i, or $l_i = \sum_{j=1}^{q_i} t_{ij}$, where t_{ij} is the size of item j in bin i, and q_i is the number of items in bin i.

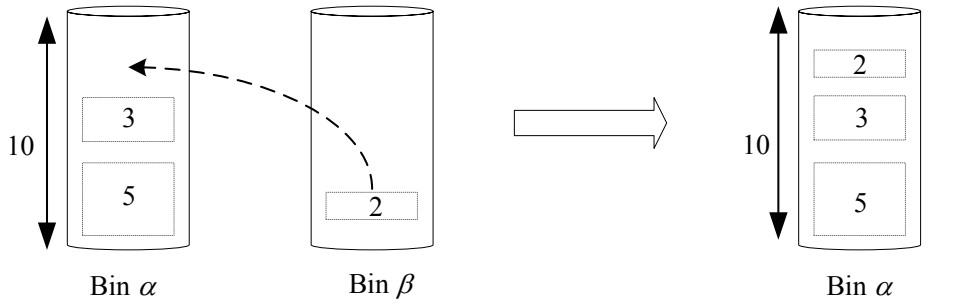


Figure 2.4 Maximizing the sum of squares of bin loads.

The objective function is given by

$$\text{Maximize} \quad f = \sum_{i=1}^{p} (l_i)^2 \tag{2.1}$$

where p is the number of bins in the current solution. Our objective function is motivated by the one developed by Fleszar and Hindi (2002) for the one-dimensional bin packing problem.

Consider the problem of packing three items with sizes 2, 3 and 5 into two bins with C = 10. In Figure 2.4, we show a solution that uses two bins and has an objective function value of 68. Given the objective function (2.1), the global optimum is 100, with all items packed into one bin. In the neighborhood search, the objective function (2.1) seeks to make moves that will maximize the sum of squares of the bin loads. This is equivalent to minimizing the number of bins that are needed to pack all of the items.

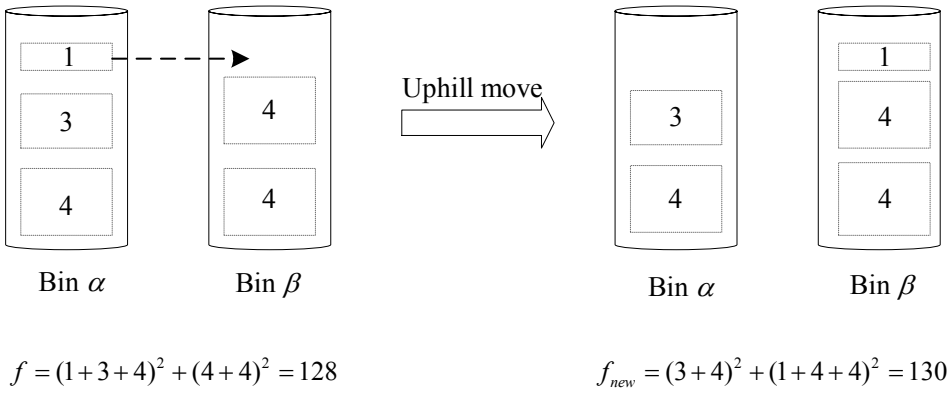


Figure 2.5 An example of Swap(1,0).

Solutions that are neighbors of a current solution are obtained by swapping items between all possible pairs of bins. Here, we propose four item exchange schemes that are analogous to the 2opt, Or-opt and 3-opt moves for the traveling salesman problem. The first two schemes were proposed by Fleszar and Hindi (2002).

2.3.2.2 Swap(1,0)

We consider moving one item from bin α to bin β, and then evaluating the change in objective function value for every item on the list for a possible uphill move.

In Figure 2.5, the uphill move is made because of the increase in f from 128 to 130. In our implementation, there is no need to compute the objective function values before and after a swap. We only need to evaluate the change in objective function value Δf as a result of moving item j with size $t_{\alpha j}$ from bin α to bin β. The change is given by

$$\Delta f = (l_\alpha - t_{\alpha j})^2 + (l_\beta + t_{\alpha j})^2 - l_\alpha^2 - l_\beta^2 \ . \qquad (2.2)$$

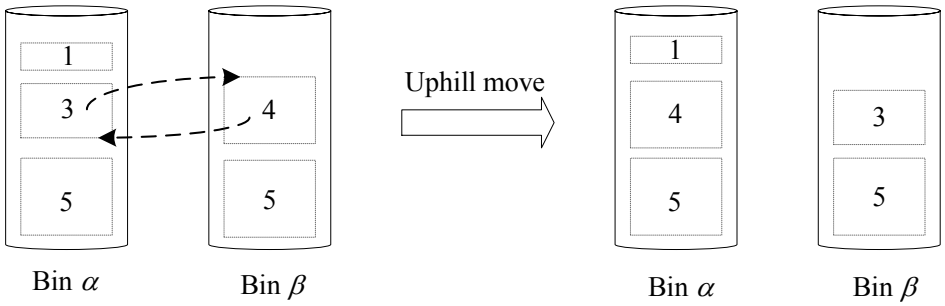


Figure 2.6 An example of Swap(1,1).

This significantly reduces the computational effort because there is no need to compute the actual value of the objective function at every evaluation step, as is the case for other procedures.

2.3.2.3 Swap(1,1)

We evaluate the change in the objective function value that results from swapping item *i* in bin α with item *j* in bin β. We make the move if it results in an increase in the objective function value. We evaluate all possible pairs *(i,j)* of items from all pairs of bins. An example of Swap(1,1) is given in Figure 2.6.

We need to evaluate only the change of objective function value Δf that results from swapping item *i* with size $t_{\alpha i}$ in bin α and item *j* with size $t_{\beta j}$ in bin β. The change is given by

$$\Delta f = (l_\alpha - t_{\alpha i} + t_{\beta j})^2 + (l_\beta - t_{\beta j} + t_{\alpha i})^2 - l_\alpha^2 - l_\beta^2 \ . \qquad (2.3)$$

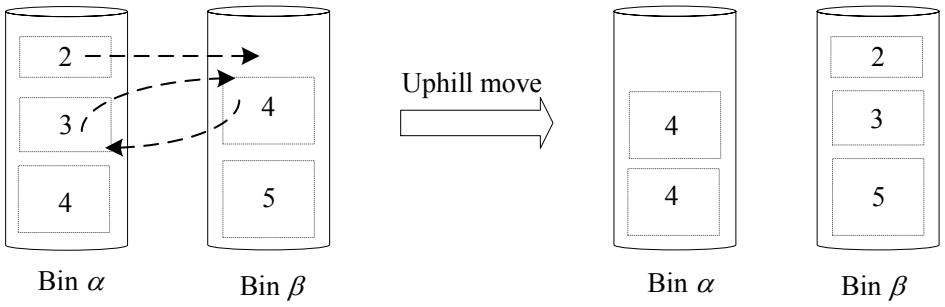


Figure 2.7 An example of Swap (1,2).

2.3.2.4 Swap (1,2)

We swap item i from bin α with items j and k from bin β. The change in the objective function value that results from swapping item i with size $t_{\alpha i}$ from bin α with item j with size $t_{\beta j}$ and item k with size $t_{\beta k}$ from bin β is given by

$$\Delta f = (l_\alpha - t_{\alpha i} + t_{\beta j} + t_{\beta k})^2 + (l_\beta - t_{\beta j} - t_{\beta k} + t_{\alpha i})^2 - l_\alpha^2 - l_\beta^2 \ . \qquad (2.4)$$

2.3.2.5 Swap(2,2)

We swap item i and item j from bin α with item k and item l from bin β. The change in the objective function value that results from swapping item i with size $t_{\alpha i}$ and item j with size $t_{\alpha j}$ from bin α with item k with size $t_{\beta k}$ and item l with size $t_{\beta l}$ from bin β is given by

$$\Delta f = (l_\alpha - t_{\alpha i} - t_{\alpha j} + t_{\beta k} + t_{\beta l})^2 + (l_\beta - t_{\beta k} - t_{\beta l} + t_{\alpha i} + t_{\alpha j})^2 - l_\alpha^2 - l_\beta^2 \ . \qquad (2.5)$$

It may not be possible to replicate all Swap(2,2) uphill moves by two successive uphill moves using Swap(1,1). We demonstrate this in Figure 2.8, where a Swap(2,2) downhill move can only be obtained by one downhill swap move, followed by one uphill swap move.

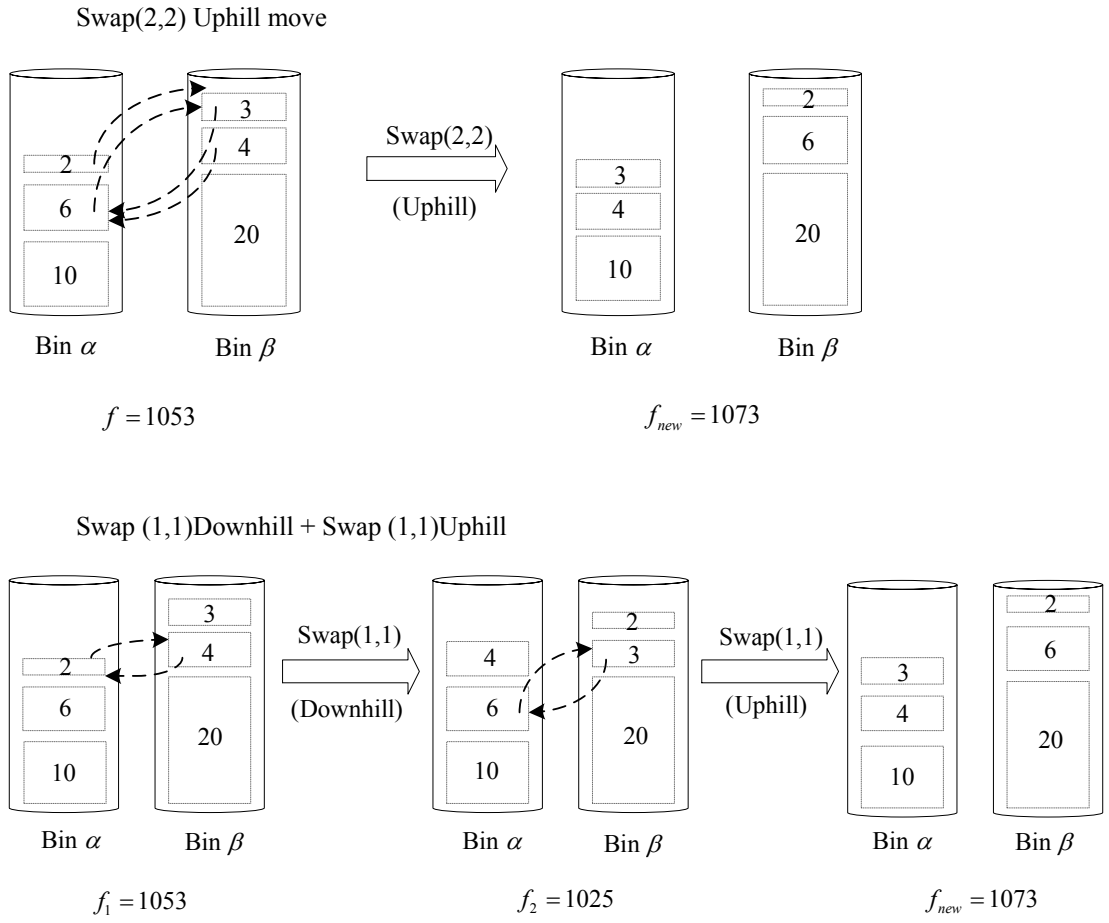


Figure 2.8 An example of a Swap(2,2) uphill move. This is equivalent to a Swap(1,1) downhill followed by a Swap(1,1) uphill.

2.3.2.6 Computational Complexity

The complexity of our algorithm depends on the type of swaps, the number of items *n* and the average number of items $\overline{p}$ in each bin. For example, the complexity of an algorithm using Swap(1,2) is $O(n^2\overline{p})$. For some large practical problem instances, the average number of items in each bin may be much smaller than *n.* Note also that, for an instance with an average of three items, a Swap(2,3) would not be computationally efficient compared to Swap(1,0).

2.3.3 The Weight Annealing Concept for the One-Dimensional Bin Packing Problem

During local search, the implementation of the four basic swaps often leads to a local optimum quickly. The key feature of the weight annealing process is the distortion of item sizes to allow for both downhill moves and uphill moves, in order to escape from poor local solutions.

2.3.3.1 Weight Assignments

The changes in the apparent sizes of the items are achieved by assigning different weights to different bins and their items according how well the bins are packed at every iteration of the algorithm. We assign to each bin i a weight w_i that is associated with all items in this bin as follows

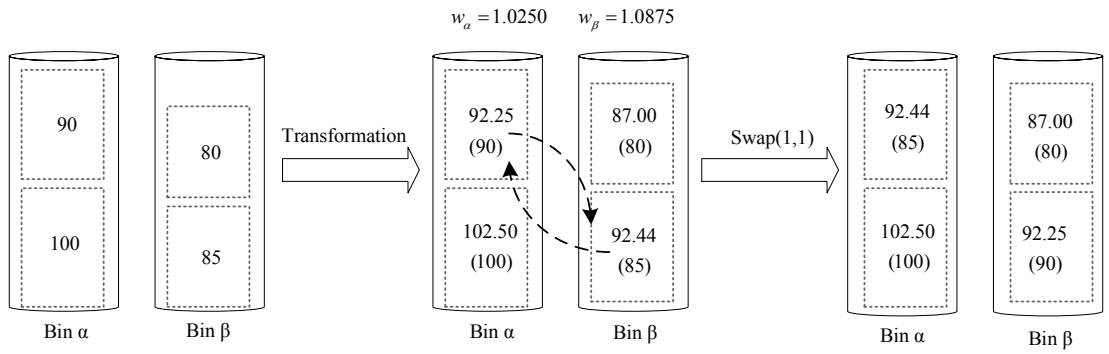
$$w_i = 1 + Kr_i \tag{2.6}$$

where r_i is the residual capacity of bin i, $r_i = \left(\frac{C - l_i}{C} \right)$, and K is a constant.

Thus, every bin and all of its items are assigned the same weight according to equation (2.6). The scaling parameter K controls the amount of size distortion for each item. The size distortion for an item will be linearly proportional to the residual capacity of its bin. At a local maximum, the not-so-well packed bins will have large residual bin capacities. Without downhill moves, there is little chance of escaping from a poor local optimum.

To enable downhill moves, the weighting function increases the sizes of the items in the poorly packed bins. Since the objective is to fill up the maximum number of completely filled bins, the size transformation increases the chance of a swap between one of the enlarged items in this bin and a smaller item from another bin. Thus, we have

$$C = 200, \quad K = +0.5, \qquad w_i = 1 + 0.5\left(\frac{200 - l_i}{200}\right)$$



Transformed Space: $f' = (102.50 + 92.25)^2 + (87.00 + 92.44)^2$ $f'_{new} = (102.50 + 92.44)^2 + (87.00 + 92.25)^2$
(Uphill move) $= 70126.3$ $= 70132.2$

Original Space: $f = (100 + 90)^2 + (85 + 80)^2$ $f_{new} = (100 + 85)^2 + (90 + 80)^2$
(Downhill move) $= 63325$ $= 63125$

Figure 2.9 An example of an uphill move in the transformed space associated with a downhill move in the original space in 1BP.

an uphill move in the transformed space, which may be a downhill move in the original space. In Figure 2.9, we illustrate this case. A swap is allowed as long as it is feasible in the original space.

2.3.3.2 Algorithm Implementation

The implementation of the various swap schemes in the transformed space can be done easily. For Swap(1,0), we evaluate only the change in objective function value Δf in the transformed space as a result of moving item i with size $t_{\alpha i}$ from bin α to bin β, as follows

$$\Delta f = (l_\alpha * w_\alpha - t_{\alpha i} * w_\alpha)^2 + (l_\beta * w_\beta + t_{\alpha i} * w_\alpha)^2 \quad - (l_\alpha * w_\alpha)^2 - (l_\beta * w_\beta)^2 \ . \qquad (2.7)$$

For Swap(1,1), we evaluate the change of objective function value Δf as a result of swapping item i of size $t_{\alpha i}$ from bin α with item j of size $t_{\beta j}$ from bin β, as follows

$$\Delta f = (l_\alpha * w_\alpha - t_{\alpha i} * w_\alpha + t_{\beta j} * w_\beta)^2 + (l_\beta * w_\beta - t_{\beta j} * w_\beta + t_{\alpha i} * w_\alpha)^2 - (l_\alpha * w_\alpha)^2 - (l_\beta * w_\beta)^2. \quad (2.8)$$

For Swap(1,2), the change in objective function value Δf that results from swapping item i with size $t_{\alpha i}$ in bin α with two items, item j of size $t_{\beta j}$ and item k of size $t_{\beta k}$ in bin β, is

$$\Delta f = (l_\alpha * w_\alpha - t_{\alpha i} * w_\alpha + t_{\beta j} * w_\beta + t_{\beta k} * w_\beta)^2 + (l_\beta * w_\beta - t_{\beta j} * w_\beta - t_{\beta k} * w_\beta + t_{\alpha i} * w_\alpha)^2 - (l_\alpha * w_\alpha)^2 - (l_\beta * w_\beta)^2. \quad (2.9)$$

Finally, for Swap(2,2), the change in objective function value Δf that results from swapping two items, item i of size $t_{\alpha i}$ and item j of size $t_{\alpha j}$ in bin α with two items, item k of size $t_{\beta k}$ and item l of size $t_{\beta l}$ in bin β, is

$$\Delta f = (l_\alpha * w_\alpha - t_{\alpha i} * w_\alpha - t_{\alpha j} * w_\alpha + t_{\beta k} * w_\beta + t_{\beta l} * w_\beta)^2 + (l_\beta * w_\beta - t_{\beta k} * w_\beta - t_{\beta l} * w_\beta + t_{\alpha i} * w_\alpha + t_{\alpha j} * w_\alpha)^2 - (l_\alpha * w_\alpha)^2 - (l_\beta * w_\beta)^2. \quad (2.10)$$

2.3.3.3 Annealing Process

For the annealing process, we have a controlling parameter T that governs the amount by which each single weight can be varied. Thus, the weight of bin i and its items at one particular iteration is a function of the residual capacity r_i and the temperature T at that iteration. We denote the weight by w_i^T. T controls the annealing process in the following way.

- At the start of the process, a high temperature allows significant changes to weights to allow for more downhill moves.
- As the temperature cools, the amount of each item distortion is reduced.

- If T reaches zero, the algorithm solves the problem in the original space with all weights being equal to 1.

2.3.4 The Weight Annealing Algorithm for the One-Dimensional Bin Packing Problem (WA1BP)

In Table 2.1, we show our weight annealing algorithm for the one-dimensional bin packing problem. The list of item sizes and the lower bound (*LB*) of the objective function value are inputs. The *LB* in our algorithm is the minimal bin number, $LB = \left\lceil 1/C \sum_{i=1}^{n} t_i \right\rceil$, where t_i is the size of item *i*. An initial solution is constructed using the FFD algorithm. The items are sorted in non-increasing order according to their sizes. At each step, we insert the item at the top of the current item list into the bin with the smallest index, provided this bin has enough residual capacity. If no such bin exists, we open a new bin to accommodate the current item. We continue inserting items until the item list is empty. Given the initial solution, we compute the bin load l_i and the residual capacity r_i for each bin *i*.

To improve the current solution, we carry out swapping operations with weight annealing. To begin, we set $T = 1$ and $K = 0.05$. We start the first iteration of *i* loop by computing the weight for bin *i* according to $w_i^T = (1 + Kr_i)^T$. The swapping procedure compares the items in the first bin with the items in the second bins, and so on. We repeat the swapping process for all pairs of bins. For a current pair bins denoted by (α,β), we carry out the swapping of items in the following order.

- Swap(1,0). We evaluate if the first item (item *i*) in bin α can be inserted into bin β without violating the capacity constraint of bin β in the original space, i.e.,

whether bin β has the sufficient original residual capacity to accommodate item i in its original size. If such a move is feasible, we next evaluate the change in objective function value Δf of the move in the transformed space. If $\Delta f \geq 0$, we move item i from bin α to bin β. With respect to the data structure, we use a single linked list to dynamically update the item list for each bin. After this move, we determine if bin α is empty. If bin α is empty and the total number of utilized bins reaches *LB*, we have obtained the optimal solution and can exit the *i loop* and output the results. If bin α is partially filled or we have not yet found the optimal solution, we exit the Swap(1,0) step and proceed to the Swap(1,1) step, after updating the residual capacities and linked lists for bin α and bin β. If the move of the first item is infeasible or $\Delta f < 0$, we consider the second item in bin α and so on, until we find a feasible move with $\Delta f \geq 0$ or have searched all items in bin α.

- Swap(1,1). We examine if the first item (item i) in bin α can be swapped with the first item (item j) in bin β in the original space without violating the capacity constraints of both bins. If such a swap is feasible and $\Delta f \geq 0$, we move item i from bin α to bin β and item j from bin β to bin α and update the linked lists and residual capacities. We then proceed to Swap(1,2) after making the improving move, and exit the i loop if the optimal solution has been found. If the swap is infeasible or $\Delta f < 0$, we consider swapping the first item from bin α with the second item from bin β, and so on, until we find a feasible move or have exhausted the search for all pairs (i, j) of items.
- Swap(1,2). We swap item i in bin α with a pair of items (j,k) in bin β, if the swap is feasible in the original space and $\Delta f \geq 0$ in the transformed space. For all tuples

(i,j,k), we examine every item in bin α and all possible pairs of items in bin β . We exit this subroutine as soon as we have found a feasible move and exit the i loop if an optimal solution has been found.

- Swap(2,2). In every iteration of this subroutine, we consider a pair of items (i,j) in bin α and another pair of items (k,l) in bin β. We evaluate all tuples (i,j,k,l) and exit the subroutine when we have found a feasible move with $\Delta f \geq 0$. We exit the i loop if an optimal solution has been found.

We now reach the end of the first temperature step and start the second iteration of i loop. The temperature T is now reduced by a factor *Tred*, i.e. $T = T \times Tred$. We set *Tred* = 0.95 for all runs. At the beginning of the second temperature step and all subsequent steps, we carrying re-weighting by computing a new set of weights w_i based on the current bin residual capacities and the current temperature T. We then carry out the four swap schemes consecutively for all pairs of bins. We repeat the iterations for 50 temperature steps (*nloop* = 50). Since T is always set arbitrarily at 1 at the beginning of the algorithm, we may remove T as a parameter by setting $w_i = (1 + Kr_i)^{(Tred)^i}$. We retain the parameter T for clarity of presentation.

The algorithm exits when the number of bins has reached the lower bound or has completed the annealing process. The outputs include distribution of items, residual capacities of the bins, and computation times.

Step 0. Initialization
Parameters are K , *nloop, T, Tred*
Set $K = 0.05$, $nloop = 50$, $T = 1$, $Tred = 0.95$
Inputs are number of items, item sizes, bin capacity, and lower bound (*LB)*

Step 1. Construct initial solution using first-fit decreasing procedure
Sort the items in non-increasing order according to item sizes
Do while (item list is not empty){
Place the first item in the list into the lowest numbered feasible bin
Open a new bin if the item cannot fit into any existing bin
Remove the item from the item list
}
Compute residual capacity r_i

Step 2. Improve current solution
For i =1 to *nloop*
Compute weights: $w_i^T = (1\text{-}K\, r_i)^T$
Do for all pairs of bins{
Perform Swap (1,0)
Evaluate feasibility and Δf
if $\Delta f \geq 0$
Move the item
Exit Swap(1,0) and,
Exit i loop if *LB* is reached

Perform Swap (1,1)
Evaluate feasibility and Δf
if $\Delta f \geq 0$
Swap the items
Exit Swap(1,1) and,
Exit i loop if *LB* is reached

Perform Swap (1,2)
Evaluate feasibility and Δf
if $\Delta f \geq 0$
Swap the items
Exit Swap(1,2) and,
Exit i loop if *LB* is reached

Perform Swap (2,2)
Evaluate feasibility and Δf
if $\Delta f \geq 0$
Swap the items
Exit Swap(2,2) and,
Exit i loop if *LB* is reached
}
$T := T$ x T_{red}
End of i loop
Step 3.
Outputs are the number of bins used, the final distribution of items, and r_i

Table 2.1 Weight annealing algorithm for the one-dimensional bin packing problem.

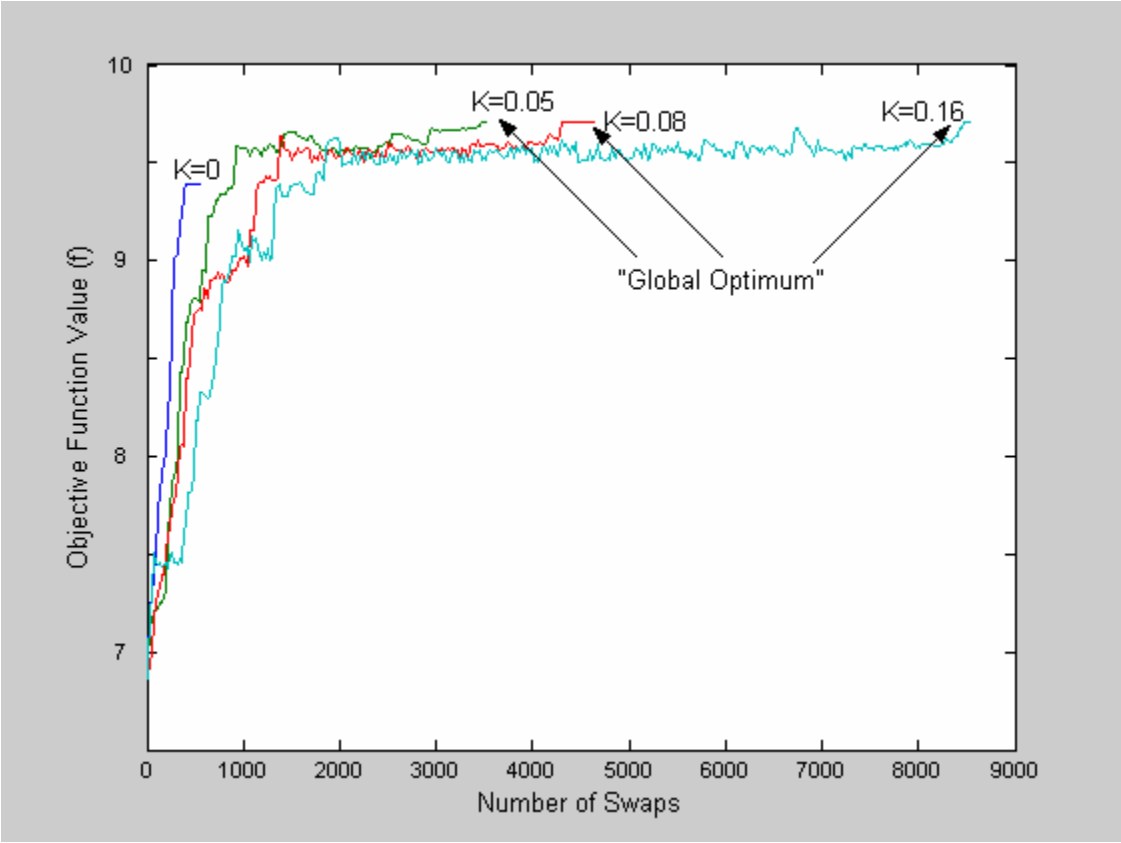


Figure 2.10 Behavior of WA1BP with K = 0.05, 0.08, and 0.16.

2.3.5 Sensitivity Analysis

The choice of a value for K, or how much each item should be distorted given a residual bin capacity, is an important decision in our algorithm. The sensitivity analysis on K is carried out with one of the benchmark problem, HARD2 from Scholl, Klein, and Jürgens (1997). As shown in Figure 2.10, without any size distortion, i.e., with K set to zero, the algorithm stops prematurely at a local optimum (no swaps are possible as the algorithm is trapped at a local maximum). On the other hand, by increasing the size of the distortion, the rate of convergence slows down. When K is 0.16, it takes about 8,500 swaps to converge. Increasing item sizes only by a small amount (K = 0.05) causes a fast convergence rate (the number of swaps to optimum is about 3000). For WA1BP, we set K equal to 0.05.

2.4 The Weight Annealing Concept for the Dual Bin Packing Problem

2.4.1 Objective Function for Solving the One-Dimensional Dual Bin Packing Problem

We consider an alternative formulation of the one-dimensional bin packing problem that partitions the set of items into a minimum number of subsets such that each subset does not violate the capacity constraint. This is a min-max problem that can be solved by starting with an infeasible solution with the known minimum number of bins, each with its size enlarged by a small fraction. The number of bins is now fixed. At every iteration, the objective function (2.9) minimizes the sum of squares of bin loads, where bin load l_i is defined as the sum of sizes of items in bin *i,* or $l_i = \sum_{j=1}^{q_i} t_{ij}$, where t_{ij} is the size of item j in bin i and q_i is the number of items in bin i. The objective function is given by

$$\text{Minimize} \quad f = \sum_{i=1}^{p} (l_i)^2 \tag{2.11}$$

where p is the number of bins in the current solution.

We seek to distribute and pack all items into the identical bins using minimum space. A starting infeasible solution may have large differences in bin loads. We want to obtain a final feasible configuration of bin loads that has uniform loading, after satisfying all capacity constraints. In Figure 2.11, we show the local search procedure using (2.11) in which an item of size 1 is moved from bin α to bin β to obtain an equal bin load and maximum residual bin capacity for both bins. The one-dimensional dual bin packing

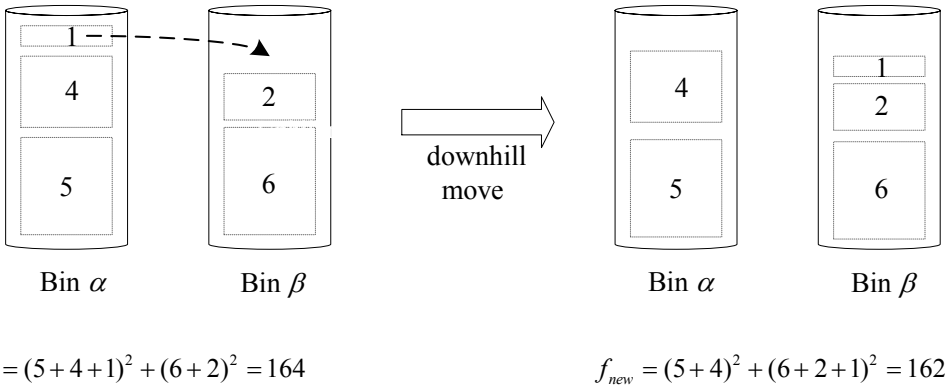


Figure 2.11 Minimizing the sum of squares of bin loads in the dual bin packing problem.

problem (1DBP) is also known in the literature as the multiprocessor scheduling problem, where the objective is to minimize the maximum time to complete a batch of jobs (Alvin et al. 2004).

2.4.2 Outline of the Algorithm to Solve the 1DPB

The structure of the algorithm that solves the 1DPB (denoted by WA1DPB) is very much similar to the structure of WA1BP. In WA1DPB, we enlarge the bin size to allow infeasible solutions initially. We start with an initial solution that has similar proportions of large and small items across the bins. The outline of WA1DPB is as follows.

- Fix the number of bins to a known lower bound (*LB)*.
- Fix the bin size at a value above the original bin size to allow infeasible solutions.
- Construct an initial solution with a modified first-fit decreasing heuristic. If there are no capacity violations in the solution, then we are done.
- Use the objective function that minimizes the sum of squares of bin loads.

- Distort the sizes of items with respect to the residual capacities of their respective bins.
- Use the four basic swap schemes to exchange items between all pairs of bins during the local search to eliminate capacity violations.
- Stop the algorithm when all capacity violations have been removed. If the current solution is still infeasible after a fixed number of iterations, increase the lower bound by one and construct a new initial solution.

2.4.3 Modified First-Fit Decreasing

As noted earlier, an approximation algorithm like FFD produces an initial solution that can have an unbalanced distribution of bin loads. This can be a poor deterministic starting solution for the dual one-dimensional bin packing problem which aims to achieve uniform loading of items across bins. Our construction heuristic is a modification to the FFD procedure, and tries to accomplish the following.

- Quickly produce a high-quality solution that may violate the capacity constraints.
- Achieve good load balancing to improve bin usability. A good load balancing is usually not achievable by an approximation algorithm like FFD.
- Introduce randomness to allow optimization runs based on different starting solutions, as opposed to a traditional approximation algorithm that produces only one deterministic starting solution.

In Table 2.2, we show the outline of the modified first-fit decreasing algorithm (denoted by MFFD).

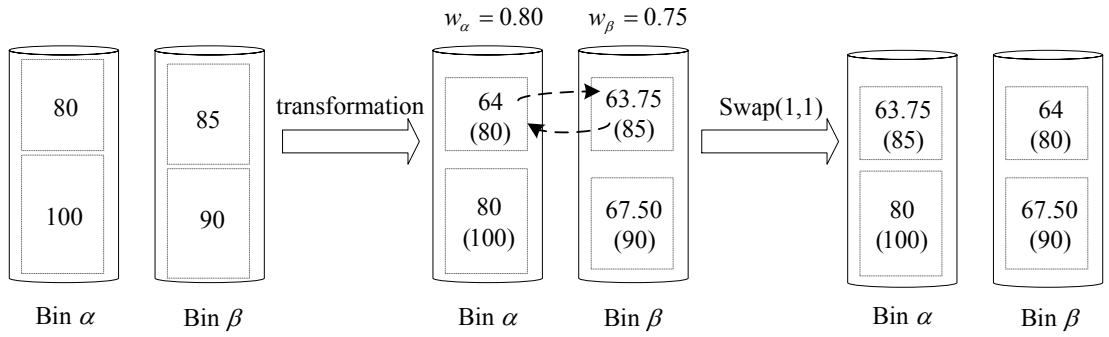
```
begin
        Set the number of bin at a fixed quantity (a known lower bound LB)
        Sort the items according to the item sizes in a non-increasing order
        Increase the size of each bin to allow infeasibility

        for i = 1: nloop do
                Select an item for insertion
                        Generate a random number (0 or 1) to pick one of the first two items
                        from the item list.
                        If the first item is not selected, re-insert it back into the item list
                        Generate another random number to decide if the second item is selected
                        Repeat, until an item is selected or only one item remains
                Insert the item into the bin with largest residual capacity
                To break a tie, pick the lower numbered bin
                Remove the inserted item from the current item list
        end
end
```

Table 2.2 Modified first-fit decreasing algorithm.

2.4.4 Weight Assignments

The changes in the apparent sizes of the items are achieved by assigning different weights to different bins at every iteration. In WA1DPB, we propose a negative value of K to reduce the apparent sizes to be linearly proportional to the residual capacity of its bin. A poor local solution may have bins that have smaller bin loads compared to the rest of the bins. Without uphill moves, there is little chance of escaping from a poor local solution. To enable uphill moves, the weighting function will have the sizes of the items in the poorly packed bins reduced by a relatively greater amount. After the size distortion, downhill moves in the transformed space may be uphill moves in the original space. In Figure 2.12, we show a case where K is set equal to -2. For WA1DPB, we typically fix K to be a small number, say -0.05, for all test instances, thus allowing only small downhill moves.

$$C = 200, \quad K = -2,$$
$$w_i = 1 - \left(\frac{200 - l_i}{200} \right) \times 2$$



Transformed Space: $f' = (64+80)^2 + (63.75+67.50)^2$ $f'_{new} = (63.75+80)^2 + (64+67.50)^2$
(Downhill move) $= 37962.6$ $= 37956.3$

Original Space: $f = (80+100)^2 + (85+90)^2$ $f_{new} = (85+100)^2 + (80+90)^2$
(Uphill move) $= 63025$ $= 63125$

Figure 2.12 An example of uphill climb in the original space associated with a downhill climb in the transformed space in 1DPB.

2.4.5 Sensitivity Analysis

The choice of a value for *K*, or how much each item should be distorted given a residual bin capacity, is an important decision in WA1DBP. In Figure 2.13, we show a sensitivity analysis on the behavior of the algorithm with different values of *K* on the HARD2 instance from Scholl, Klein, and Jürgens (1997). We see that a large size distortion, either a big size increase with a large positive *K* value (*K* = 5.0) or a large size reduction with a small negative *K* value (*K* = -1.0) causes the algorithm to diverge. For WA1DBP, we set *K* equal to -0.05 and this choice results in a fast convergence after 3000 swaps or so.

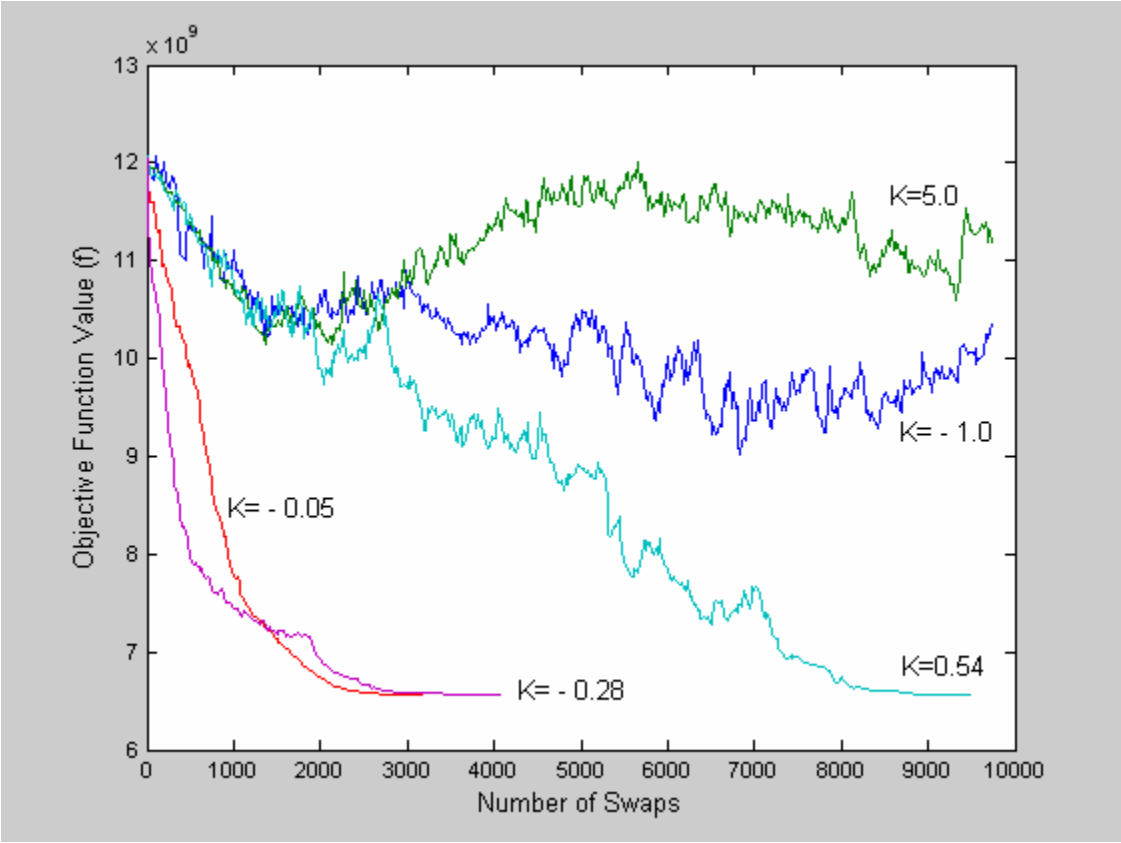


Figure 2.13 Behavior of WA1DBP with K = -1.0, -0.28, -0.05, 0.54, and 5.0.

2.4.6 The Weight Annealing Algorithm for the Dual Bin Packing Problem

In Table 2.3, we give the steps of WA1DPB. The algorithm generates an initial solution that is likely to have capacity violations with respect to the original bin size. We set the enlarged bin size to be 1.1 times the original size (the size of the multiplier is arbitrary and has no implications on the performance of the algorithm as long as it is large enough to allow for swaps to take place during the local search). As the algorithm proceeds, the solution improves with a decreasing number of capacity violations. If the current solution is still not feasible after a fixed number of iterations, we increase the lower bound by one and construct a new initial solution. The program exits when there are no capacity violations.

Step 0. Initialization
Parameters are *K, nloop1, nloop2, T,* and *Tred*
Set $K = -0.05$, $nloop1 = 20$, $nloop2 = 50$, $T = 1$, and $Tred = 0.95$
Inputs are number of items, item sizes, bin capacity and number of bins

Step 1. Optimization runs
For j=1 to *nloop1*
Construct initial solution using modified first-fit decreasing procedure
Compute residual capacity r_i
LB = number of bins
bin capacity = 1.1 × bin capacity
T=1
For k =1 to *nloop2*
Compute weights: $w_i^T = (1-K\, r_i)^T$
Do for all pairs of bins{
Perform Swap (1,0)
Evaluate feasibility and Δf
If $\Delta f \le 0$
Move the item
Exit Swap(1,0) and,
Exit j loop and k loop if LB is reached

Perform Swap (1,1)
Evaluate feasibility and Δf
if $\Delta f \le 0$
Swap the items
Exit Swap(1,1) and,
Exit j loop and k loop if LB is reached

Perform Swap (1,2)
Evaluate feasibility and Δf
if $\Delta f \le 0$
Swap the items
Exit Swap(1,2) and,
Exit j loop and k loop if LB is reached

Perform Swap (2,2)
Evaluate feasibility and Δf
if $\Delta f \le 0$
Swap the items
Exit Swap(2,2) and,
Exit j loop and k loop if LB is reached
}
$T := T \times T_{red}$
End of k loop
LB=LB+1
End of j loop
Step 3. Outputs are the final distribution of items and r_i

Table 2.3 Weight annealing algorithm for the dual bin packing problem.

Name	Notation	Comments
Uniform	U120, U250, U500, U1000	The bin capacity is $C = 150$. Items with integer sizes are drawn from a uniform distribution between 20 and 150. U120 denotes $n = 120$ items. For each value of $n = 120$, 250, 500, and 1000, there are 20 instances. These problems were developed by Falkenauer (1996) and have been solved optimally by Carvalho (1999). They are available on-line at http://people.brunel.ac.uk/~mastjjb/jeb/orlib/binpackinfo.html.
Triplet	T60, T120, T249, T501	The bin capacity is $C = 1000$. Items with integer sizes are drawn from a uniform distribution between 250 and 500. T60 denotes $n = 60$ items. For each value of $n = 60$, 120, 249, and 501, there are 20 instances. The optimal solutions are known and have exactly three items per bin (hence the name triplets). These problems were developed by Falkenauer (1996). They are available on-line at http://people.brunel.ac.uk/~mastjjb/jeb/orlib/binpackinfo.html.
Set	Set1, Set 2, Set3	Set1 has 720 instances with items drawn from a uniform distribution on three intervals [1, 100], [20, 100], and [30, 100]. The bin capacity is $C = 100$, 120, and 150 and $n = 50$, 100, 200, and 500. Set 2 has 480 instances with $C = 1000$ and $n = 50$, 100, 200, and 500. Each bin has an average of 3 to 9 items. Set 3 has 10 instances with $C = 100000$, $n = 200$, and items are drawn from a uniform distribution on [20000, 35000]. Set3 is considered the most difficult of the three sets. These problems were developed by Scholl, Klein, and Jürgens (1997) and they reported that 1184 of the problems have been solved to optimality. Alvim, Ribeiro, F. Glover, and D. Aloise (2004) reported the optimal solutions for the remaining 26 problems. The three problem sets are available on-line at http://www.wiwi.uni-jena.de/Entscheidung/binpp/index.htm.

Table 2.4 Descriptions of benchmark test problems.

Name	Notation	Comments
Was	Was1, Was2	Was1 has 100 instances with $C = 1000$ and $n = 100$. The minimum size of an item is 150 and the maximum size is 200.
		Was 2 has 100 instances with $C = 1000$ and $n = 120$. The minimum size of an item is 150 and the maximum size is 200.
		These problems were developed by Schwerin and Wäscher (1997, 1999). All problems have been solved to optimality. Both problem sets are available on-line at http://www.apdio.pt/sicup/Sicuphomepage/research.htm.
Gau	Gau1	These 17 problems are taken from Wäscher and Gau (1996). They are reported as difficult problems by the authors. Some of these problems have been solved to optimality. This problem set is available on-line at http://www.apdio.pt/sicup/Sicuphomepage/research.htm.

Table 2.4 (continued)

2.5 Computational Experiments of 1DBP

We coded WA1BP in C and C++ and used a 3 GHz Pentium 4 computer with 256 MB of RAM.

2.5.1 Benchmark Problems

In Table 2.4, we describe six well-known sets of benchmark test problems that contain 1,587 problems.

2.5.2 Results for WA1BP

In Table 2.5, we show the results for HI_BP, PMBS′ + VNS, and WA1BP to 1,370 instances from the Uniform, Triplet, and Set benchmark problem sets. The results for HI_BP and PMBS′ + VNS are taken from the literature. The optimal solution is

known for each instance, so we tabulate the number of times each procedure obtains the optimal solution. For example, there are 20 instances in problem set U120 and all three procedures find the optimal solution to each instance (there is a value of 20 in the Number Optimal column of each procedure). We see that both HI_BP and WA1BP found optimal solutions to all 1,370 instances. PMBS′ + VNS found optimal solutions to 1,329 instances. Even though the computing platforms are different, each procedure is very fast with an average solution time that is less than a quarter of a second.

In Table 2.6, we show the results for BISON, HI_BP, PMBS′ + VNS, MTPCS, and WA1BP to 1,210 instances from the Set benchmark problem set. The results for HI_BP, BISON, PMBS′ + VNS, and MTPCS are taken from the literature. The papers on BISON and MTPCS report results for the instances in Set and do not report results for instances in Uniform and Triplet. On Set1 and Set2 with MTPCS, Schwerin and Wäscher (1999) set a time limit of 1,000 seconds. This limit was insufficient for obtaining a lower bound for each instance in Set3. The authors ran their lower bounding procedure for 1,500 seconds to 3,000 seconds depending on the specific instance, started MTPCS with the lower bound, and set a time limit of 1,000 seconds. We see that both HI_BP and WA1BP found optimal solutions to all 1,210 instances. BISON, PMBS′ + VNS, and MTPCS fell short and did not find the optimal solutions to 37, 40, and 94 instances, respectively. We note that BISON and MTPCS were run on very slow machines so that their computation times are very long when compared to HI_BP, PMBS′ + VNS, and WA1BP.

		HI_BP		PMBS′ + VNS		WA1BP	
Name	Instances	Number Optimal	Average Time (s)	Number Optimal	Average Time (s)	Number Optimal	Average Time (s)
U120	20	20	0.00	20	0.02	20	0.00
U250	20	20	0.12	19	0.03	20	0.03
U500	20	20	0.00	20	0.04	20	0.18
U1000	20	20	0.01	20	0.07	20	1.24
T60	20	20	0.37	20	0.01	20	0.00
T120	20	20	0.85	20	0.02	20	0.00
T249	20	20	0.22	20	0.02	20	0.01
T501	20	20	2.49	20	0.06	20	0.03
Set1	720	720	0.19	694	0.15	720	0.17
Set2	480	480	0.01	474	0.10	480	0.19
Set3	10	10	4.60	2	3.74	10	0.13
Total	1370	1370		1329		1370	
Average			0.20		0.15		0.18

HI_BP 1.7 GHz Pentium 4
MBS′ + VNS 400 MHz Pentium
WA1BP 3 GHz Pentium 4

Table 2.5 Results for HI_BP, PMBS′ + VNS, and WA1BP to 1,370 instances from the Uniform, Triplet, and Set benchmark problem sets.

		BISON		HI_BP		PMBS′ + VNS		MTPCS		WA1BP	
Name	Instances	Number Optimal	Average Time (s)	Number Optimal	Average Time (s)	Number Optimal	Average Time (s)	Number Optimal	Average Time (s)	Number Optimal	Average Time (s)
Set1	720	697	32.4	720	0.19	694	0.15	717	7.3	720	0.17
Set2	480	473	16.3	480	0.01	474	0.10	394	221	480	0.19
Set3	10	3	700.2	10	4.60	2	3.74	5	3164.8	10	0.13
Total	1210	1173		1210		1170		1116		1210	
Average			31.53		0.16		0.15		118.2		0.18

BISON 66 MHz 80486 DX2

MTPCS 90 MHz Pentium. Set1 and Set 2 have a time limit of 1000s. In Set3, the lower bounding procedure is run anywhere from 1500s to 3000s and MTPCS is then started with the lower bound and given a time limit of 1000s.

Table 2.6 Results for BISON, HI_BP, PMBS′ + VNS, MTPCS, and WA1BP to 1,210 instances from the Set benchmark problem set.

		HI_BP		WA1BP	
Name	Instances	Number Optimal	Average Time (s)	Number Optimal	Average Time (s)
Was1	100	100	NA	100	0.01
Was2	100	100	NA	100	0.01
Tota1		200		200	
NA	Not Available				

Table 2.7 Results for HI_BP and WA1BP to 200 instances from the Was benchmark problem set.

In Table 2.7, we show the results for HI_BP and WA1BP to 200 instances from the Was benchmark problem set. The results for HI_BP are taken from the literature. The computation times for HI_BP were not reported. We note that Alvim, Ribeiro, Glover, and Aloise (2004) reported that HI_BP improved the solutions to three instances (BPP56, BPP71, and BPP 81) in Was1 and Was2. We see that HI_BP and WA1BP found optimal solutions to all 200 instances. WA1BP took 0.01 seconds on average to solve an instance.

When Alvim, Ribeiro, Glover, and Aloise (2004) applied HI_BP to the 17 instances from the Gau benchmark set of problems, they reported that HI_BP found eight new, improved solutions and missed the optimal solutions to five instances. HI_BP was very fast with computation times of 0.01 seconds or less reported for the eight new solutions (no other times were given).

When MTPCS was applied to the 17 instances from the Gau set, it generated the same best-known solutions to eight problems, found the optimal solution to one problem, and failed to find the optimal solutions to eight problems. The results for MTPCS were given at http://www.apdio.pt/sicup/Sicuphomepage/research.htm. The computation times for MTPCS were not reported on the web page.

	Best	WA1BP				WA1BP	
Instance	Known	Solution	Time (s)	Instance	Optimal	Solution	Time (s)
TEST0005	29	29	0.03	TEST0044	14	14	0.08
TEST0014	24	24	0.00	TEST0049	11	11	0.03
TEST0022	15	15	0.00	TEST0054	14	14	0.03
TEST0030	28	28	0.05	TEST0055	15	15	0.03
TEST0058	21	**20**	0.20	TEST0055	20	20	0.19
TEST0065	16	16	0.00	TEST0075	13	13	0.33
TEST0068	13	**12**	0.54	TEST0084	16	16	0.00
TEST0082	25	**24**	0.00	TEST0095	16	16	0.03
Bold	New optimal solution			TEST0097	12	12	0.03

Table 2.8 Results for WA1BP to 17 instances from the Gau benchmark problem set.

We applied WA1BP to the 17 instances from the Gau set and present our results in Table 2.8. There are eight instances that have best-known solutions (listed on the left side of Table 2.8) and WA1BP found new, optimal solutions to three of these instances (TEST0058, TEST0068, and TEST0082). WA1BP generated the same best-known solutions to the remaining five instances. There are nine instances that have known optimal solutions (listed on the right side of Table 2.8) and WA1BP found the same optimal solutions to all nine instances. WA1BP took 0.09 second on average to solve an instance.

2.5.3 Experiment on the Re-Weighting Process

In WA1BP, it is important that we carry out re-weighting at the beginning of every loop according to $w_i^T = (1 + Kr_i)^T$. In Table 2.6, we see that WA1BP is able to find the optimal solutions to all 10 instances in Set3. Our experiment has shown that when we did not carry out re-weighting at all, and used the initial weights assigned the items (set T

= 1) for all iterations, the algorithm could find only three optimal solutions to the 10 instances in Set3; it could not find the optimal solutions to HARD2, HARD3, HARD4, HARD5, HARD 6, HARD7 and HARD9. If we carried out re-weighting at every loop, but did not reduce the amount of re-weighting (set $T = 1$) for all iterations, the algorithm could find eight optimal solutions to the 10 instances in Set3; it could not find the optimal solutions to HARD2 and HARD3.

2.5.4 WA1BP versus WA1DBP

WA1BP seeks to completely fill up as many bins as possible. WA1DBP attempts to obtain a uniform loading amongst the bins. In Appendix A, we show the results of WA1BP and WA1DBP in solving TEST0044 of the Gau set. For WA1BP, the optimum solution has 13 out of 14 bins completely packed to the capacity of 10000 and the remaining bin with a residual capacity of 11. We see that this packing solution is a global optimal solution obtained by WA1BP. By contrast, the solution obtained by WA1DBP has a more uniform loading, with eight out of 14 bins completely packed, and the maximum residual capacities of the remaining six bins not exceeding three.

In Table 2.9, we show that the results of WA1DBP in solving the 10 hard benchmark problem from Set3. We see that the results produced by WA1BP and WA1DBP are comparable. In Table 2.10, we show that both WA1BP and WA1DPB outperform BISON, PMBS′ + VNS , and HI_BP in solving Set3. In Table 2.11, we show the two new optimal solutions found by WA1DBP for the Gau set.

		WA1BP		WA1DBP	
Instance	Optimal	Solution	Time(s)	Solution	Time(s)
HARD0	56	56	0.02	56	0.03
HARD1	57	57	0.16	57	0.00
HARD2	56	56	0.52	56	0.05
HARD3	55	55	0.39	55	0.03
HARD4	57	57	0.08	57	0.03
HARD5	56	56	0.08	56	0.03
HARD6	57	57	0.05	57	0.02
HARD7	55	55	0.06	55	0.02
HARD8	57	57	0.05	57	0.02
HARD9	56	56	0.06	56	0.02

Table 2.9 Results for WA1BP and WA1DBP to the 10 hard instances from Set3 of Scholl, Klein, and Jürgens (1997).

	Instances	BISON	PMBS′+ VNS	HI_BP	WA1BP	WA1DBP
Optimum	10	3	2	10	10	10
Average Time(s)		1.8	-	4.60	0.15	0.03
Maximum Time(s)		700.2	-	44.76	0.52	0.05

Table 2.10 Number of instances solved to optimality on the 10 hard instances from Set3 of Scholl, Klein, and Jürgens (1997).

		WA1DBP	
Instance	Optimal	Solution	Time (s)
TEST0005	28	28	0.00
TEST0030	27	27	28.57

Table 2.11 New optimal values obtained by WA1DBP for the Gau benchmark problem set.

2.6 Conclusions

We developed a powerful new procedure (WA1BP) that implements the concept of weight annealing to solve the one-dimensional bin packing problem. WA1BP is easy to understand and easy to follow, and it generated very high-quality solutions very quickly. We see that weight annealing is able to focus computational efforts on poorly packed bins by creating distortions to the items in these bins, and allowing higher frequencies of item exchanges, compared to items in fully packed bins, which will have no distortions applied to them. The search for the global optimum is accelerated as a consequence.

When applied to 1,587 benchmark instances, WA1BP found the best-known or optimal solutions to 1,584 instances and generated new optimal solutions to the remaining three instances. Over all benchmark instances, it averaged 0.16 seconds in computation time.

Our computational experiments showed that WA1BP performed slightly better than a sophisticated procedure based on bounding and tabu search (HI_BP). It performed much better than a variable neighborhood search algorithm (PMBS′ + VNS) and a branch-and-bound procedure with a cutting stock lower bound (MTPCS).

We also developed WA1DPB that solves the one-dimensional dual bin packing problem. Our computational experiments showed that WA1DPB is just as competitive as WA1BP in terms of speed and accuracy.

Chapter 3

The Guillotine Cutting Two-Dimensional Bin Packing Problem

3.1 Introduction

The applications of the two-dimensional bin packing problem (2BP) are very common in the glass, wood, and metal industries where stocks of standard sizes have to be cut into rectangular components to meet demands with minimal material waste. In logistics and warehousing, items of rectangular shapes need to be packed on shelves efficiently to save storage space. In the publishing industries, articles and advertisements have to be paginated on to a minimum number of pages.

In the two-dimensional bin packing problem, we are given n rectangular items, each with a specified width and height, and need to allocate all items to a minimum number of identical rectangular bins, each with height H and width W, without overlapping the items. The items have to be packed with their edges parallel to the edges of the bins.

There are four variants of the two-dimensional bin packing problem as specified in the topology of Lodi, Martello, and Vigo (1999b).

- 2BP|O|G. All items have fixed orientations (O) and are obtained through a sequence of edge-to-edge guillotine (G) cuts that are parallel to the edges of a bin. The guillotine constraint originates from the technological requirements of automated cutting machines.

- 2BP|R|G. The orientations of the items may be rotated (R) through 90˚ and guillotine cutting (G) is required.
- 2BP|O|F. The orientations (O) of the items are fixed and free (F) cutting applies.
- 2BP|R|F. The orientations of the items may be rotated (R) through 90˚ and free (F) cutting applies.

Over the last eight years or so, several methods have been developed to solve each of the two-dimensional bin packing problems (all four variants are NP-hard). A good overview of methods for solving the 2BP developed through the late 1990s and early 2000s including descriptions of upper and lower bounds, exact algorithms, and metaheuristics is given by Lodi, Martello, and Vigo (2002b).

Lodi, Martello, and Vigo (1999a) developed a tabu search algorithm for solving 2BP|O|G and applied their algorithm to problem instances taken from the literature including those proposed by Berkey and Wang (1987). The authors found that the solutions of tabu search were closer to known lower bounds than solutions produced by two well-known procedures (finite first-fit and finite best strip). Lodi, Martello, and Vigo (1998) proposed a tabu search algorithm for 2BP|R|G. Lodi, Martello, and Vigo (1999b) developed a unified tabu search framework (TS) for solving each of the four problems. They considered 10 classes of problems – six from Berkey and Wang (1987) and four from Martello and Vigo (1998) – and found that TS was effective in all cases. Faroe, Pisinger, and Zachariasen (2003) presented a heuristic based on guided local search (GLS) that solved both the three-dimensional and the two-dimensional bin packing

problems. GLS produced solutions that were as good as those produced by TS on the 10 classes of the two-dimensional problems. Monaci and Toth (2006) solved the 10 problem classes for the 2BP|O|F variant using a set covering heuristic (SCH). They compared SCH to the exact algorithm (EA) of Martello and Vigo (2001), the constructive algorithm (HBP) of Boschetti and Mingozzi (2003), TS, and GLS. The authors concluded that SCH is very competitive with the best procedures found in the literature.

In this chapter, we present our weight annealing algorithm for solving the two variants of the two-dimensional bin packing problem with the guillotine cutting constraint, namely, 2BP|O|G and 2BP|R|G.

3.2 Weight Annealing Algorithm for 2BP with Oriented Items and Guillotine Cuts (WA2BPG)

The guillotine cutting constraint arises out of certain cutting problems that require all items be obtained through a sequence of edge-to-edge cuts that are parallel to the edges of a bin. For example, in the glass industry, the cutting machine can only operate in the guillotine mode. For industrial applications related to corrugated irons or wallpaper, the fixed orientation requirement applies. We develop a new heuristic to solve 2BP|O|G in three phases. The first two phases of the algorithm are based on the hybrid first-fit (HFF) algorithm by Chung, Garey, and Johnson (1982).

3.2.1 Hybrid First-Fit

In the first phase of HFF, the items are arranged in the order of non-increasing height. We pack the items from left to right into levels that have an identical width W.

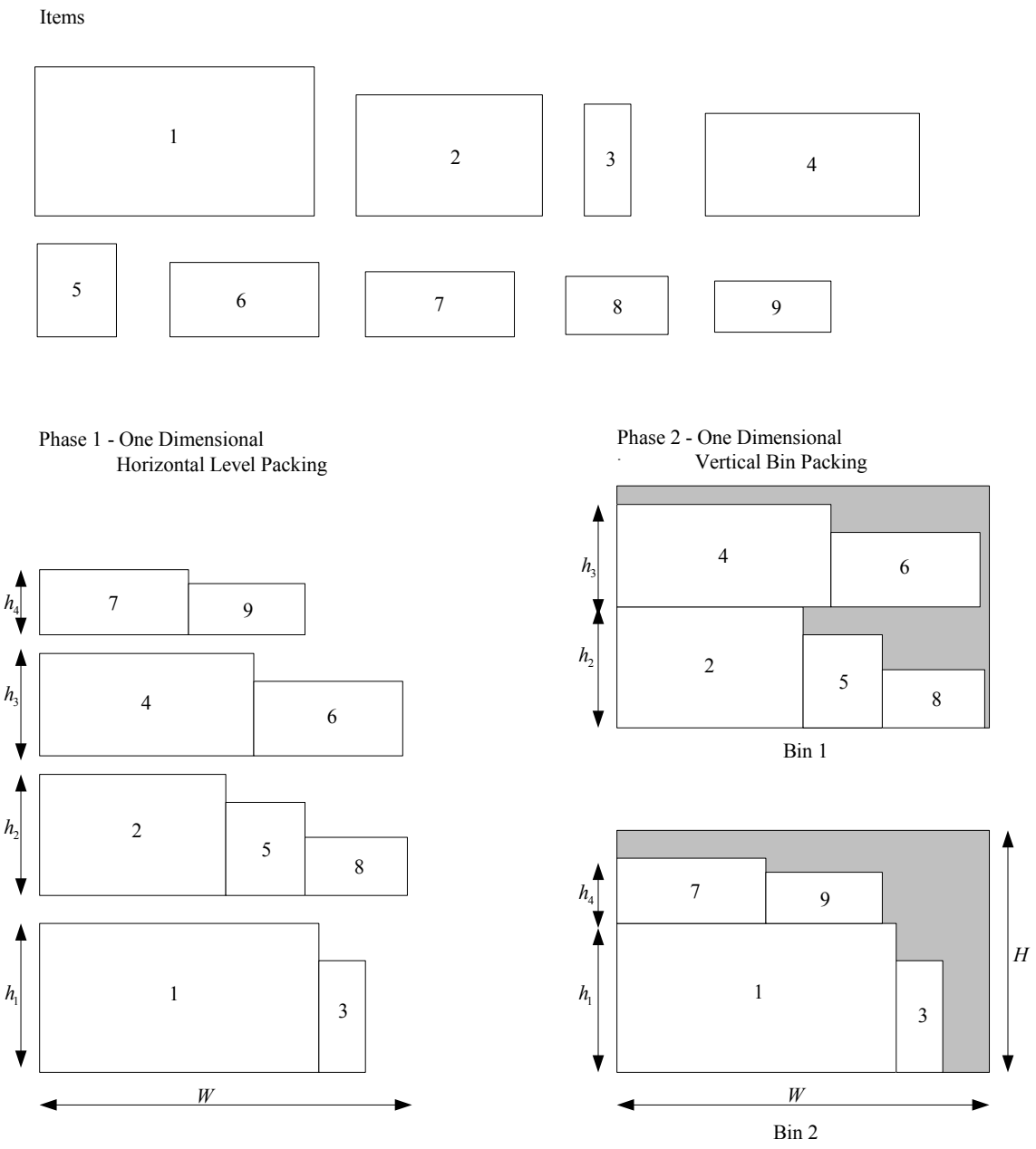


Figure 3.1 An example that shows the two phases of the hybrid first-fit algorithm.

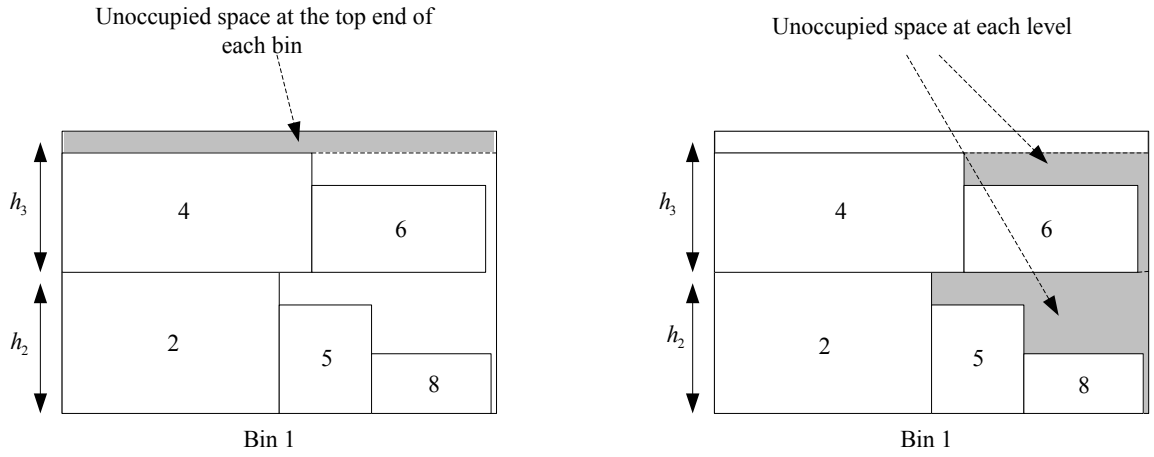


Figure 3.2 An example of unoccupied space created by the hybrid first-fit algorithm.

The height h_i of level i is the height of the tallest item to be allocated to and located at the left end of this level. Following the first-fit decreasing height (FFDH) strategy, we pack an item left justified on the first level that it can fit and, if the item can not fit on any level, we pack it into a new level. At the end of first phase, we will arrive at a solution with $h_1 \geq h_2 \geq h_3 \geq \ldots$. An example of HFF is as shown in Figure 3.1.

In the second phase of HFF, we solve the one-dimensional bin packing problem with item sizes h_i, and bin height H, using a first-fit decreasing algorithm. In Figure 3.1, we see that four strips are packed into two bins, each with height H, without violating the guillotine cutting constraint.

HFF is an approximation algorithm. It can create wasted space at the top right hand corner of each level, and at the top of each vertical bin. In Figure 3.2, we highlight the potential unused space in the levels and bins. Our weight annealing algorithm should address this type of shortcoming of HFF.

3.2.2 Phase 1

3.2.2.1 Initial Solution

We construct an initial solution using a hybrid first-fit procedure (HFF) from Chung, Garey, and Johnson (1982) that we have modified in the following way. We order the items by non-increasing height and select an item for packing with probability 0.5. In other words, we start with the first item on the ordered list and, based on a coin toss, we pack it into a bin if it is selected, or leave it on the ordered list if it is not selected. We continue down the ordered list until an item is selected for packing. We then pack the second item in the same manner, and so on, until we reach the bottom of the list.

3.2.2.2 Objective Function for the Local Search

Using HFF, we pack items into horizontal levels where each level has width b_i and $b_i \leq W$ (bin width). To help minimize the total number of levels that are used, we swap (exchange) items between levels with an objective function that maximizes the sum of the squared level widths b_i, where $b_i = \sum_{j=1}^{m_i} t_{ij}$, m_i is the number of items in level i, and t_{ij} is the width of item j in level i. In the local search, we accept a swap between level i and level k if it results in an increase in $b_i^2 + b_k^2$. Our objective function is given by

$$\text{maximize } f = \sum_{i=1}^{p} b_i^2 \tag{3.1}$$

where p is the number of levels. Our objective function is motivated by the one developed by Fleszar and Hindi (2002) for the one-dimensional bin packing problem.

We illustrate our objective function in Figure 3.3. We have two levels and we

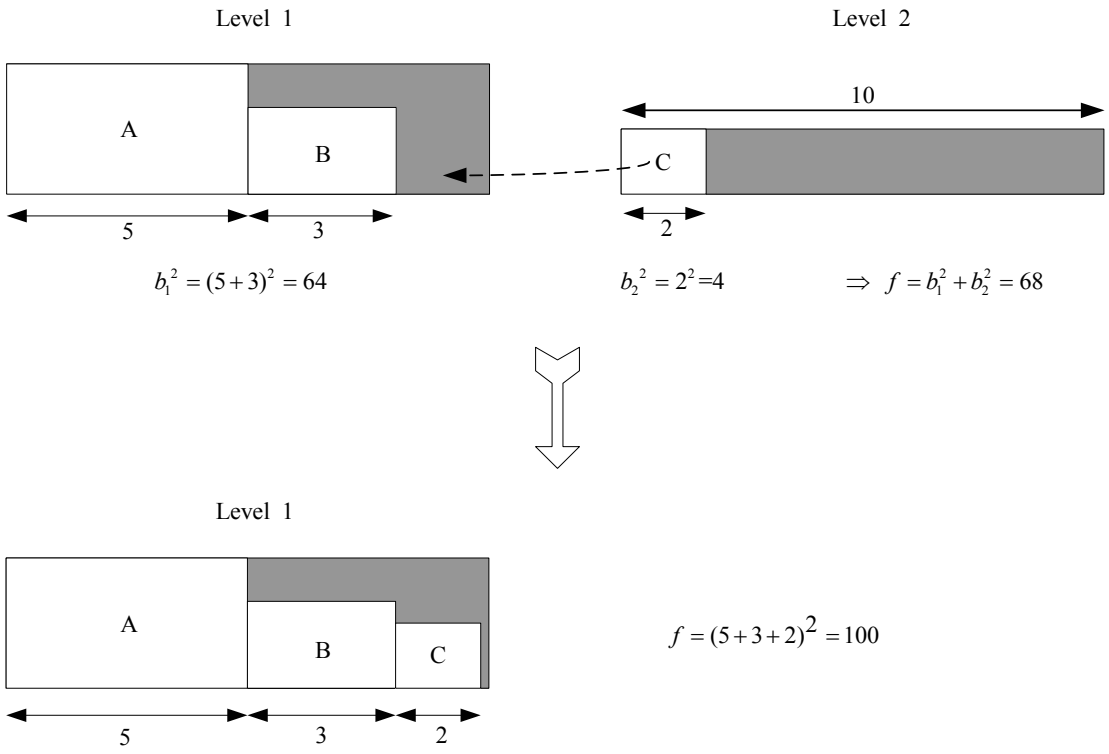


Figure 3.3 Moving one item between two levels (called Swap (1,0)) uses one less level and increases the objective function value.

move item C from level 2 to level 1. This move results in the use of one less level, does not violate the bin width constraint of 10, and increases the objective function value from 68 to 100. We denote the swap of one item between levels as Swap (1,0).

The objective function (3.1) is equivalent to minimizing the number of levels, but does not attempt to reduce the unused area in each level (this is a key weakness of HFF). In Figure 3.4, we see that swapping item B in level 1 with item C in level 2 reduces the unused area to zero and reduces the sum of heights $(h_1 + h_2)$, even though the values of $(b_1^2 + b_2^2)$ before and after the swap are the same.

We would like our objective function to minimize the number of levels used and

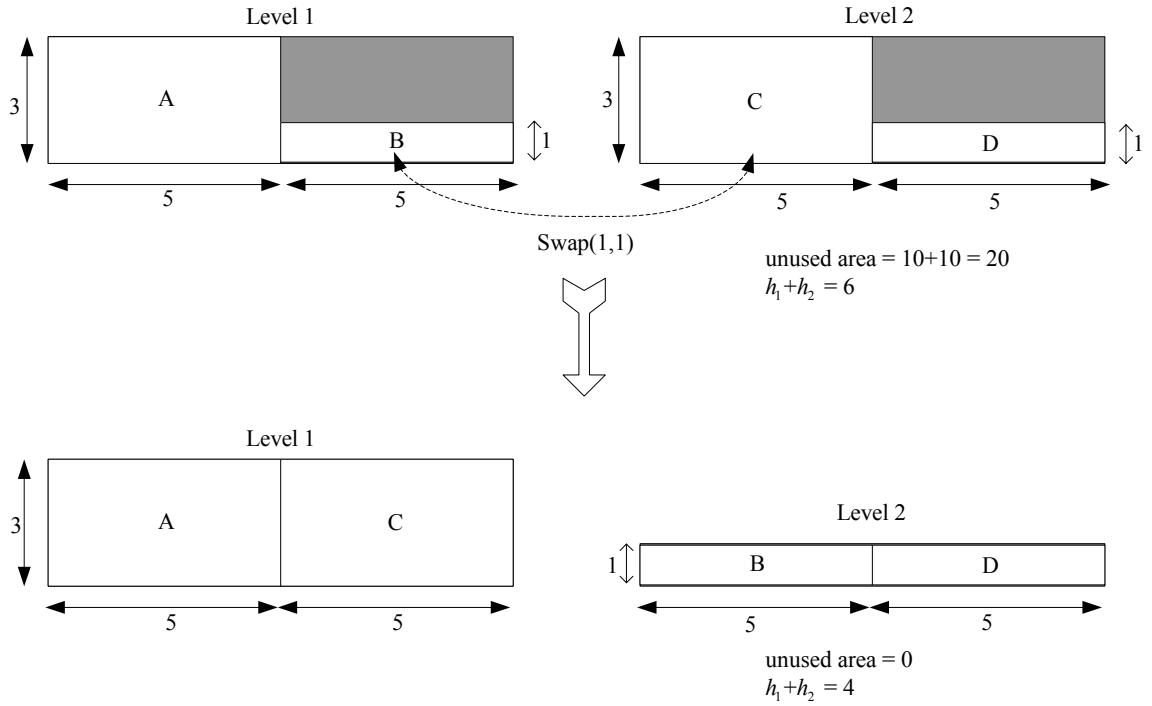


Figure 3.4 An example of the need to reduce the unused areas of two levels with equal width.

also minimize the sum of the heights of the levels. We accomplish this with the following objective function

$$\text{maximize } f = \sum_{i=1}^{p} b_i^2 - \sum_{i=1}^{p} (Wh_i - A_i) \tag{3.2}$$

where h_i is the height of level i and A_i is the sum of the areas of all items in bin i (that is, $A_i = \sum_{j=1}^{m_i} a_{ij}$ where m_i is the number of items in level i and a_{ij} is the area of item j in level i).

To summarize, in Phase 1 using the objective function (3.2), we try to pack all of the items into a minimum number of levels with minimum wasted space. Our local search procedure uses three types of swaps: Swap (1,0), Swap (1,1) (swap one item from a level

with one item from a different level), and Swap (1, 2) (swap one item from a level with two items from a different level).

3.2.3 Phase 2

Using the solution produced in Phase 1, we apply a first-fit decreasing algorithm to generate an initial solution for the one-dimensional bin packing problem with items of size $h_{ij,}$ where h_{ij} is the height of level j in bin i, and bins of height H. Let d_i be the stack height which is the sum of the heights of the levels in each bin. In order to minimize the total number of bins, we use swap schemes that exchange levels between all pairs of bins with an objective function that maximizes the sum of the squares of stack heights d_i, where $d_i = \sum_{j=1}^{m_i} h_{ij}$, and m_i is the number of levels in bin i. In the local search, we accept a swap between level i and level k if it results in an increase in $d_i^2 + d_k^2$. Our objective function is given by

$$\text{maximize} f = \sum_{i=1}^{q} d_i^2 \tag{3.3}$$

where q is the number of bins. As in Phase 1, we use the three swap schemes (Swap(1,0), Swap(1,1), and Swap(1,2)) between all pairs of bins.

3.2.4 Phase 3

This phase can be regarded as post-optimization in which we try to fill unused space. We look at unused space within a level and the unused space at the top of a bin.

3.2.4.1 Unoccupied Space within Each Level

In order to fill the unused space within a level, we partition the level with a grid system that preserves the guillotine cutting constraint. In Figure 3.5, we show four ways of partitioning the unused space. In option 1, there are vertical partitions that originate from the top of each item. In option 2, there are horizontal partitions at the top right of each item. In option 3, both vertical and horizontal partitions are used. In option 4, there is a horizontal partition at the top of the level and vertical partitions beneath it (this is the option we use in our algorithm). We allow a feasible move to occupy a partition starting at its left edge or the remaining space to the right of an item already in that partition, but not the space above the item. In Figure 3.5, option 4, we show four items that have been moved in this way to fill unused space.

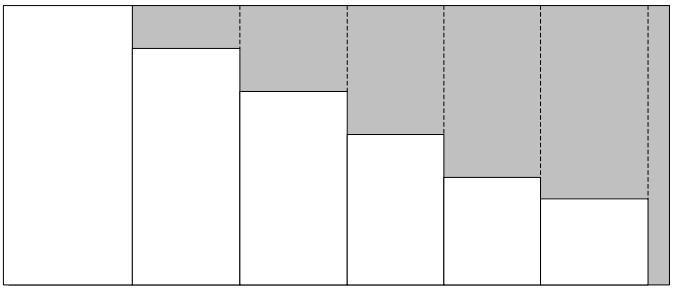
In the local search, our objective function is given by

$$\text{maximize } f = \sum_{i=1}^{q} A_i^2 \tag{3.4}$$

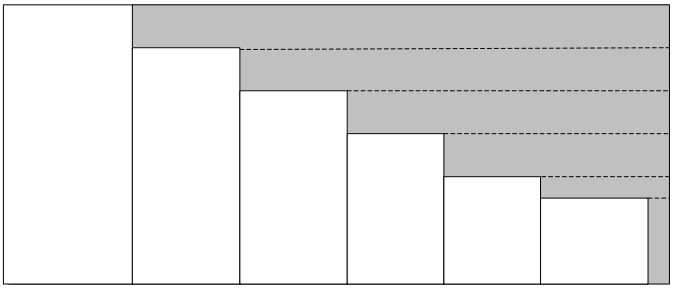
where q is the number of bins and A_i is the sum of the areas of all items in bin i.

For ease of implementation, we use only Swap(1,0) moves. Within each partition, only the remaining space to the right of an item can be filled up by additional items. In other words, the first item that we move to fill an empty space will have its left edge touching the left side of the partition. The next item will have its left edge touching the right edge of the first item and so on, as along as the sum of the item widths does not exceed the width of the partition. In Figure 3.6, we move all items from one bin to fill the unoccupied space of another bin, and this results in the use of one less bin.

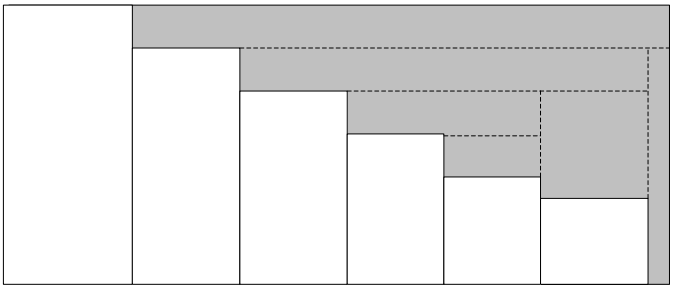
Option 1: vertical cells



Option 2: horizontal cells



Option 3: hybrid



Option 4: one long horizontal cell plus vertical cells

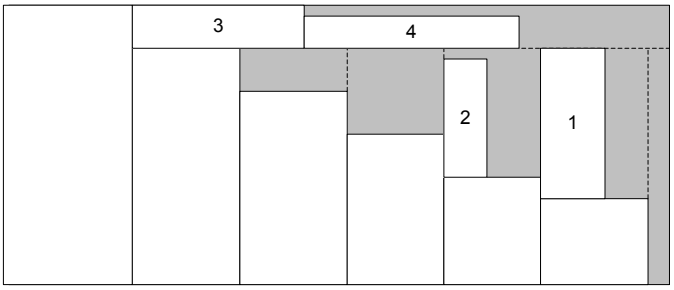


Figure 3.5 Options for partitioning the unused space.

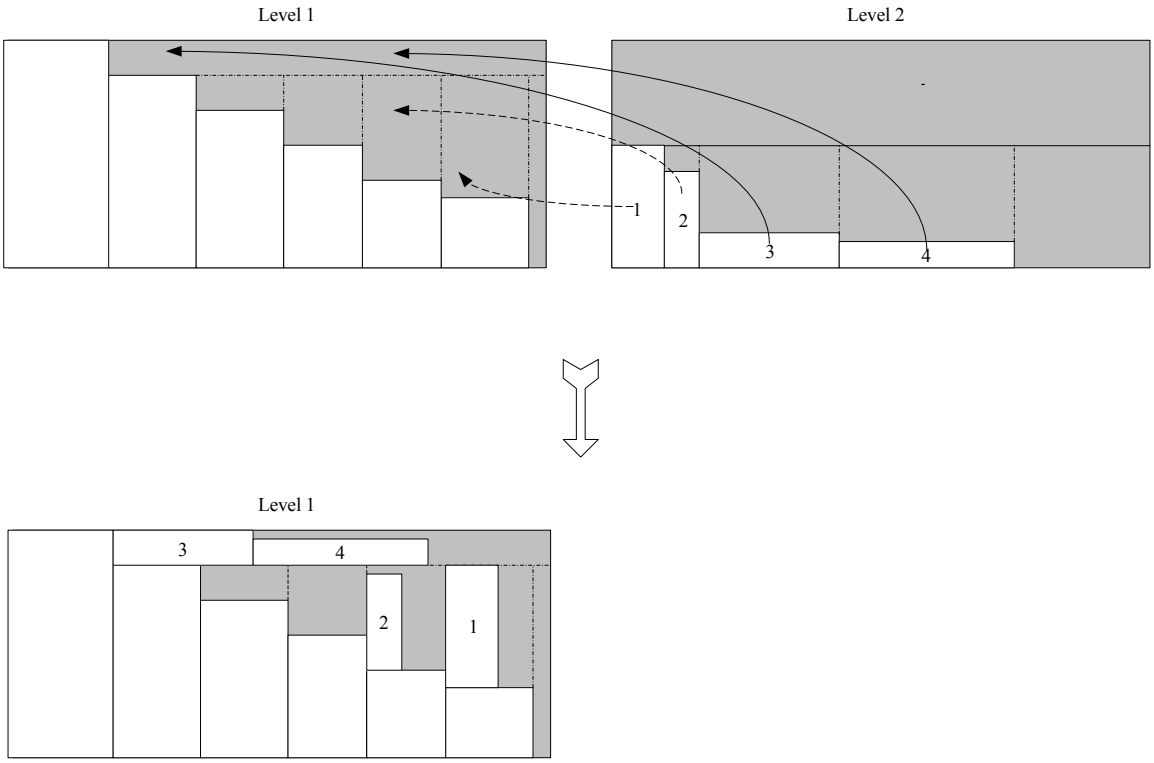


Figure 3.6 Filling unused space within each level.

3.2.4.2 Unoccupied Space at the Top of Each Bin

Using the objective function (3.4), we are able to fill the unoccupied space at the top of each bin as much as possible with Swap(1,0) moves. In Figure 3.7, we illustrate this type of move.

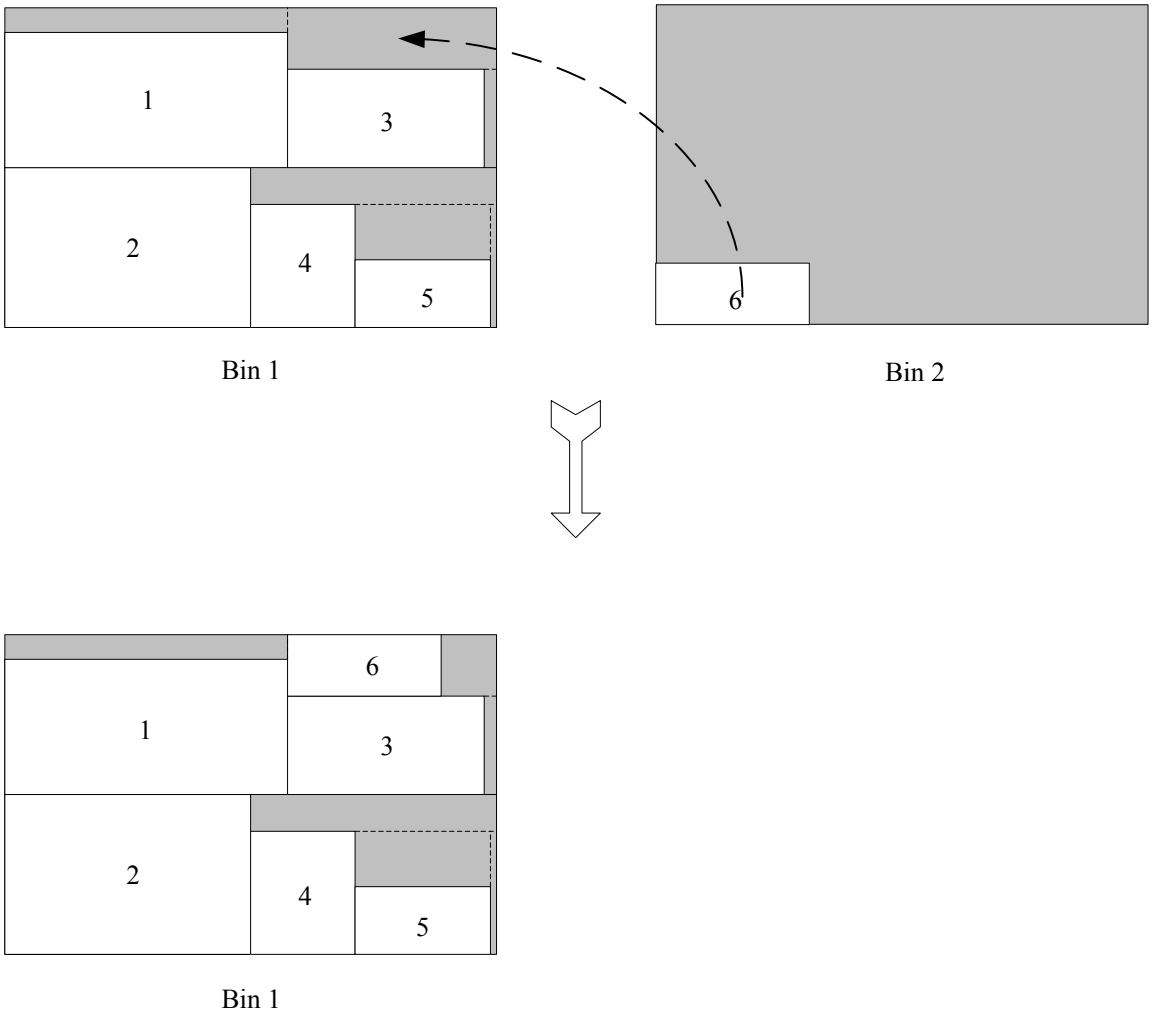


Figure 3.7 Filling unused space at the top of each bin.

3.2.5 Weight Assignments

In our algorithm, we assign different weights to the bins and levels, and their items according to how well the bins and levels are packed. This distortion of item sizes allows for both uphill and downhill moves and is a key feature of our algorithm.

In Phase 1, for each level i, we assign weight w_i^T according to

$$w_i^T = (1 + Kr_i)^T \tag{3.5}$$

where W is the width of each bin, b_i is the width of level i, K is a constant, T is a temperature parameter, and the residual capacity of level i is $r_i = (W - b_i)/W$. The scaling parameter K controls the amount of size distortion for each item. The size distortion for an item is proportional to the residual capacity of its level.

In Phase 2, for bin i, we compare the bin height H to the stack height d_i and assign weight w_i^T according to

$$w_i^T = (1 + Kr_i)^T \tag{3.6}$$

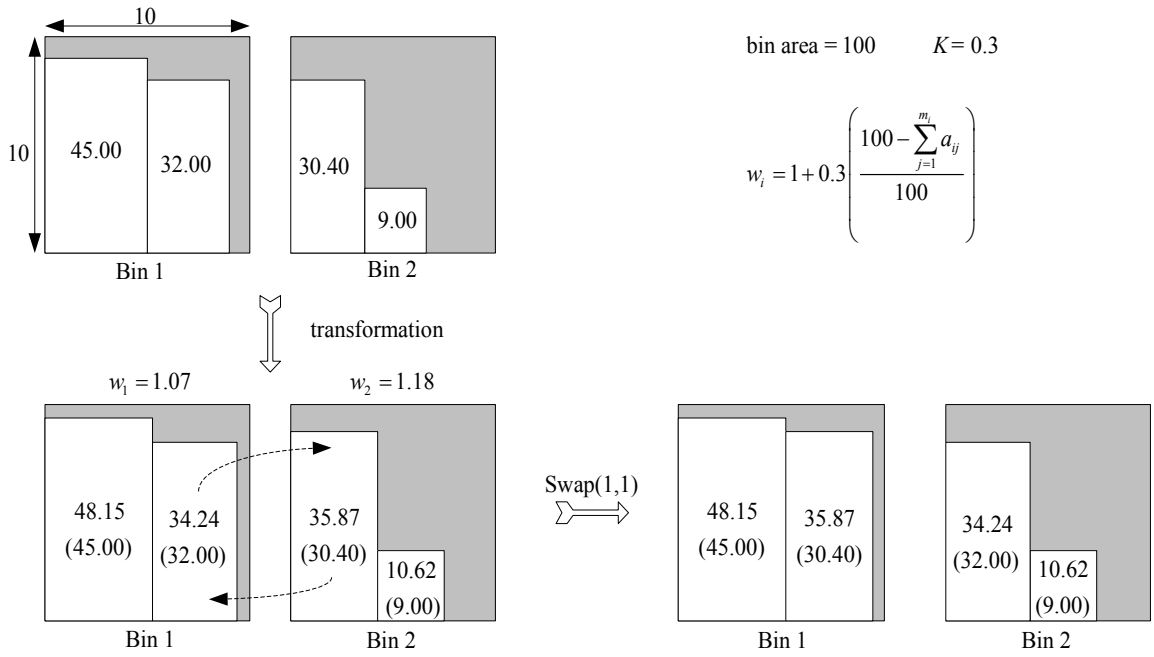
where the residual capacity of bin i is $r_i = (H - d_i)/H$.

In Phase 3, for bin i, we compare A_i (the sum of the areas of all items in bin i) to the available bin area (HW) and assign weight w_i^T according to

$$w_i^T = (1 + Kr_i)^T \tag{3.7}$$

where the residual capacity of bin i is $r_i = (HW - A_i)/(HW)$.

The use of weights increases the sizes of items in poorly packed bins and helps our algorithm to escape a poor local maximum through downhill moves. We illustrate this in Figure 3.8 where the bin area is 100 and $K = 0.3$, and we make a Swap (1,1) move. We see that we have an uphill move in the transformed space (the objective function value increases) which is actually a downhill move in the original space (the objective function value decreases). We make a move as long as it is feasible in the original space. (We point out that, after the transformation, but before the swap, item sizes in bin 1 are $48.15 = 1.07 \times 45$ and $34.24 = 1.07 \times 32$.)



Transformed space:

$$f' = (48.15 + 34.24)^2 + (35.87 + 10.62)^2 = 8949.43$$

$$f'_{new} = (48.15 + 35.87)^2 + (34.24 + 10.62)^2 = 9071.78$$

Original space:

$$f = (45.00 + 32.00)^2 + (30.40 + 9.00)^2 = 7481.36$$

$$f_{new} = (45.00 + 30.40)^2 + (32.00 + 9.00)^2 = 7366.16$$

Figure 3.8 A feasible uphill move in the transformed space is a downhill move in the original space in 2BP.

3.2.6 Weight Annealing Algorithm

In Table 3.1, we give our weight annealing algorithm for both variants of the two-dimensional bin packing problem with guillotine cuts (2BP|O|G, 2BP|R|G). We denote our algorithm by WA2BPG.

In Phase 1, WA2BPG starts with an initial solution that is generated by our modified hybrid first-fit procedure. Swapping operations with weight annealing are used to improve a solution. A temperature parameter (T) controls the amount by which a single

weight can be varied. At the start, a high temperature (T = 1) allows for higher frequencies of downhill moves. As the temperature is gradually cooled (the temperature is reduced at the end of every iteration, that is, $T \times 0.95$), the amount of item distortion decreases and the problem space looks more like the original problem space.

We compute a weight for each level (according to $w_i^T = (1 + Kr_i)^T$) and then apply the weight to the width of each item in the level. The swapping process begins by comparing the items in the first level with the items in the second level, and so on, sequentially down to the last level in the initial solution and is repeated for every possible pair of levels.

For a current pair of levels (α, β), the swapping of items by Swap (1,0) is carried out as follows. The algorithm evaluates whether the first item (item *i*) in level α can be moved to level β without violating the width constraint of level β in the original space. In other words, does level β have enough original residual capacity to accommodate the original width of item *i*? If the answer is yes (the move is feasible), the change in objective function value of the move in the transformed space is evaluated. If the change in objective function value is nonnegative, then item *i* is moved from level α to level β. After this move, the algorithm exits Swap (1,0) and proceeds to Swap (1,1). If the move of the first item is infeasible or the change in objective function value is negative, then the second item in level α is evaluated and so on, until a feasible move with a nonnegative change in objective function value is found or all items in level α have been considered and no feasible move with a nonnegative change has been found. The algorithm then performs Swap (1,1) followed by Swap (1,2). In each of the swapping schemes, we always make the first feasible swap that has a nonnegative change in the

objective function value. We point out that the improvement step is carried out 100 times (*nloop2* = 100) starting with T =1, followed by $T = 1 \times 0.95 = 0.95$, $T = 0.95 \times 0.95 = 0.9025$, etc.

In Phase 2, we solve a one dimensional bin packing problem treating each level as an item with size (height) h_{ij}. For bin *i,* we compute the stack height $d_i = \sum_{j=1}^{m_i} h_{ij}$, the residual capacity (r_i) based on the bin height H, and its weight w_i. We apply the same weight to all levels in the bin. We swap levels between bins and stop making swaps when the lower bound for the number of bins is reached. The improvement step is carried out 50 times (*nloop3* = 50) or until the lower bound is reached (we discuss the lower bounds in more detail in Section 3.4 on computational results).

In Phase 3, we try to move one item between bins to fill unused space. We start by determining the locations and sizes of unused space. For each bin *i,* we compute A_i (the sum of the areas of all items in bin *i*), the residual capacity r_i, and the weight w_i for each bin, and apply the same weight to all items in bin *i*. The improvement step is carried out 50 times (*nloop3* = 50) or until the lower bound is reached.

If we have not obtained the lower bound at the end of the first optimization run, we start another run with a new initial solution generated by our modified hybrid first-fit algorithm in Phase 1. We terminate the algorithm as soon as the lower bound is reached or after 20 runs (*nloop1* = 20).

Step 0. Initialization
Parameters are K (scaling parameter), *nloop1, nloop2, nloop3*, T (temperature), and *Tred*
Set $K = 0.05$, *nloop1* = 20, *nloop2* = 100, *nloop3* = 50, $T = 1$, and *Tred* = 0.95
Inputs are height and width of each item, height and width of a bin, and lower bound (*LB*)

Step 1. Optimization runs
for $k = 1$: *nloop1* **do**

Step 1.1 Perform Phase 1
Construct an initial solution using modified hybrid first-fit procedure
set $T := 1$
for $j = 1$: *nloop2* **do**
Compute weight of level $i := w_i^T$ and weighted width of item j for all j
Do for all pairs of levels{
Swap items between two levels
Allow item rotations for 2BP|R|G
Perform Swap (1,0)
Perform Swap (1,1)
Perform Swap (1,2)
}
$T := T \times Tred$
end

Step 1.2 Perform Phase 2
set $T := 1$
for $j = 1$: *nloop3* **do**
Compute weight of bin $i := w_i^T$ and weighted height of level j for all j
Do for all pairs of bins{
Swap levels between two bins.
Perform Swap (1,0), exit j loop and k loop if *LB* is reached
Perform Swap (1,1), exit j loop and k loop if *LB* is reached
Perform Swap (1,2), exit j loop and k loop if *LB* is reached
}
$T := T \times Tred$
end

Step 1.3 Perform Phase 3
Determine the locations and sizes of the partitions
set $T := 1$
for $j = 1$: *nloop3* **do**
Compute weight of bin $i = w_i^T$ and weighted area of item j for all j
Do for all pairs of bins{
Perform Swap (1,0), exit j loop and k loop if *LB* is reached
Allow item rotations for 2BP|R|G
}
$T := T \times Tred$
end
end

Step 2. Outputs are the number of bins and the final distribution of items

Table 3.1 Weight annealing algorithm for 2BP|O|G and 2BP|R|G.

3.2.7 Number of Parameters

There are five parameters in our weight annealing algorithm, and this is fewer than the number of parameter used in tabu search. In Table 3.2, we summarize the parameters for our weight annealing algorithm and for the tabu search algorithm of Lodi, Martello, and Vigo (1998).

3.3 Weight Annealing Algorithm for 2BP with Non-Oriented Items and Guillotine Cuts (WA2BPG)

We now describe the modifications to our algorithm that are needed to solve problems with items that can be rotated. WA2BPG solves the 2BP|R|G instances by allowing item rotations during Phase 1 and Phase 3.

During Phase 1, we allow for the rotation of an item through 90° to minimize the unused space within each level and each bin. In order to reduce computation time, we allow for only a feasible move to occupy a partition starting from its left edge or the remaining space to the right of an item already in the partition, but not the space above the item. We would like to rotate an item through 90° if this produces a tighter fit or results in a greater utilization of the unused space above an item that has its original orientation. We illustrate rotating two items in Figure 3.9. These rotations free a substantial amount of space to the right of the two items and this space can now be used by other items. In WA2BPG, during Phase 1, we allow an item rotation if it reduces a level's width (b_i), or if it results in a feasible swap with a nonnegative change in the objective function value.

S/No	Weight Annealing Parameters	Tabu Search Parameters
1.	Size distortion factor $K = 0.05$	Relative weight of the two terms defining bin weakness $\alpha = 5.0$
2.	Value of temperature reduction $Tred = 0.95$	Length of tabu tenure list 1 $\tau_1 = 3$
3.	Maximum number of iterations for loop 1 $nloop1 = 20$	Length of tabu tenure list 2 $\tau_2 = 5$
4.	Maximum number of iterations for loop 2 $nloop2 = 100$	Number of restarts for type R_1 $R = 5$
5.	Maximum number of iterations for loop 3 $nloop3 = 50$	Number of restarts for type R_2 $T = L/5$ (overall time limit L=100 sec)
6.		Number of moves in the second neighborhood search $\mu = 50$

Table 3.2 Summary of weight annealing and tabu search parameters.

During Phase 3 of WA2BPG for 2BP|O|G, an item from one bin will be moved into the unused space within a level or at the top of a bin if the move is feasible and improving. In Figure 3.10, item 6 has been rotated and moved from bin 2 to bin 1 and now occupies two types of unused spaces -- within a level and at the top of a bin.

3.4 Computational Experiments and Results

We coded our algorithms in C/C++, and ran them on a desktop computer with a 3 GHz Pentium 4 processor and 256 MB of RAM. For the test problems, we consider the widely cited six classes of benchmark problems from Berkey and Wang (1987) and the four classes from Martello and Toth (1998).

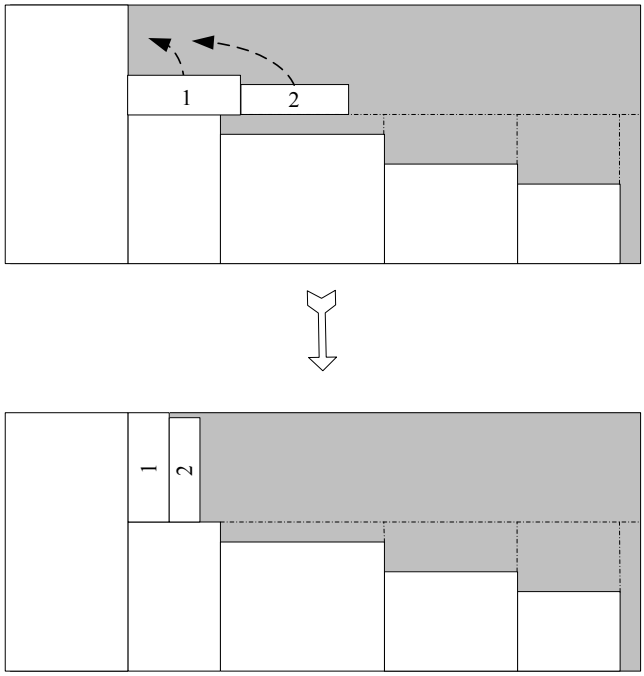


Figure 3.9 Rotating items through 90° to produce a tighter packing.

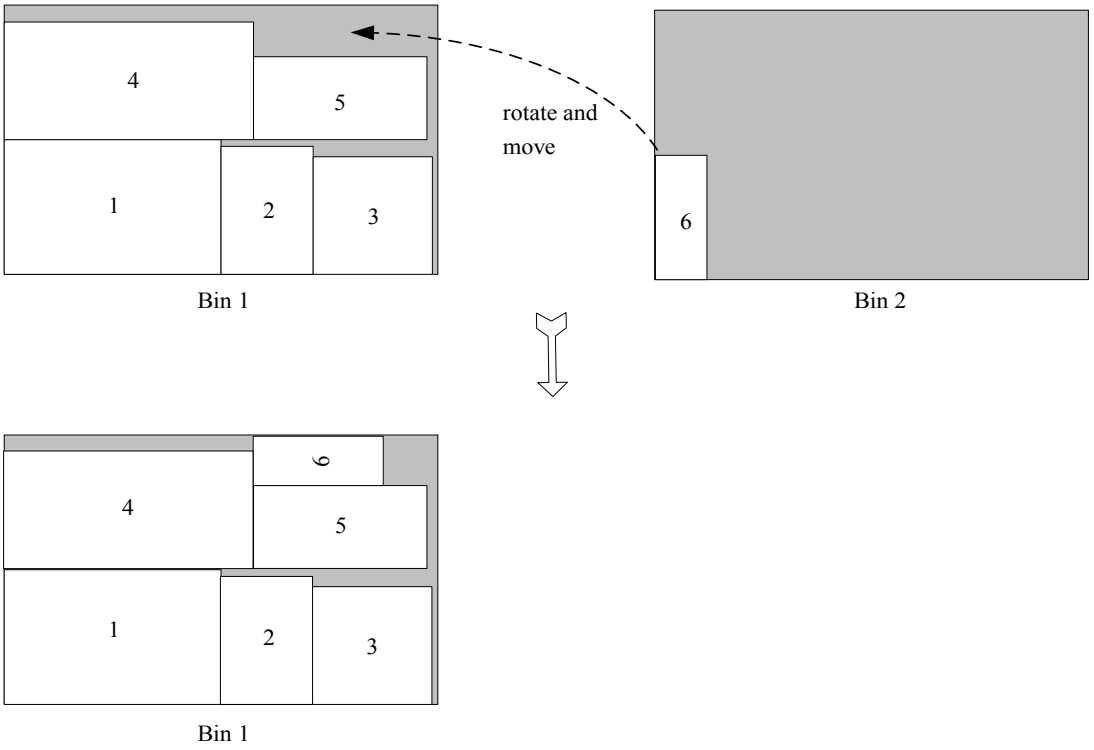


Figure 3.10 Rotating an item through 90° and moving it to another bin.

Class	Item height, width	Bin height (H), width (W)
I	U[1, 10]	10
II	U[1, 10]	30
III	U[1, 35]	40
IV	U[1, 35]	100
V	U[1, 100]	100
VI	U[1, 100]	300

Table 3.3 Six classes of problems from Berkey and Wang (1987).

	Type Probabilities (%)			
Class	1	2	3	4
VII	70	10	10	10
VIII	10	70	10	10
IX	10	10	70	10
X	10	10	10	70

Type	Item width	Item height
1	U[2/3W, W]	U[1, 1/2H]
2	U[1, 1/2W]	U[2/3H, H]
3	U[1/2W, W]	U[1/2H, H]
4	U[1, 1/2W]	U[1, 1/2H]

$H = W = 100$

Table 3.4 Four classes of problems from Martello and Vigo (1998).

3.4.1 Benchmark Problems

In Table 3.3, we describe the six classes of randomly generated benchmark test problems from Berkey and Wang (1987). The height and width of an item are selected from a uniform distribution. The height and width of a bin are the same value (e.g., in Class I, a bin has $H = W = 10$). For each class, we set $n = 20, 40, 60, 80, 100$ and generate 10 instances; this produces 300 test instances.

In Table 3.4, we describe the four classes of randomly generated benchmark test problems from Martello and Vigo (1998). There are four types of items where the height and width of an item are selected from a uniform distribution with $H = W = 100$. Each class of problems is a mixture of the four item types (e.g., 70% of the items in Class VII are Type 1). For each class, we set $n = 20, 40, 60, 80, 100$ and generate 10 instances; this produces 200 test instances. Overall, we have a total of 500 test instances (the test problems of Berkey and Wang (1987) and Martello and Vigo (1998) are available at http://www.or.deis.unibo.it/research.html.)

3.4.2 Results for 2BP|O|G and 2BP|R|G

In Tables 3.5 and 3.6, we show the results generated by WA2BPG on the 2BP|O|G and 2BP|R|G problems. WA2BPG used 7,373 bins and needed 24.77 seconds to solve the problems with oriented items, while it used 7,279 bins and needed 44.01 seconds to solve the problems with rotated items.

Lodi, Martello, and Vigo (1999b) published results produced by TS for the 10 classes of 2BP|O|G and 2BP|R|G problems. Comparisons based on average ratios cannot be made between WA2BPG and TS because the lower bounds used by Lodi, Martello, and Vigo were not published in their paper. Lodi, Martello, and Vigo (1998) published results produced by TS for 10 classes of 2BP|R|G problems. The ratios reported in the 1998 paper are different from the values reported in the 1999b paper. Lodi (2005) commented that the experiments in the two papers were run on different machines and probably with different parameter settings.

			WA2BPG				WA2BPG
Class	*n*	Bins	Time(s)	Class	*n*	Bins	Time(s)
I	20	72	0.12	VI	20	10	0.04
	40	137	0.06		40	19	0.07
	60	202	1.78		60	22	0.05
	80	277	0.72		80	30	0.06
	100	326	0.24		100	35	0.07
II	20	10	0.05	VII	20	56	0.07
	40	20	0.04		40	115	0.29
	60	26	0.41		60	164	0.22
	80	33	0.06		80	233	0.32
	100	40	0.07		100	275	0.27
III	20	54	0.04	VIII	20	60	0.06
	40	98	0.21		40	116	0.18
	60	143	0.94		60	163	0.60
	80	196	2.36		80	230	0.17
	100	230	2.28		100	283	0.17
IV	20	10	0.05	IX	20	143	0.19
	40	20	0.04		40	279	0.04
	60	26	0.06		60	438	0.12
	80	33	0.06		80	577	0.16
	100	39	0.19		100	695	0.23
V	20	67	0.04	X	20	44	0.15
	40	123	0.13		40	77	0.06
	60	185	0.35		60	105	0.12
	80	251	4.28		80	131	2.69
	100	291	3.45		100	164	0.34
				Total		7373	24.77

Table 3.5 Number of bins and running times for WA2BPG on 10 classes of 2BP|O|G problems.

		WA2BPG				WA2BPG	
Class	*n*	Bins	Time(s)	Class	*n*	Bins	Time(s)
I	20	71	0.04	VI	20	10	0.04
	40	136	0.04		40	19	0.05
	60	201	3.47		60	22	0.05
	80	275	0.93		80	30	0.05
	100	323	0.37		100	34	1.96
II	20	10	0.04	VII	20	56	0.07
	40	20	0.24		40	110	0.83
	60	26	0.51		60	158	2.53
	80	32	0.88		80	227	1.21
	100	39	0.59		100	273	2.80
III	20	52	0.05	VIII	20	58	0.92
	40	95	0.24		40	113	0.07
	60	141	1.39		60	159	0.08
	80	193	0.81		80	223	1.97
	100	229	4.67		100	275	0.40
IV	20	10	0.05	IX	20	143	0.02
	40	19	0.80		40	278	0.05
	60	25	0.12		60	437	0.10
	80	33	0.06		80	577	0.15
	100	38	0.17		100	695	0.23
V	20	66	0.06	X	20	42	0.21
	40	120	0.48		40	74	0.08
	60	181	4.51		60	102	0.88
	80	248	3.35		80	130	0.33
	100	288	4.70		100	163	0.36
				Total		7279	44.01

Table 3.6 Number of bins and running times for WA2BPG on 10 classes of 2BP|R|G problems.

3.5 Conclusions

We developed highly competitive weight annealing algorithms to solve the two variants of the two-dimensional bin packing problem with the guillotine cutting constraint. The key features of weight annealing are its simplicity and the ease of implementation. Our computational experiment with 500 benchmark problems showed that the weight annealing algorithms produced high-quality results very quickly.

Chapter 4

The Free Cutting Two-Dimensional Bin Packing Problem

4.1 Introduction

In this chapter, we develop an algorithm for the two-dimensional bin packing problem with free cutting. Guillotine cutting may not be imposed in some cutting problems. For example, in the steel or rubber industries, free cutting of items to meet demands may reduce trim losses and save material costs.

4.2 Weight Annealing Algorithm for 2BP with Oriented Items and Free Cuts (WA2BPF)

4.2.1 Alternate Directions Algorithm

For problems with free cuts, Lodi, Martello, and Vigo (1999b) developed an alternate directions algorithm that exploited non-guillotine patterns by packing items in alternate directions. We adopt this feature for packing bins in our weight annealing algorithm. Specifically, we sort items by non-increasing height and then pack bands of items in alternate directions. We start by packing the first band from left to right at the bottom of a bin using a best-fit decreasing strategy. The first item in this band is placed in the lowest position with its left edge touching the left edge of the bin. The second item is placed in the lowest position with its left edge touching the right edge of the first item. We then pack all subsequent items in the same way as the second item until no items can be inserted into the band. In this way, we have produced the first left-to-right band. We now pack items in the opposite direction with the first right-to-left band above the first

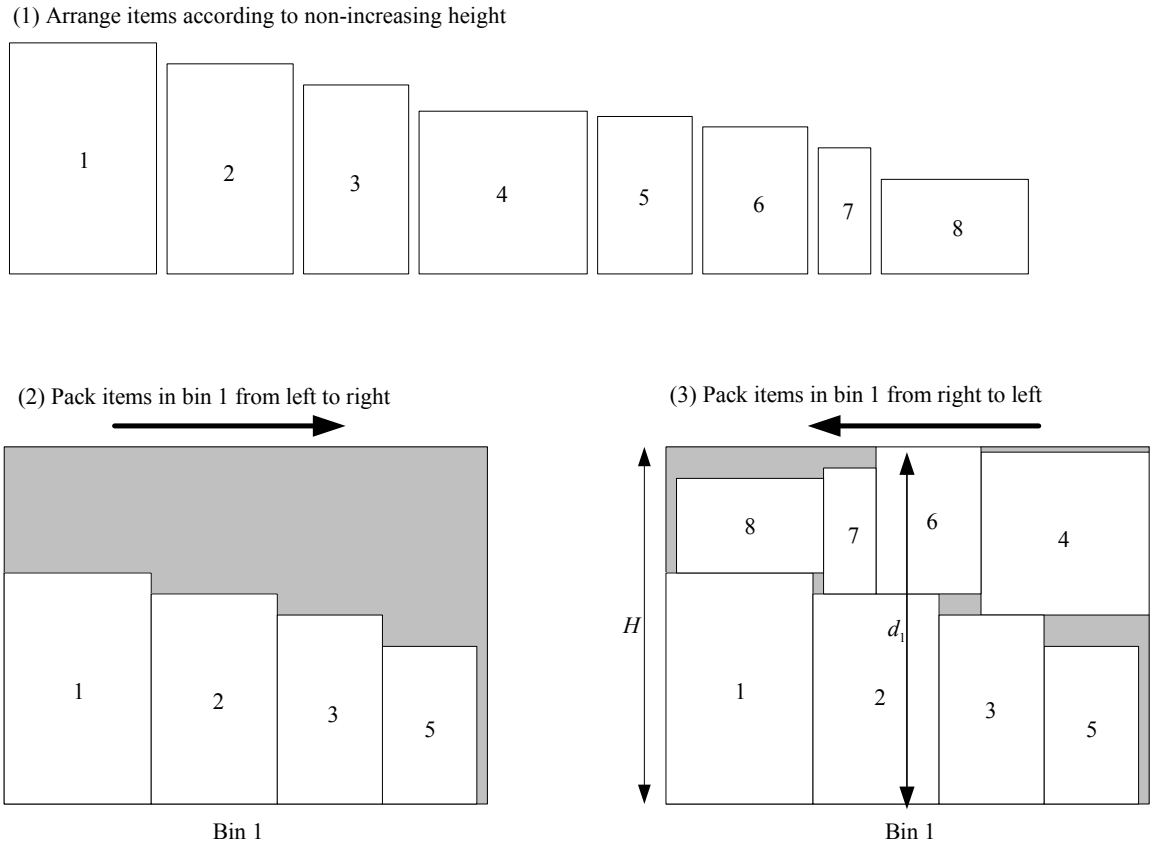


Figure 4.1 Packing items into a bin with the alternate directions algorithm.

left-to-right band in the lowest position. We continue to pack items in alternate directions as long as the bin height constraint is not violated, or the stack height, which is defined as the top edge of the highest item amongst the stack of items in the bin, is less than the bin height. In Figure 4.1, we show how items are packed using the alternate-directions algorithm. We see that the stack height of items in bin 1 (this is the sum of the heights of item 2 and item 6) is $d_1 \leq H$.

4.2.2 Initial Solution

A feasible solution to 2BP|O|G is also a feasible solution to 2BP|O|F. We use the final solution to 2BP|O|G produced by WA2BPG as our initial solution.

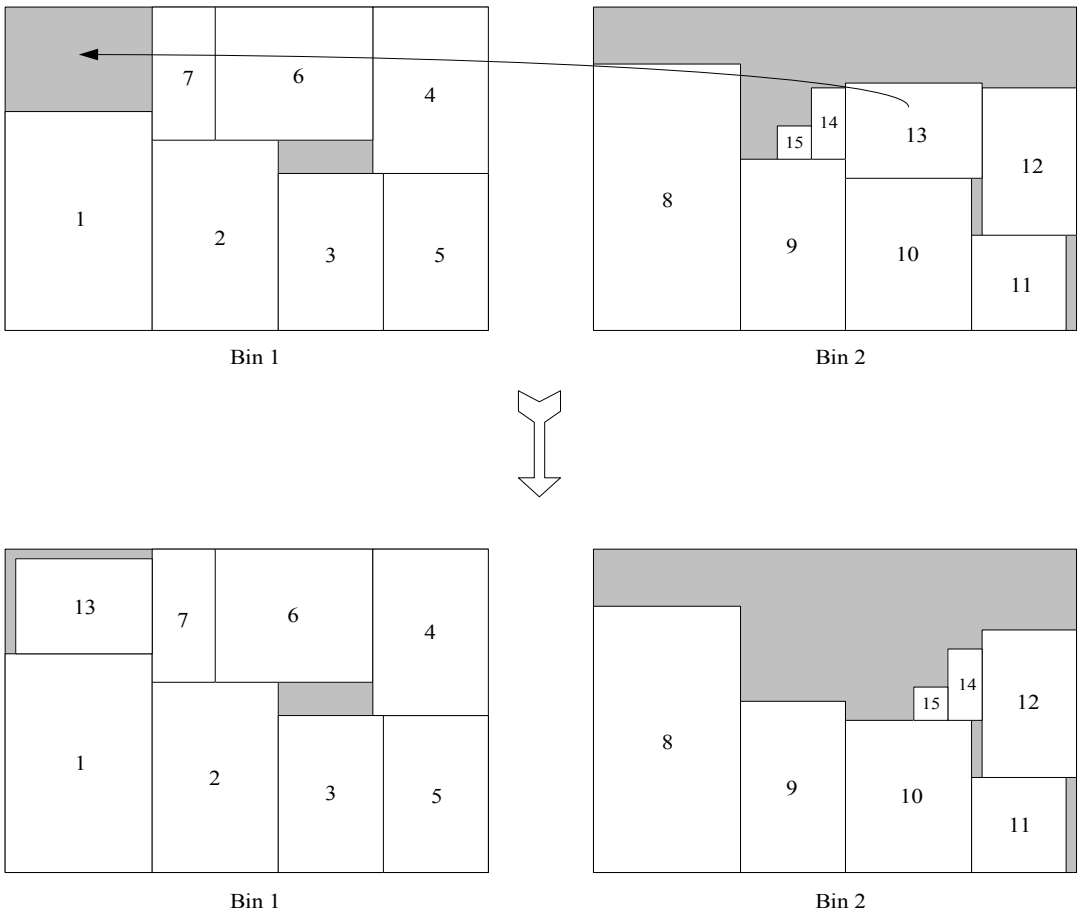


Figure 4.2 Moving an item from bin 2 to bin 1 and then repacking bin 2.

4.2.3 Objective Function for the Local Search

In the local search, our objective function is given by

$$\text{maximize } f = \sum_{i=1}^{q} A_i^2 \tag{4.1}$$

where q is the number of vertical bins and A_i is the sum of the areas of all items in bin i.

We select a pair of bins, swap items between bins (we can use Swap (1,0), Swap (1,1), and Swap (1,2)), and then repack each bin with the alternate directions algorithm. If

a swap between bins is feasible and results in a nonnegative change in the objective function value, then we make the move. If not, we select another pair of bins for evaluation and continue for all pairs of bins. In Figure 4.2, we move one item between bins with Swap (1,0) and then repack a bin.

4.2.4 Post-optimization Processing

When packing items into a bin, there can be dead spaces that we can try to fill in a post-optimization process. In Figure 4.3, we show an example of three dead spaces in a bin. We use Swap (1,0) to move item 15 from bin 2 to the dead space in bin 1. We make this type of move in order to empty a less-filled bin ($A_2 < A_1$) which results in a larger objective function value.

4.2.5 Weight Assignments

Clearly, for a solution to be feasible, we must have its stack height less than the bin height. For bin i, we compare the bin height H to the stack height d_i and assign weight w_i^T according to

$$w_i^T = (1 + Kr_i)^T \tag{4.2}$$

where the residual capacity of bin i is $r_i = (H - d_i)/H$.

4.2.6 Weight Annealing Algorithm

In Table 4.1, we give our weight annealing algorithm for both variants of the two-dimensional bin packing problem with free cuts (2BP|O|F, 2BP|R|F). We denote our algorithm by WA2BPF.

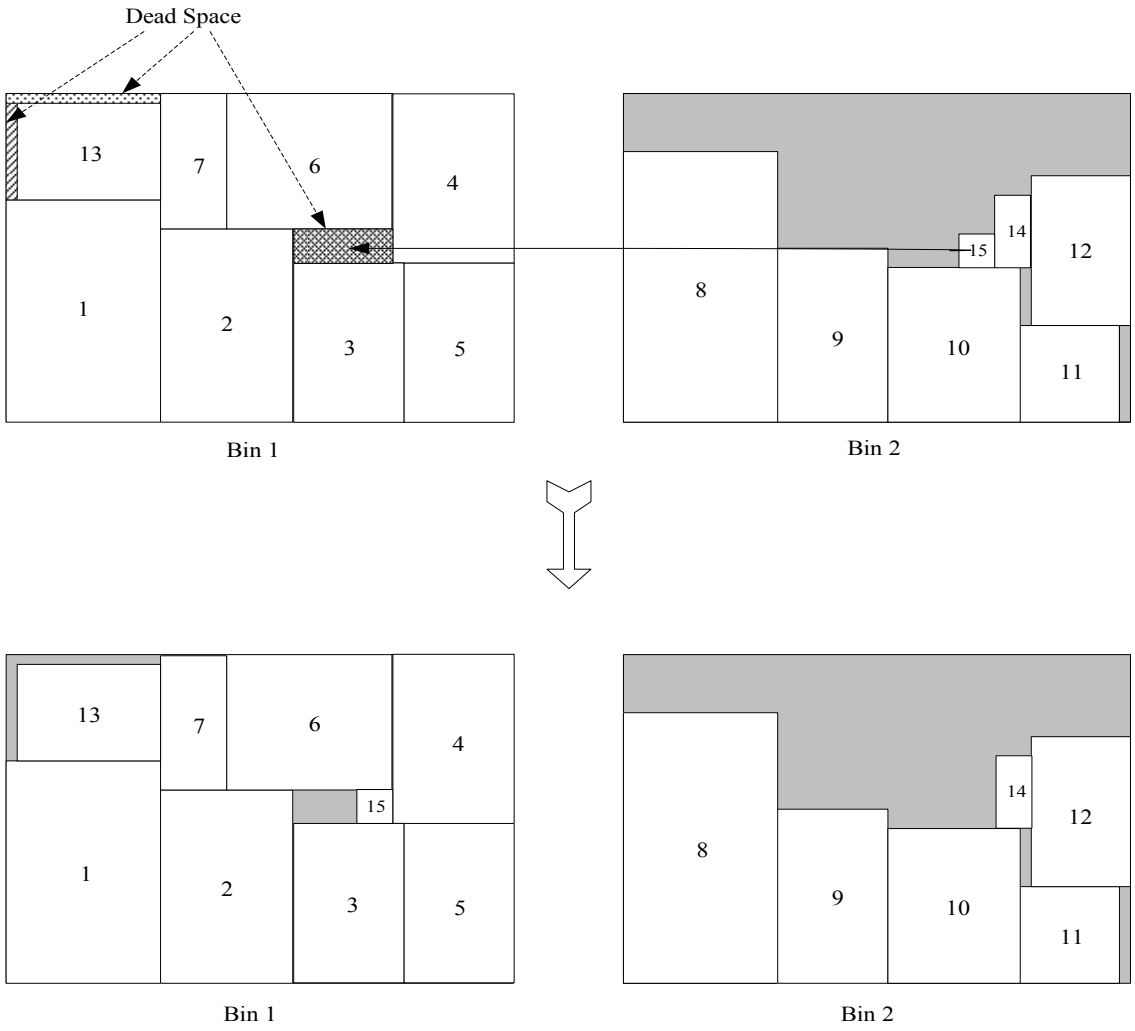


Figure 4.3 Moving an item from bin 2 to bin 1 to occupy a dead space.

Step 0. Initialization.
Parameters are K (scaling parameter), *nloop1*, *nloop2*, T (temperature), and *Tred*
Set $K = 0.05$, *nloop1* = 20, *nloop2* = 50, $T = 1$, and *Tred* = 0.95
Inputs are height and width of each item, height and width of a bin, and lower bound (*LB*)
Δf is defined as the change in objective function value

Step 1. Optimization runs
for $k = 1$: *nloop1* **do**
Construct an initial solution using the 2BP|O|G algorithm, exit k loop if *LB* is reached
Set $T := 1$
for $j = 1$: *nloop2* **do**
Compute weight of bin $i := w_i^T$ and weighted area of item j for all j
Do for all pairs of bins{
Perform Swap (1,0)
Evaluate feasibility and Δf of alternate directions packing
if feasible and $\Delta f \geq 0$
Swap the item
Exit Swap (1,0) and,
Exit k loop and j loop if *LB* is reached
else restore the original solution

Perform Swap (1,1)
Evaluate feasibility and Δf of alternate directions packing
if feasible and $\Delta f \geq 0$
Swap the items
Exit Swap (1,0) and,
Exit k loop and j loop if *LB* is reached
else restore the original solution

Perform Swap (1,2)
Evaluate feasibility and Δf of alternate directions packing
if feasible and $\Delta f \geq 0$
Swap the items
Exit Swap (1,0) and,
Exit k loop and j loop if *LB* is reached
else restore the original solution

}
$T := T \times Tred$
end

Compute the sizes and coordinates of the dead spaces in all bins
Do for all pairs of bins.{
Perform Swap (1,0)
Evaluate feasibility and allow item rotations for 2BP|R|F
if feasible and $\Delta f \geq 0$
Move the item
Exit k loop and j loop if *LB* is reached
}
end

Step 2. Outputs are the number of bins and the final distribution of items

Table 4.1 Weight annealing algorithm for 2BP|O|F and 2BP|R|F.

4.3 Weight Annealing Algorithm for 2BP with Non-Oriented items and Free Cuts (WA2BPF)

In this section, we discuss the application of weight annealing to the two-dimensional bin packing problem where the items are non-oriented and free cutting applies. We point out that the solution generated by our algorithm to 2BP|O|F is a feasible solution to 2BP|R|F. To handle non-oriented items, we modify the post-optimization process to allow rotations of items through 90° to fill up dead spaces in a bin. In Figure 4.4, we illustrate this modification in an example that is the same as the example in Figure 4.3. In Figure 4.3, due to the size and orientation constraints, only item 15 can be moved from bin 2 to bin 1. However, in Figure 4.4, we see that once the orientation restriction is removed, item 14 which is larger than item 15, can fit into a dead space in bin 1. This results in a better packing that has a larger objective function value, i.e., $(A_1^2 + A_2^2)$, than the objective function value of the example in Figure 4.3.

4.4 Computational experiments and results

4.4.1 Results for 2BP|O|F

We coded WA2BPF in C/C++ and solved test problems on a 3 GHz Pentium 4 computer with 256 MB of RAM. There are several papers in the literature containing the published results for 2BP|O|F. However, it is not a straightforward task to compare results, given the way they have been reported in the literature.

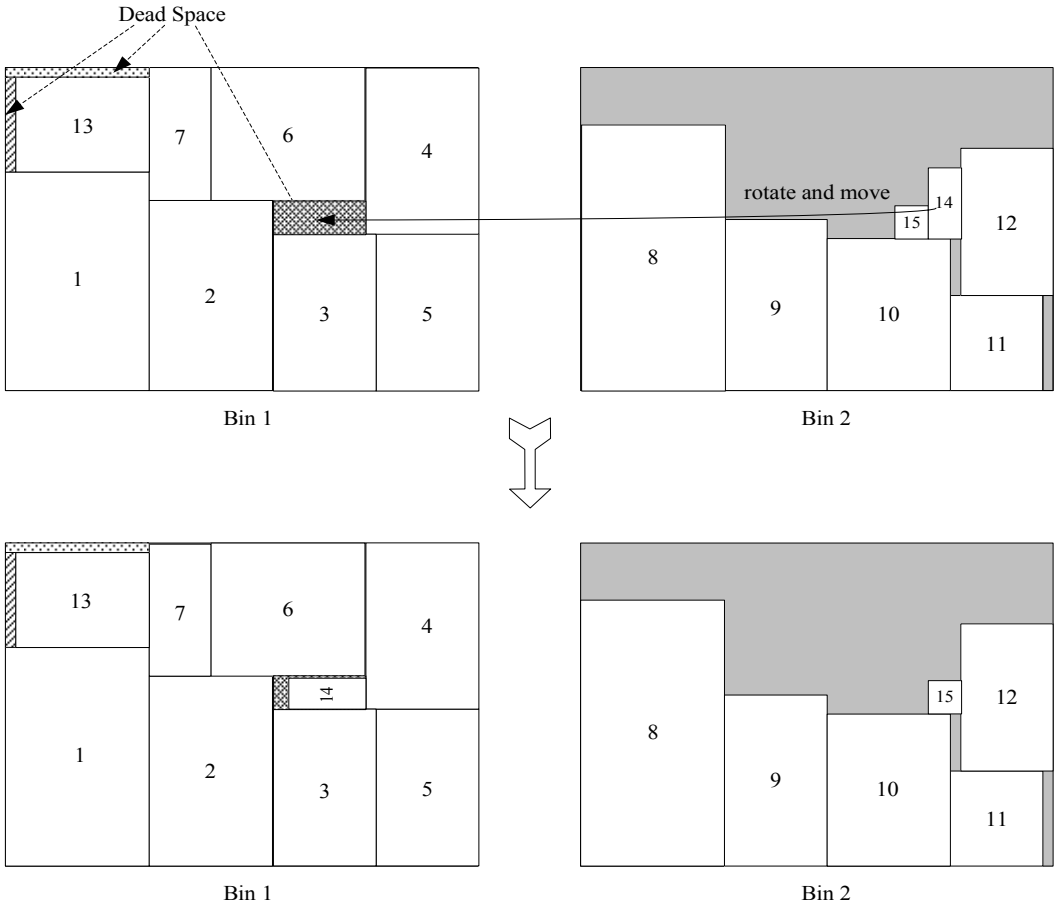


Figure 4.4 Rotating an item through 90° and moving it to occupy dead space in another bin.

Lodi, Martello, and Vigo (1999b) reported the *average ratio*s for tabu search (TS solution value/lower bound) for 10 problem instances computed on the 10 classes of problems, but did not provide the lower bounds they used. Furthermore, these ratios are different from the values reported in Lodi, Martello, and Vigo (1999a).

Faroe, Pisinger, and Zachariasen (2003) reported the *number of bins used* by GLS for the 10 classes, but no running times. They also gave the number of bins used by TS from Lodi, Martello, and Vigo (1999b) (although this paper reported only average ratios). Faroe, Pisinger, and Zachariasen also gave the lower bounds they say were reported in Lodi, Martello, and Vigo (1999a) (although this paper provided no such bounds).

Monaci and Toth (2006) thanked various researchers for providing "…results, for each 2DBP instance, of their algorithms." Monaci and Toth reported the number of bins used by TS for the 10 problem classes. These results do not agree with the TS results given in Faroe, Pisinger, and Zachariasen (2003) (the TS results in Monaci and Toth are slightly better; perhaps they were updated through private communication with Lodi, Martello, and Vigo; however, the computation times for TS reported by Monaci and Toth are *exactly* the same as the times reported in Lodi, Martello, and Vigo (1999b)). Monaci and Toth (2006) provided lower bounds for each problem.

In Table 4.3, we show results from the literature for five algorithms and results for WA2BPF for the 10 classes of 2BP|O|F problems. In order to bring some consistency to our comparison of results, here are the bounds and algorithms given in Table 4.2.

L_{LMV}	Lower bounds based on Lodi, Martello, and Vigo (1999a) that are given in Faroe, Pisinger, and Zachariasen (2003).
LB*	Lower bounds that are given in Monaci and Toth (2006).
TS	Tabu search results obtained by Lodi, Martello, and Vigo (1999b) that are given in Faroe, Pisinger, and Zachariasen (2003).
GLS	Guided local search results that are given in Faroe, Pisinger, and Zachariasen (2003).
EA	Exact algorithm results that are given in Monaci and Toth (2006). Monaci and Toth "…ran the corresponding code [of Martello and Vigo (2001)] on [their] machine." .
HBP(TL)	Constructive algorithm results that are given in Monaci and Toth (2006). This is Monaci and Toth's implementation of the the algorithm of Boschetti and Mingozzi (2003) with a time limit for computation.
SCH	Set covering heuristic results that are given in Monaci and Toth (2006).

				TS		GLS	EA	
Class	*n*	L_{LMV}	LB*	Bins	Time(s)	Bins	Bins	Time(s)
I	20	67	71	71	24.00	71	71	0.01
	40	128	134	136	36.11	134	134	4.62
	60	193	197	201	48.93	201	201	21.01
	80	269	274	282	48.17	275	275	15.01
	100	314	317	327	60.81	321	322	24.07
II	20	10	10	10	0.01	10	10	0.00
	40	19	19	21	0.01	19	20	3.00
	60	25	25	28	0.09	25	27	6.00
	80	31	31	33	12.00	32	34	9.00
	100	39	39	40	6.00	39	40	3.00
III	20	46	51	55	54.00	51	51	0.01
	40	88	92	98	54.02	95	95	12.01
	60	133	136	140	45.67	140	140	15.04
	80	184	187	199	54.31	193	195	24.01
	100	217	221	237	60.10	229	228	27.70
IV	20	10	10	10	0.01	10	10	0.00
	40	19	19	19	0.01	19	20	3.00
	60	23	23	26	0.14	25	27	12.00
	80	30	30	33	18.00	33	33	9.00
	100	37	37	38	6.00	39	40	9.00
V	20	60	65	67	36.02	65	65	0.01
	40	114	119	119	27.07	119	119	5.39
	60	172	179	182	56.77	181	180	15.19
	80	236	241	250	56.18	250	249	27.00
	100	273	279	295	60.34	288	286	27.01

TS	Silicon Graphics INDY R10000sc (195 MHz)
GLS	Digital 500au (500 MHz) with a 30 second time limit
EA, HBP(TL), SCH	Digital Alpha (533 MHz) with a 30 second time limit
WA2BPF	Pentium 4 (3 GHz)

Table 4.2 Number of bins and running times for six algorithms that solve 10 classes of 2BP|O|F problems.

				TS		GLS	EA	
Class	*n*	L_{LMV}	LB*	Bins	Time(s)	Bins	Bins	Time(s)
VI	20	10	10	10	0.01	10	10	0.00
	40	15	15	21	0.03	18	19	12.00
	60	21	21	22	0.04	22	22	3.01
	80	30	30	30	0.01	30	30	0.01
	100	32	32	34	12.00	34	35	9.01
VII	20	53	55	55	12.02	55	55	0.06
	40	108	109	114	37.01	113	111	11.58
	60	155	156	163	36.44	161	162	18.00
	80	223	224	232	54.52	233	234	30.00
	100	268	269	276	47.43	276	276	21.00
VIII	20	55	58	58	18.04	58	58	0.00
	40	111	112	114	18.72	114	113	6.00
	60	159	159	162	20.99	163	164	15.00
	80	222	223	226	37.95	228	226	12.01
	100	273	274	284	52.66	282	281	21.00
IX	20	143	143	143	0.01	143	143	0.00
	40	274	278	277	24.05	278	278	0.01
	60	433	437	437	24.26	437	437	0.12
	80	569	577	575	54.31	577	577	3.51
	100	689	695	696	34.11	695	695	12.80
X	20	40	42	44	12.00	42	42	0.05
	40	71	74	75	25.18	74	74	2.28
	60	97	98	104	42.13	102	103	16.86
	80	123	123	130	47.30	130	132	28.29
	100	153	153	165	60.10	163	164	30.00
Total		7064	7173	7364	1436.09	7302	7313	524.69

Table 4.2. (continued)

				HBP(TL)		SCH		WA2BPF	
Class	*n*	L_{LMV}	LB*	Bins	Time(s)	Bins	Time(s)	Bins	Time(s)
I	20	67	71	71	3.08	71	0.07	71	0.21
	40	128	134	134	9.68	134	3.93	134	0.06
	60	193	197	201	12.12	200	2.50	200	0.67
	80	269	274	275	3.08	275	2.50	275	3.07
	100	314	317	319	6.33	317	3.38	317	9.21
II	20	10	10	10	0.06	10	0.06	10	0.05
	40	19	19	19	0.46	19	0.31	20	0.04
	60	25	25	25	0.07	25	0.07	25	0.43
	80	31	31	31	0.48	31	0.07	31	13.89
	100	39	39	39	0.26	39	0.37	39	8.70
III	20	46	51	51	6.28	51	0.07	53	0.04
	40	88	92	94	6.48	94	1.10	94	2.15
	60	133	136	140	12.14	139	2.66	139	0.16
	80	184	187	190	9.93	190	6.09	189	3.16
	100	217	221	225	12.59	223	5.10	224	7.52
IV	20	10	10	10	0.07	10	0.06	10	0.05
	40	19	19	19	0.08	19	0.07	19	0.14
	60	23	23	25	6.13	25	1.86	25	0.05
	80	30	30	32	7.10	32	3.15	31	12.19
	100	37	37	38	4.04	38	2.32	38	0.36
V	20	60	65	65	0.10	65	0.06	65	0.10
	40	114	119	119	9.31	119	1.21	119	0.17
	60	172	179	180	8.21	180	1.05	180	1.49
	80	236	241	248	18.80	247	8.82	247	2.66
	100	273	279	286	18.50	283	5.30	283	3.50

Table 4.2 (continued)

				HBP(TL)		SCH		WA2BPF	
Class	*n*	L_{LMV}	LB*	Bins	Time(s)	Bins	Time(s)	Bins	Time(s)
VI	20	10	10	10	0.07	10	0.07	10	0.04
	40	15	15	17	6.91	17	2.81	19	0.07
	60	21	21	21	0.16	21	0.35	22	0.05
	80	30	30	30	0.24	30	0.23	30	0.06
	100	32	32	34	6.39	34	2.75	33	21.00
VII	20	53	55	55	6.09	55	0.12	55	0.05
	40	108	109	112	10.25	111	1.41	111	2.12
	60	155	156	160	13.09	158	3.50	159	6.79
	80	223	224	232	24.24	232	15.71	232	0.27
	100	268	269	273	13.01	271	9.86	271	1.92
VIII	20	55	58	58	0.07	58	0.06	58	0.05
	40	111	112	113	3.47	113	0.49	113	0.21
	60	159	159	162	9.33	162	3.36	162	0.16
	80	222	223	225	6.36	224	3.90	224	0.33
	100	273	274	279	15.48	279	13.30	277	0.06
IX	20	143	143	143	0.06	143	0.06	143	0.19
	40	274	278	278	0.07	278	0.06	279	0.04
	60	433	437	437	0.08	437	0.07	438	0.12
	80	569	577	577	0.08	577	0.08	577	0.16
	100	689	695	695	0.11	695	0.11	695	0.23
X	20	40	42	42	4.78	42	0.12	43	0.29
	40	71	74	74	6.11	74	0.11	74	0.21
	60	97	98	102	16.13	101	4.20	102	0.16
	80	123	123	130	21.26	130	14.48	129	5.42
	100	153	153	160	26.63	160	19.07	159	9.26
Total		7064	7173	7265	345.85	7248	148.46	7253	119.33

Table 4.2 (continued)

In Table 4.2, for all 500 test instances, we see that SCH used 7,248 bins, closely followed by WA2BPF with 7,253 bins and HBP with 7,265 bins. GLS, EA, and TS needed more than 7,300 bins. The total number of bins used by SCH and WA2BPF are about 2.6% and 1.1% above the total number of bins for the lower bounds L_{LMV} and LB*, respectively. Although the computers are different, WA2BPF and SC are very fast, taking 119.33 seconds and 148.46 seconds, respectively, to solve all 500 test instances. We point out that, in Class IX with $n = 40$, the TS solution of 277 reported by Faroe, Pisinger, and and Zachariasen (2003) is less than the lower bound of 278 given by Monaci and Toth (2006).

4.4.2 Results for 2BP|R|F

In Table 4.3, we show the results generated by WA2BPF on the 2BP|R|F problems (items can be rotated). WA2BPF used 7,222 bins and needed 66.66 seconds for all 500 test instances. Since the items can be rotated, 31 fewer bins were used when compared to the oriented solutions produced by WA2BPF that are given in Table 4.2.

Lodi, Martello, and Vigo (1999b) published results produced by TS for the 10 classes of 2BP|R|F problems. They reported average ratio results. The lower bounds from an unpublished paper by Dell'Amico, Martello, and Vigo (1999) that were used by Lodi, Martello, and Vigo are not currently available, thereby making average ratio comparisons between WA2BPF and TS impossible.

		WA2BPF				WA2BPF	
Class	*n*	Bins	Time(s)	Class	*n*	Bins	Time(s)
I	20	71	0.21	VI	20	10	0.04
	40	134	0.06		40	19	0.07
	60	197	3.25		60	22	0.05
	80	274	0.25		80	30	0.06
	100	317	9.21		100	33	6.30
II	20	10	0.05	VII	20	55	0.05
	40	20	0.04		40	111	1.26
	60	25	0.11		60	156	4.31
	80	31	1.70		80	225	1.34
	100	39	1.50		100	269	1.87
III	20	52	0.04	VIII	20	58	0.05
	40	94	2.23		40	112	0.21
	60	138	0.16		60	159	0.17
	80	189	1.08		80	223	0.13
	100	224	3.12		100	274	0.57
IV	20	10	0.05	IX	20	143	0.10
	40	19	0.14		40	279	0.04
	60	25	0.05		60	438	0.12
	80	31	1.42		80	577	0.16
	100	38	0.36		100	695	0.23
V	20	65	0.10	X	20	43	0.29
	40	119	0.17		40	74	0.21
	60	180	0.66		60	101	5.96
	80	244	4.65		80	128	6.85
	100	283	2.17		100	159	3.44
				Total		7222	66.66

Table 4.3 Number of bins and running times for WA2BPF on 10 classes of 2BP|R|F problems.

4.5 Conclusions

We developed an algorithm based on weight annealing that solved two variants of 2BP with free cuts (2BP|O|F and 2BP|R|F). With respect to oriented items and free cutting, our weight annealing algorithm generated results that were comparable in terms of accuracy and computational speed to the best results found in the literature.

Chapter 5

The Maximum Cardinality Bin Packing Problem

5.1 Introduction

In the maximum cardinality bin packing problem (MCBP), we are given n items with sizes $t_i,\ i \in N = \{1,...,n\}$, and m bins of identical capacity c. The objective is to assign a maximum number of items to the fixed number of bins without violating the capacity constraint. The problem formulation is given by

$$\text{maximize} \quad \sum_{i=1}^{n}\sum_{j=1}^{m} x_{ij} \tag{5.1}$$

subject to

$$\sum_{i=1}^{n} t_i x_{ij} \le c \qquad j \in \{1,...,m\}$$

$$\sum_{i=1}^{m} x_{ij} \le 1 \qquad i \in \{1,...,n\}$$

$$x_{ij} = 0 \text{ or } 1 \qquad i \in \{1,...,n\}, \quad j \in \{1,...,m\}$$

where $x_{ij} = 1$ if item i is assigned to bin j and 0 otherwise.

The MCBP is NP-hard (Labbé, Laporte, and Martello 2003). Applications of the MCBP include the following.

- Computing. We need to assign variable-length records to storage, and the objective is to maximize the number of records stored in fast memory so as to ensure a minimum access time to the records given a fixed amount storage space (Labbé, Laporte, and Martello 2003).

- Management of real time multi-processors. The objective is to maximize the number of completed tasks with varying job durations before a given deadline (Coffman, Leung, and Ting 1978).
- Computer design. Ferreira, Martin, and Weismantel (1996) applied the MCBP with side constraints to designing processors for mainframe computers and, designing the layout of electronic circuits.

Heuristics for solving the MCBP were developed by Coffman, Leung, and Ting (1978, 1979). Bruno and Downey (1985) and Coffman, Leung, and Ting (1978) provided probabilistic lower bounds for the MCBP. Kellerer (1999) considered the MCBP as a special case of the multiple knapsack problem where all items have the same profit = 1 and all knapsacks (or bins) have the same capacity *c*. This problem was solved with a polynomial approximation scheme for the multiple knapsack problem. Labbé, Laporte, and Martello (2003) developed upper bounds, and embedded them into an enumeration algorithm for solving the MCBP. Peeters and Degraeve (2006) introduced combinatorial upper bounds and heuristics into a branch-and-price framework.

In this chapter, we propose a weight annealing algorithm for solving the MCBP. We evaluate the performance of our weight annealing algorithm against several algorithms using benchmark problems developed by Labbé, Laporte, and Martello (2003) and Peeters and Degraeve (2006).

5.2 Upper Bounds on the Number of Items

Our algorithm uses the upper bounds that were developed by Labbé, Laporte, and Martello (2003) The three a prior upper bounds ($\overline{U}_0, \overline{U}_1, \overline{U}_2$) are presented below. Without the loss of generality, problem data are integer and $1 \le t_1 \le t_2 \le \ldots \le t_n \le c$.

The first upper bound for the optimal solution z^* of a given instance is given by

$$\overline{U}_0 = \max_{1 \le k \le n} \left\{ k : \sum_{i=1}^{k} t_i \le mc \right\}. \tag{5.2}$$

Since the optimal solution is obtained by selecting the first z^* smallest items, all items with sizes t_i for which $i > \overline{U}_0$ can be disregarded.

To illustrate the computation of $\overline{U}_0$, we consider a MCBP instance with $n = 6$, $t_i =$ (5, 6, 6, 6, 7, 8), $m = 3$, and $c = 10$. Based on (5.2), we have $\overline{U}_0 = 5$. Note that the optimal solution has z^* equals to 3, and the first three smallest items {5, 6, 6} packed into three bins.

The second upper bound $\overline{U}_1$ is formulated as follows. Let $Q(j)$ be the upper bound on the number of items that can be assigned to j bins. We have

$$Q(j) = \max \left\{ k : j \le k \le n, \sum_{i=1}^{k} t_i \le jc \right\} \quad \text{for } j = 1,\ldots,m. \tag{5.3}$$

An upper bound on z^* bins is given by

$$U_1(j) = Q(j) + \lfloor Q(j)/j \rfloor (m - j) \tag{5.4}$$

since $\lfloor Q(j)/j \rfloor$ is an upper bound on the number of items that can be assigned to each of the remaining (m-j) bins. By taking the minimum over all j, we obtain the upper bound

$$\overline{U}_1 = \min_{j=1,\ldots,m} U_1(j). \tag{5.5}$$

Note that $\overline{U}_1$ dominates $\overline{U}_0$. To illustrate the computation of $\overline{U}_1$, we use the same MCBP instance with $n = 6$, $t_i = (5, 6, 6, 6, 7, 8)$, $m = 3$, and $c = 10$. Based on (5.4), we have $U_1(1) = 3$, $U_1(2) = 4$, and $U_1(3) = 5$. By taking the minimum over these three values, we have $\overline{U}_1 = 3$.

The third upper bound $\overline{U}_1$ is formulated as follows. Let i be the smallest item in an instance with m bins. Then $m\lfloor c/t_i \rfloor$ is an upper bound on the number of items that can be assigned to m bins because $\lfloor c/t_i \rfloor$ is an upper bound on the number of items that can be packed into one bin. A valid upper bound is given by

$$U_2(i) = (i-1) + m\lfloor c/t_i \rfloor. \tag{5.6}$$

If i is not the smallest item, then an optimal solution will contain all items $j < i$, and by taking minimum over all i, we obtain a valid upper bound

$$\overline{U_2} = \min_{j=1,\ldots,n} U_2(j). \tag{5.7}$$

To illustrate the computation of $\overline{U}_2$, we use the same MCBP instance with $n = 6$, $t_i = (5, 6, 6, 6, 7, 8)$, $m = 3$, and $c = 10$. Based on (5.6), we have $U_2(1) = 6$, $U_2(2) = 4$, $U_2(3) = 5$, $U_2(4) = 6$, $U_2(5) = 7$, and $U_2(6) = 8$, . By taking the minimum over these six values, we have $\overline{U}_2 = 4$.

It follows that the best a priori upper bound $U^* = \min\{\overline{U}_0, \overline{U}_1, \overline{U}_2\}$. Since the optimal solution is obtained by selecting the first z^* smallest items, all items with sizes t_i for which $i > U^*$ can be disregarded. In our example, it turns out that $U^* = 3$, which is also our optimal solution, and the items picked are the first three smallest items $\{5, 6, 6\}$.

5.3 Lower Bounds on the Number of Bins

To improve the upper bound U^*, we use a procedure that incorporates the L_2 and L_3 bounds by Martello and Toth (1990).

5.3.1 L_1 bound

Given an 1BP instance I, the L_1 bound (also known as the simple lower bound) on the minimum number of bins required is given by

$$L_1 = \left\lceil \sum_{i=1}^{n} t_i / c \right\rceil. \tag{5.8}$$

To illustrate the computation of L_1, we use a 1BP instance with $n = 9$, $t_i = (70, 60, 50, 33, 33, 33, 11, 7, 3)$, and $c = 100$. In this case $L_1 = \lceil 300/100 \rceil = 3$, or the lower bound on the minimum number of bins required is three. Note that, the optimal solution requires four bins for item sets {70, 11,7,3}, {60, 33}, {50, 33}, and {33}.

L_1 can be expected to have good average behavior for those instances with items of sufficiently small sizes relatively to the bin capacity since in such cases the evaluation is not greatly affected by the relaxation of integrality constraints. For problems of larger items sizes, Martello and Toth (1990) proposed a better lower bound, L_2.

5.3.2 L_2 bound

Given an 1BP instance I for one dimensional bin packing, the lower bound L_2 on the optimal number of bins $z(I)$ can be computed as follows.

Given any integer $\alpha, 0 \le \alpha \le c/2$, let

$$J_1 = \{ j \in N : t_j > c - \alpha \},$$
$$J_2 = \{ j \in N : c - \alpha \ge t_j > c/2 \},$$
$$J_3 = \{ j \in N : c/2 \ge t_j \ge \alpha \},$$

then

$$L(\alpha) = |J_1| + |J_2| + \max\left(0, \left\lceil \frac{\sum_{j \in J_3} t_j - \left(|J_2| c - \sum_{j \in J_2} t_j \right)}{c} \right\rceil \right) \tag{5.9}$$

is a lower bound of z(I).

Since each item in $J_1 \cup J_2$ needs a separate bin, so $|J_1| + |J_2|$ bins are required in any feasible solution. Because of the capacity constraint, no items in J_3 can be assigned to a bin containing an item of J_1. The $|J_2|$ bins has a total residual capacity $\bar{c} = |J_2| c - \sum_{j \in J_2} t_j$.

In the best case, $\bar{c}$ can be completely filled by items in J_3; if not, the residual total weight $\bar{t} = \sum_{j \in J_3} t_j - \bar{t}$ will require at most $\lceil \bar{t} / c \rceil$ additional bins.

L_2 is obtained by taking the maximum over α, or

$L_2 = \max\{ L(\alpha) : 0 \le \alpha \le c/2, \alpha \text{ integer} \}$ is a lower bound of z(I). (5.10)

To illustrate the computation of L_2, we use the same 1BP instance with n = 9, t_i = (70, 60, 50, 33, 33, 33, 11, 7, 3), and c = 100. In this case, we have

$$L(50) = 2 + 0 + \max\left(0, \lceil (50 - 0)/100 \rceil\right) = 3;$$
$$L(33) = 1 + 1 + \max\left(0, \lceil (149 - 40)/100 \rceil\right) = 4.$$

Evaluating $L(\alpha)$ for the remaining values of α, and taking the maximum value as stated in (5.10), we obtain $L_2 = 4$, which is greater than $L_1 = 3$. In this instance, it turns out that L_2 is also the optimal number of bins used.

5.3.3 Reduction Techniques

In our algorithm, we use the reduction procedure MTRP that was developed by Martello and Toth (1990) to determine L_3, which nominates L_2. A feasible set of items is any subset $F \subseteq N$ such that $\sum_{j \in F} t_j \leq c$. Given two feasible sets F_1 and F_2, we say that F_1 dominates F_2 if and only if the number of bins in some optimal solution can be obtained by imposing for a bin, say i^*, $x_{i^* j} = 1$ if $j \in F_1$, and $x_{i^* j} = 0$ if $j \notin F_1$, is no greater than that obtained by imposing $x_{i^* j} = 1$ if $j \in F_2$, and $x_{i^* j} = 0$ if $j \notin F_2$. A possible way to check such conditions is to determine if there exists a partition $P_1,\ldots, P_l$ of F_2 and a subset $\{j_1,\ldots, j_l\}$ of F_1 such that $t_{j_h} \geq \sum_{k \in P_h} t_k$ for $h = 1,...,l$.

If a feasible set F dominates all others, then the items of F can be removed from N and assigned to a bin. In this way, MTRP iteratively reduces the size of an instance of bin packing problem until no such F exists or all items have been assigned. Clearly, checking all feasible sets is impractical, and MTRP therefore limits the search to feasible sets of cardinality of at most three (we will illustrate the steps with an example later).

5.3.4 L_3 Bound

MTRP can be used to compute a new lower bound L_3 which is stronger than L_2. Let I be the original instance, z_1^r be the number of bins reduced after the first application

of MTRP to *I,* and $I(z_1^r)$ be the corresponding residual instance. If $I(z_1^r)$ is relaxed by removing its smallest item, we can obtain a lower bound by applying L_2 to $I(z_1^r)$ yielding $L_1^{'} = z_1^r + L_2(I(z_1^r)) \geq L_2(I)$. Iterating the process until the residual instance is empty, we obtain for iteration *k*, a lower bound $L_k^{'} = z_1^r + z_2^r + ... + z_k^r + L_2(I(z_k^r))$. Then

$$L_3 = \max\left\{L_1^{'} + L_2^{'} + ... + L_{k_{\max}}^{'}\right\} \text{ is a valid lower bound for } I \qquad (5.11)$$

where k_{max} is the number of iterations needed to have the residual instance becoming empty.

Consider the instance of 1BP as stated in Martello and Toth (1990)

n = 14,

t_i = (99, 94, 79, 64, 50, 46, 43, 37, 32, 19,18, 7, 6, 3),

c = 100.

In the first iteration of MTRP, we have F = {99} dominating all other subsets. Removing the item {99} from *N,* we are left with the residual instance {94, 79, 64, 50, 46, 43, 37, 32, 19, 18, 7, 6, 3}.

In the second iteration of MTRP, we have F = {94, 6} dominating all other subsets in the current residual instance. Removing items {94, 6}, the residual instance becomes {79, 64, 50, 46, 43, 37, 32, 19, 18}. Computing the L_2 bound for the residual instance, we have L_2 = 4, and so L_3 = 6. Iterating the process until the residual instance is empty, we will end up with L_3 = 7.

5.4 Weight Annealing Algorithm for the MCBP (WAMCBP)

The MCBP is a variant of the one-dimensional bin packing problem, and can be solved with the same weight annealing approach that we developed for solving 1BP. There are four steps in our weight annealing algorithm for solving the MCBP.

Step 1. Compute the tightest a prior upper bound $U^* = \min\left\{\overline{U}_0, \overline{U}_1, \overline{U}_2\right\}$.

Step 2. Improve the upper bound U^* with L_3 bound.

Step 3. Find feasible packing solution with WA1BP.

Step 4. Output results.

In Table 1, we present out weight annealing algorithm for the maximum cardinality bin packing problem. We denote our algorithm by WAMCBP.

The number of items (n), the ordered list of item sizes, bin capacity (c) and the number of bins (m) are inputs. For the ordered list, the data are integers and $1 \le t_1 \le t_2 \le \ldots \le t_n \le c$, where t_i is the size of item i.

We begin by computing the upper bounds $\overline{U}_0$, $\overline{U}_1$ and $\overline{U}_2$. Taking the minimum of the computed upper bounds, we have an a priori bound $U^* = \min\left\{\overline{U}_0, \overline{U}_1, \overline{U}_2\right\}$. Since the optimal solution of any instance is obtained by selecting the first z^* smallest items, we update the ordered list by removing item i with size t_i for which $i > U^*$.

To improve the upper bound, we compute L_3 by applying MTRP. If L_3 is greater than m, it will not be feasible to pack the items on the ordered list into the given number of m bins, and we can thus reduce U^* by 1. Accordingly, we update the ordered list by removing item i with size t_i for which $i > U^*$. We iterate this step until $L_3 = m$.

Step 0. Initialization
Parameters are *K, nloop1, nloop2, T, Tred*
Set $K = 0.05$, *nloop1* = 20, *nloop2* = 50, $T = 1$, *Tred* = 0.95
Inputs are number of items (n), the item size ordered list , bin capacity (c) and number of bins (m)

Step 1. Compute a priori upper bound $U^* = \min\left\{\overline{U}_0, \overline{U}_1, \overline{U}_2\right\}$

Step 2. $n = U^*$
Remove item $i > U^*$ from the ordered list

Step 3. Improve the upper bound
While ($L_3 > m$) do{
$U^* = U^* - 1$
Remove item $i > U^*$ from the ordered list
Compute L_3 }

Step 4. For $j = 1$ to *nloop1*
Step 4.1 Construct initial solution with the ordered list with modified first-fit decreasing algorithm
Step 4.2 Improve the current solution
Set T=1
Compute residual capacity r_i of bin i
For k =1 to *nloop2*
Compute weights: $w_i^T = (1\text{-}K\, r_i)^T$
Do for all pairs of bins{
Perform Swap (1,0)
Evaluate feasibility and Δf
If $\Delta f \geq 0$
Move the item
Exit Swap(1,0) and,
Exit j loop and k loop if m is reached
Perform Swap (1,1)
Evaluate feasibility and Δf
if $\Delta f \geq 0$
Swap the items
Exit Swap(1,1) and,
Exit j loop and k loop if m is reached
Perform Swap (1,2)
Evaluate feasibility and Δf
if $\Delta f \geq 0$
Swap the items
Exit Swap(1,2) and,
Exit j loop and k loop if m is reached
Perform Swap (2,2)
Evaluate feasibility and Δf
if $\Delta f \geq 0$
Swap the items
Exit Swap(2,2) and,
Exit j loop and k loop if m is reached }
$T := T \times T_{red}$
End of k loop
End of j loop

Step 5. Outputs are the number and final distribution of items

Table 5.1 Weight annealing algorithm for MCBP.

Next we solve the one-dimensional bin packing problem with the current ordered list. We start with an initial solution generated by the first-fit decreasing procedure that we have modified in the following way. We select an item for packing with probability 0.5. In other words, we start with the first item on the ordered list and, based on a coin toss, we pack it into a bin if it is selected, or leave it on the ordered list if it is not selected. We continue down the ordered list until an item is selected for packing. We then pack the second item in the same manner, and so on, until we reach the bottom of the list. For each bin i in the FFD solution, we compute the bin load (l_i) and the residual capacity (r_i).

To improve a solution, we carry out swapping operations with weight annealing. A temperature parameter (T) controls the amount by which a single weight can be varied. At the start, a high temperature ($T = 1$) allows for more downhill moves. As the temperature is gradually cooled (the temperature is reduced at the end of every iteration, that is, $T \times 0.95$), the amount of item distortion decreases and the problem space looks more like the original problem space.

We compute a weight for each bin according to $w_i^T = (1 + Kr_i)^T$ and then apply the weight to each item in the bin. Without weight annealing, it is not possible to escape from a poor local solution. In Figure 2.9 of Chapter 2, we show an example of an uphill move in the transformed space that associated with a downhill move in the original space. The swapping process begins by comparing the items in the first bin with the items in the second bin, and so on, sequentially down to the last bin in the initial solution and is repeated for every possible pair of bins.

For a current pair of bins (α, β), the swapping of items by Swap (1,0) is carried out as follows. The algorithm evaluates whether the first item (item *i*) in bin α can be moved to bin β without violating the capacity constraint of bin β in the original space. In other words, does bin β have enough original residual capacity to accommodate the original size of item *i*? If the answer is yes (the move is feasible), the change in objective function value of the move $\Delta f \geq 0$ in the transformed space is evaluated. If $\Delta f \geq 0$, item *i* is moved from bin α to bin β. After this move, if bin α is empty and the total number of utilized bins reaches the specified number of bins (*m*), the algorithm stops and outputs the final results. If bin α is still partially filled, or the lower bound has not been reached, the algorithm exits Swap (1,0) and proceeds to Swap (1,1). If the move of the first item is infeasible or $\Delta f < 0$, the second item in bin α is evaluated and so on, until a feasible move with $\Delta f \geq 0$ is found or all items in bin α have been considered and no feasible move with $\Delta f \geq 0$ has been found . The algorithm then performs Swap (1,1), followed by Swap (1,2), and Swap (2,2). In each of the swapping schemes, we always take the first feasible move with $\Delta f \geq 0$ that we find.

We point out that the improvement step (Step 4.2) is carried out 50 times (*nloop2* = 50) starting with $T = 1$, followed by $T = 1 \times 0.95 = 0.95$, $T = 0.95 \times 0.95 = 0.9025$, etc. At the end of Step 4, if the total number of utilized bins has not reached *m*, we repeat Step 4 with another initial solution. We exit the program as soon as the required number of bins reaches *m* or after 20 runs (*nloop1* = 20).

5.5 Computational Experiments

We coded WAMCBP in C/C++ and used a 3 GHz Pentium 4 computer with 256 MB of RAM. We compare the results produced by our weight annealing algorithm WAMCBP to the results produced by the algorithm of Labbé, Laporte, and Martello (2003) (denoted LA), and the branch-and-price algorithm of Peeters and Degraeve (2006) (denoted BP). The procedure LA was tested on a VAX station 3100/30 which has speed comparable to PC486/33. The procedure BP was tested on a COMPAQ Armada 700 M, 500 MHz Intel Pentium 3 computer. We summarize the computational results in Table 5.2 to Table 5.9. All computation times are given in seconds and, where applicable, followed by the number of instances solved to optimality out of 10 instances. The procedure LA has a limit on the number of branch-and-bound nodes. The BP procedure imposes a time limit of 900 seconds.

5.5.1 Benchmark Problems

We tested our algorithm on the benchmark problems of Labbé, Laporte, and Martello (2003). They randomly generated instances using 180 combinations of three parameters: number of bins (m = 2, 3, 5, 10, 15, 20), capacity (c = 100, 120, 150, 200, 300, 400, 500, 600, 700, 800), and size interval [t_{min} , 99] (t_{min} = 1, 20, 50). For each combination (m, c, t_{min}), they created 10 instances by generating item size t_i in the given size interval according to discrete uniform distribution until the condition $\sum_i t_i > mc$ is met.

Peeters and Degraeve (2006) extended these problems by multiplying the capacity by a factor of 10 and enlarging the size interval to [t_{min} , 999]. Rather than fixing the

number of bins, they fixed the expected number of generated items (denoted here as $E(n')$). However, $E(n')$ is not an input for generating the instances; it is implicitly determined by the number of bins and capacity. Since the item sizes are uniformly distributed on the size interval $[t_{min}, 999]$, the expected item size equals $(t_{min} + 999)/2$. Thus, we have $E(n') = 2cm/(t_{min} + 99)$. Given the number of expected items $\bar{n}$ as an input, the number of bins m must be $\bar{n}(t_{\min} + 999)/2c$. The instances were generated for each of the 270 combinations of three parameters: desired number of items ($\bar{n}$ = 100, 150, 200, 250, 300, 350, 400, 450, 500), capacity (c = 1000, 1200, 1500, 2000, 3000, 4000, 5000, 6000, 7000, 8000), and size interval $[t_{min}, 999]$ (t_{min} = 1, 20, 50).

5.5.2 Results for WAMCBP

In Table 5.2, we show the average number of items (n) generated for each triplet $(t_{min,}\ m,\ c)$ for the instances developed by Labbé, Laporte, and Martello (2003). In Table 5.3, we report the number of instances solved to optimality by LA, BP, and WAMCBP. When the number of instances solved to optimality is less than 10 for WAMCBP, we show in parentheses the maximum absolute error from the optimal solution in terms of number of items. We see that of the 920 instances, BP was able to solve all the instances. WAMCBP found optimal solutions to 917 instances, and LA found optimal solutions to 890 instances. In Table 5.4, we show the running times of the algorithms. We note that LA was tested on a slow machine. WAMCBP was very fast in producing the solutions, with the overall average running time being slightly more than 0.03 second.

		m			
t_{min}	c	5	10	15	20
1	100	11	20	30	41
	120	12	24	36	49
	150	15	31	46	60
	200	20	41	62	85
20	100	8	17	26	34
	120	10	21	30	41
	150	13	25	39	51
	200	16	34	51	70
	300	26	51	79	101
	400	34	70	101	134
	500	42	84	125	167
	600	52	100	149	201
	700	60	116	177	236
	800	67	134	203	267
50	120	8	16	24	32
	150	10	20	30	40
	200	13	27	40	54
	300	20	41	61	82
	400	27	54	81	107
	500	34	68	101	134
	600	41	81	121	160
	700	48	93	140	188
	800	54	107	161	215

Table 5.2. Average value of n over 10 instances for the instances developed by Labbé et al. (2003).

		m											
		5			10			15			20		
t_{min}	c	LA	BP	WA	LA	BP	WA	LA	BP	WA	LA	BP	WA
1	100	10	10	10	10	10	10	10	10	10	10	10	10
	120	10	10	10	10	10	10	10	10	10	10	10	10
	150	10	10	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	10	10	10	10	10	10	10
20	100	10	10	10	10	10	9 (1)	10	10	10	10	10	10
	120	10	10	10	10	10	10	10	10	10	10	10	10
	150	10	10	10	10	10	9 (1)	10	10	9 (1)	10	10	10
	200	10	10	10	10	10	10	10	10	10	10	10	10
	300	10	10	10	10	10	10	10	10	10	10	10	10
	400	10	10	10	10	10	10	10	10	10	10	10	10
	500	10	10	10	10	10	10	10	10	10	10	10	10
	600	10	10	10	10	10	10	10	10	10	10	10	10
	700	10	10	10	10	10	10	10	10	10	10	10	10
	800	10	10	10	10	10	10	10	10	10	10	10	10
50	120	10	10	10	10	10	10	10	10	10	10	10	10
	150	10	10	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	10	7	10	10	8	10	10
	300	10	10	10	9	10	10	7	10	10	6	10	10
	400	10	10	10	10	10	10	7	10	10	5	10	10
	500	10	10	10	10	10	10	10	10	10	5	10	10
	600	10	10	10	10	10	10	10	10	10	9	10	10
	700	10	10	10	10	10	10	10	10	10	7	10	10
	800	10	10	10	10	10	10	10	10	10	10	10	10
	Total	230	230	230	229	230	228	221	230	229	210	230	230

WA is the weight annealing algorithm for the maximum cardinality bin packing problem (WAMCBP)
() The maximum absolute error of WAMCB is shown in the parentheses

Table 5.3 Number of instances solved to optimality by LA, BP and WAMCBP.

		m											
		5			10			15			20		
t_{min}	c	LA	BP	WA	LA	BP	WA	LA	BP	WA	LA	BP	WA
1	100	0.2	0.0	0.0	0.4	0.0	0.0	0.6	0.0	0.0	0.7	0.0	0.3
	120	0.0	0.0	0.0	0.3	0.0	0.0	0.6	0.0	0.0	0.6	0.0	0.1
	150	0.2	0.0	0.0	0.1	0.0	0.0	1.1	0.0	0.0	0.1	0.0	0.0
	200	0.1	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
20	100	0.1	0.0	0.0	0.3	0.0	0.0 (9)	0.3	0.0	0.0	0.6	0.0	0.1
	120	0.1	0.0	0.0	0.4	0.0	0.0	0.6	0.0	0.0	1.0	0.0	0.0
	150	0.2	0.0	0.0	0.4	0.0	0.0 (9)	0.5	0.0	0.0 (9)	3.7	0.0	0.0
	200	0.1	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	2.9	0.0	0.0
	300	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.0
	400	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.1	0.0	0.1
	500	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.1	0.0	0.1
	600	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.0	0.0
	700	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.2	0.1	0.0	0.5
	800	0.1	0.0	0.0	0.0	0.0	0.1	0.1	0.0	0.0	0.2	0.0	0.9
50	120	0.1	0.0	0.0	0.1	0.0	0.0	0.2	0.0	0.0	0.2	0.0	0.0
	150	0.3	0.0	0.0	0.2	0.0	0.0	0.4	0.0	0.0	0.5	0.0	0.0
	200	0.1	0.0	0.0	0.4	0.0	0.0	192.2 (7)	0.0	0.0	177.1 (8)	0.0	0.0
	300	0.1	0.0	0.0	41.7 (9)	0.0	0.0	258.5 (7)	0.0	0.1	268.1 (6)	0.1	0.1
	400	0.0	0.0	0.0	3.5	0.0	0.0	202.0 (7)	0.0	0.0	360.7 (5)	0.1	0.0
	500	0.0	0.0	0.0	0.1	0.0	0.0	0.6	0.0	0.0	133.0 (5)	0.1	0.0
	600	0.0	0.0	0.0	0.4	0.0	0.0	4.8	0.0	0.0	67.1 (9)	0.0	0.1
	700	0.1	0.0	0.0	0.1	0.0	0.0	9.3	0.0	0.1	745.2 (7)	0.0	0.2
	800	0.0	0.0	0.0	1.2	0.0	0.1	185.4	0.0	0.2	141.2	0.0	0.4

WA is the weight annealing algorithm for the maximum cardinality bin packing problem (WAMCBP)

LA VAX station 3100/30
BP 400 MHz Pentium
WA 3 GHz Pentium

Table 5.4 Average running times in seconds for LA, BP, and WAMCBP.

In Table 5.5, we show the number of bins m for each of the 270 combinations of desirable number of items, capacity, and minimum item size. In Table 5.6, we show the results of BP and WAMCBP to the 1350 instances with bin capacity ranging from 1000 to 3000 from the extended problem set developed by Peeters and Degraeve (2006). When the number of instances solved to optimality is less than 10 for WAMCBP, we show in parentheses the maximum absolute error from the optimal solution in terms of number of items. In Table 5.7, we show the results produced by BP and WAMCBP to the 1350 instances with bin capacity ranging from 4000 to 8000. We see that WAMCBP clearly outperformed BP. WAMCBP found optimal solutions to 2665 instances out of the 2700 instances. BP found optimal solutions to 2519 instances. In Table 5.7, for instances with a larger value for the expected number of items ($\bar{n}$ = 350, 400, 450, 500), large bin capacity (c = 5000, 6000, 7000, 8000) with item size interval [500, 999], BP was not able to solve any of these instances except when $\bar{n}$ = 350 and c = 8000. BP had difficulties solving those instances with a large number of items and a small number of bins, and those large instances with a large average number of items per bin.

In Table 5.8, we show the average running times of BP and WAMCBP to the 1350 instances with bin capacity ranging from 1000 to 3000. In Table 5.9, we show the average running times of BP and WAMCBP to the 1350 instances with bin capacity ranging from 4000 to 8000. For BP, the average computation time for those instances that can be solved to optimality is 2.85 seconds per instance. WAMCBP is faster with an average time of 0.20 second per instance solved to optimality.

		c									
$E(n')$	t_{min}	1000	1200	1500	2000	3000	4000	5000	6000	7000	8000
100	1	50	42	33	25	17	13	10	8	7	6
100	200	60	50	40	30	20	15	12	10	9	7
100	500	75	62	50	37	25	19	15	12	11	9
150	1	75	63	50	38	25	19	15	13	11	9
150	200	90	75	60	45	30	22	18	15	13	11
150	500	112	94	75	56	37	28	22	19	16	14
200	1	100	83	67	50	33	25	20	17	14	13
200	200	120	100	80	60	40	30	24	20	17	15
200	500	150	125	100	75	50	37	30	25	21	19
250	1	125	104	83	63	42	31	25	21	18	16
250	200	150	125	100	75	50	37	30	25	21	19
250	500	187	156	125	94	62	47	37	31	27	23
300	1	150	125	100	75	50	38	30	25	21	19
300	200	180	150	120	90	60	45	36	30	26	22
300	500	225	187	150	112	75	56	45	37	32	28
350	1	175	146	117	88	58	44	35	29	25	22
350	200	210	175	140	105	70	52	42	35	30	26
350	500	262	219	175	131	87	66	52	44	37	33
400	1	200	167	133	100	67	50	40	33	29	25
400	200	240	200	160	120	80	60	48	40	34	30
400	500	300	250	200	150	100	75	60	50	43	37
450	1	225	188	150	113	75	56	45	38	32	28
450	200	270	225	180	135	90	67	54	45	39	34
450	500	337	281	225	169	112	84	67	56	48	42
500	1	250	208	167	125	83	63	50	42	36	31
500	200	300	250	200	150	100	75	60	50	43	37
500	500	375	312	250	187	125	94	75	62	54	47

Table 5.5 Average number of bins for the 2700 instances developed by Peeters et al. (2006).

		c									
E(n')	t_{min}	1000		1200		1500		2000		3000	
		BP	WA	BP	WA	BP	WA	BP	WA	BP	WA
100	1	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	7 (1)	10	10	10	10
	500	10	10	10	10	10	10	10	10	10	10
150	1	10	10	10	9 (1)	10	10	10	10	10	10
	200	10	10	10	10	10	9 (1)	10	10	10	10
	500	10	10	10	10	10	10	10	10	10	10
200	1	10	8 (1)	10	8 (1)	10	10	10	10	10	10
	200	10	10	10	10	10	10	10	10	10	10
	500	10	10	10	10	10	9 (3)	10	10	10	10
250	1	10	9 (1)	10	10	10	10	10	10	10	10
	200	10	10	10	9 (1)	10	9 (1)	10	10	10	10
	500	10	10	10	10	10	10	10	10	10	10
300	1	10	9 (1)	10	8 (1)	10	10	10	10	10	10
	200	10	10	10	10	10	8 (1)	10	10	10	10
	500	10	10	10	10	10	10	10	10	10	10
350	1	10	9 (1)	10	9 (1)	10	10	10	10	10	10
	200	10	10	10	9 (2)	10	10	10	10	10	10
	500	10	10	10	10	10	10	10	10	10	10
400	1	10	8 (1)	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	8 (1)	10	10	10	10
	500	10	10	10	10	10	10	10	10	10	10
450	1	10	9 (1)	10	8 (1)	10	10	10	10	10	10
	200	10	10	10	9 (2)	10	9 (1)	10	10	9	10
	500	10	10	10	10	10	7 (4)	10	10	10	10
500	1	10	9 (1)	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	9 (1)	10	10	9	10
	500	10	10	10	10	10	10	10	10	10	10
	Total	270	261	270	259	270	255	270	270	268	270

WA is the weight annealing algorithm for the maximum cardinality bin packing problem (WAMCBP)
() The maximum absolute error of WAMCBP is shown in the parentheses
Number of instances solved to optimality: BP 1348
WAMCBP 1315

Table 5.6 Number of instances solved to optimality by BP and WAMCBP for instances with small capacity bins.

		c									
E(n')	t_{min}	4000		5000		6000		7000		8000	
		BP	WA	BP	WA	BP	WA	BP	WA	BP	WA
100	1	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	10	10	10	10	10
	500	10	10	10	10	10	10	10	10	10	10
150	1	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	10	10	10	10	10
	500	10	10	10	10	10	10	10	10	10	10
200	1	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	10	10	10	10	10
	500	10	10	10	10	10	10	10	10	10	10
250	1	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	10	10	10	10	10
	500	10	10	10	10	10	10	10	10	10	10
300	1	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	10	10	10	10	10
	500	10	10	1	10	3	10	3	10	10	10
350	1	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	10	10	10	10	10
	500	10	10	0	10	0	10	0	10	9	10
400	1	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	10	10	10	10	10
	500	10	10	0	10	0	10	0	10	0	10
450	1	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	10	10	10	9	10
	500	10	10	0	10	0	10	0	10	0	10
500	1	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	10	10	10	9	10
	500	7	10	0	10	0	10	0	10	0	10
	Total	267	270	221	270	223	270	223	270	237	270

WA is the weight annealing algorithm for the maximum cardinality bin packing problem (WAMCBP)
Number of instances solved to optimality: BP 1171
WAMCBP 1350

Table 5.7 Number of instances solved to optimality by BP and WAMCBP for instances with large capacity bins.

		c									
E(n')	t_{min}	1000		1200		1500		2000		3000	
		BP	WA	BP	WA	BP	WA	BP	WA	BP	WA
100	1	0.1	0.2	0.1	0.9	0.3	0.3	0.0	0.0	0.0	0.0
	200	0.1	0.0	0.1	0.0	0.4	0.2 (7)	0.1	0.0	0.0	0.0
	500	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.0	1.4	0.0
150	1	0.1	0.0	0.1	0.6 (9)	2.3	0.1	0.0	0.1	0.0	0.1
	200	0.1	0.0	0.1	0.6	2.2	0.2 (9)	1.7	0.1	0.0	0.1
	500	0.0	0.0	0.0	0.0	0.1	0.1	4.8	0.0	8.0	0.0
200	1	0.5	0.1 (8)	3.0	0.6 (8)	6.0	0.2	0.0	0.2	0.0	0.2
	200	0.2	0.0	0.3	0.3	6.1	8.9	3.3	0.2	0.0	0.2
	500	0.0	0.0	0.1	0.0	0.2	0.1 (9)	6.1	0.0	24.3	0.1
250	1	1.7	0.1 (9)	1.9	6.6	11.8	0.2	0.0	0.2	0.0	0.2
	200	0.3	0.0	0.8	0.1 (9)	14.9	9.3 (9)	15.5	0.3	0.0	0.2
	500	0.0	0.0	0.1	0.3	0.3	0.5	13.4	0.1	36.4	0.1
300	1	1.3	1.4 (9)	7.2	3.4 (8)	15.8	0.6	10.7	0.6	0.0	0.8
	200	0.5	0.0	0.6	0.9	27.4	5.0	43.8	0.6	39.8	0.5
	500	0.4	0.0	0.1	0.0	0.4	0.4	16.9	0.1	71.7	0.2
350	1	2.1	0.1 (9)	29.3	5.4 (9)	0.0	0.9	0.0	1.1	0.0	1.5
	200	0.6	0.1	1.4	0.1 (9)	41.2	11.7	86.6	1.2	94.1	0.9
	500	0.0	0.0	0.1	0.1	0.6	0.8	27.7	0.1	117.9	0.3
400	1	3.3	0.1 (8)	47.2	1.8	52.4	1.3	0.0	1.8	0.0	2.6
	200	0.8	0.1	1.6	0.0	68.7	6.3 (8)	139.1	1.9	0.1	1.3
	500	0.0	0.0	0.2	0.1	0.8	2.5	39.5	0.2	165.0	0.4
450	1	8.2	5.2 (9)	46.5	18.2 (8)	128.6	1.8	35.2	2.3	0.0	3.3
	200	1.0	0.1	2.3	0.2 (9)	88.8	4.2 (9)	207.5	3.0	0.1 (9)	1.7
	500	0.0	0.0	0.2	0.1	0.7	1.0 (7)	50.7	0.3	237.4	0.6
500	1	13.2	0.2 (9)	64.9	8.6	71.7	2.9	7.7	3.6	0.0	6.0
	200	1.6	0.1	1.7	0.1	127.6	5.2 (9)	374.7	4.7	0.4 (9)	2.3
	500	0.0	0.0	0.2	0.1	1.1	1.8	58.1	0.4	250.4	0.9

WA is the weight annealing algorithm for the maximum cardinality bin packing problem (WAMCBP)

Table 5.8 Average running time in seconds for BP and WAMCBP for instances with small capacity bins.

E(n')	t_{min}	4000		5000		6000		7000		8000	
		BP	WA	BP	WA	BP	WA	BP	WA	BP	WA
100	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	200	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.0
	500	0.6	0.0	1.5	0.0	0.6	0.0	0.0	0.0	0.0	0.1
150	1	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.8	0.0	0.1
	200	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.2
	500	13.9	0.1	10.9	0.1	10.3	0.1	0.3	0.1	0.7	0.2
200	1	0.0	0.3	0.0	0.3	0.0	0.3	0.0	0.3	0.0	0.2
	200	0.0	0.3	0.0	0.3	0.0	0.3	0.0	0.4	0.0	0.4
	500	47.8	0.1	41.2	0.1	23.9	0.1	0.8	0.2	9.7	0.3
250	1	0.0	0.2	0.0	9.2	0.0	0.1	0.0	0.1	0.0	0.1
	200	0.0	0.2	0.0	0.2	0.0	0.1	0.0	0.1	0.0	0.1
	500	98.8	0.1	268.6	0.2	204.0	0.1	50.4	0.1	6.3	0.1
300	1	0.0	1.2	0.0	1.2	0.0	1.1	0.0	1.1	0.0	1.3
	200	0.0	1.3	0.0	1.2	0.0	1.4	0.1	1.6	0.0	1.5
	500	239.6	0.4	326.6 (1)	0.5	115.3 (3)	0.6	16.7 (3)	1.0	7.4	1.0
350	1	0.0	2.0	0.0	2.1	0.0	2.3	0.0	2.0	0.0	1.8
	200	0.0	2.2	0.0	2.2	0.0	2.3	0.0	2.0	0.0	2.8
	500	318.1	0.6	** (0)	0.7	** (0)	0.9	** (0)	1.1	99.6 (9)	1.5
400	1	0.0	2.9	0.0	3.4	0.0	3.5	0.0	3.6	0.0	2.8
	200	0.0	3.2	0.0	3.3	0.0	4.6	0.0	4.4	0.0	3.6
	500	400.9	0.8	** (0)	1.0	** (0)	1.2	** (0)	1.6	** (0)	2.6
450	1	0.0	5.1	0.0	5.2	0.0	6.0	0.0	5.7	0.0	4.8
	200	0.1	5.0	0.1	4.8	0.0	5.8	0.0	6.8	0.0	6.3
	500	578.4	1.2	** (0)	1.3	** (0)	1.9	** (0)	2.7	** (0)	3.3
500	1	0.0	7.1	0.0	7.3	0.0	8.5	0.0	8.6	0.0	7.6
	200	0.1	6.8	0.2	7.4	0.0	8.2	0.0	9.9	0.0	8.9
	500	693.7	1.5	** (0)	2.0	** (0)	2.4	** (0)	3.1	** (0)	4.3

WA is the weight annealing algorithm for the maximum cardinality bin packing problem (WAMCBP)

** BP cannot solve the 10 instances

Table 5.9 Average running in seconds for BP and WAMCBP for instances with large capacity bins.

5.6 Conclusions

We developed a new algorithm (WAMCBP) that was able to produce high-quality solutions very quickly to the maximum cardinality bin packing problem. When applied to the 920 problem instances developed by Labbé, Laporte, and Martello (2003), WAMCBP was comparable in performance to BP, and outperformed LA in terms of speed and the number of instances solved to optimality. Our computational experiments showed that, WAMCBP clearly outperformed BP when applied to the 2700 problem instances generated by Peeters and Degraeve (2006). When compared to BP, WAMCBP solved more instances to optimality, especially those large instances with large average numbers of items per bin, and was faster.

Chapter 6

The Multidimensional Knapsack Problem

6.1 Introduction

The multidimensional knapsack problem (MDKP) is a knapsack problem with multiple resource constraints, or one constraint consisting of a multidimensional attribute. We are given n items and each item j produces profit p_j (j = 1,…, n) and has a demand t_{ij} for resource i (i = 1,..., d). The objective is to maximize the total profit without exceeding the resource capacities c_i. The formulation of the MDKP is as follows:

$$\text{maximize} \quad \sum_{j=1}^{n} p_j x_j \tag{5.1}$$

subject to

$$\sum_{j=1}^{n} t_{ij} x_j \le c_i \qquad i = 1,...,d$$

$$x_j \in \{0,1\} \qquad j = 1,...,n$$

where x_j = 1 if item j is in the knapsack and 0 otherwise.

The MDKP is also known as the 0-1 multidimensional knapsack problem. Without the loss of generality, we can assume that p_j , t_{ij}, and c_i are positive integers. To avoid trivial constraints, where the total possible demand for resources i is less than the available resource i, we assume that $\sum_{i=1}^{n} t_{ij} \ge c_i, i = 1,...,d.$ To ensure that an item can be packed at all, we have $t_{ij} \le c_i,\ \ j = 1,...,n, i = 1,...,d.$

The practical applications of the multidimensional knapsack problem include multi-period capital budgeting (Lu, Chiu, and Cox 1999), combinatorial auctions (de

Vries and Vokra 2001), projection selection (Kleywegt and Papastavrov 2001), stock cutting (Caprara, Kellerer, Pferschy, and Pisinger 2000), and inventory allocation in assemble-to-order systems (Akcay and Xu 2004).

The MDKP is strongly NP-hard (Garey and Johnson 1979), and has generated considerable interest in the literature. Senju and Toyoda (1968) developed a dual gradient method to find the approximation solution of MDKP. Starting with an infeasible solution, their method follows an effective gradient path to achieve feasibility by dropping the non-rewarding variables one at a time. Toyoda (1975) developed a primal gradient method that improved an initial feasible solution by following the path of the steepest effective gradient. Loulou and Michaelides (1979) integrated the primal gradient method of Toyoda into their greedy-like heuristic to solve the MDKP. Magazine and Oguz (1984) proposed a procedure that combined the dual gradient method and the generalized Lagrange multipliers method. Pirkul (1987) developed a heuristic through the surrogate duality approach to solve MDKP. Volgenant and Zoon (1990) proposed a procedure that improved on Magazine and Oguz's method by computing more than one multiplier at a time and readjusting these values at the end of the algorithm. Chu and Beasly (1998) proposed a genetic algorithm that generated the best solutions to several benchmark problems in the literature.

In this Chapter, we develop a weight annealing algorithm for solving the MDKP. We report the results of our weight annealing algorithm for a set of benchmark problems and compare these results to those produced by other methods.

6.2 Weight Annealing Algorithm for Solving the Multidimensional Knapsack Problem (WAMDKP)

In this section, we present our weight annealing algorithm for solving the MDKP. We denote our algorithm by WAMDKP. WAMDKP has four steps.

Step 1. Construct an initial solution.

Step 2. Assign a weight to each item not in the knapsack to distort its profit. These items will be aggregated in a bin, denoted by bin 0.

Step 3. Perform local search by swapping items between the knapsack and bin 0.

Step 4. Return to Step 2 until a fixed number of iterations is reached.

6.2.1 Initial Solution

We construct an initial solution with the primal greedy heuristic of Kellerer, Pferschy, and Pisinger (2004) in the following way.

- Compute the *efficiency* e_j of each item j, where

$$e_j = \frac{p_j}{\sum_{i=1}^{m} \frac{t_{ij}}{c_i}}.$$

- Sort the *efficiency* e_j in non-increasing order.

- Insert items into the knapsack sequentially without violating $\sum_{j=1}^{n} t_{ij} x_j \le c_i$, $i = 1,\ldots,m$.

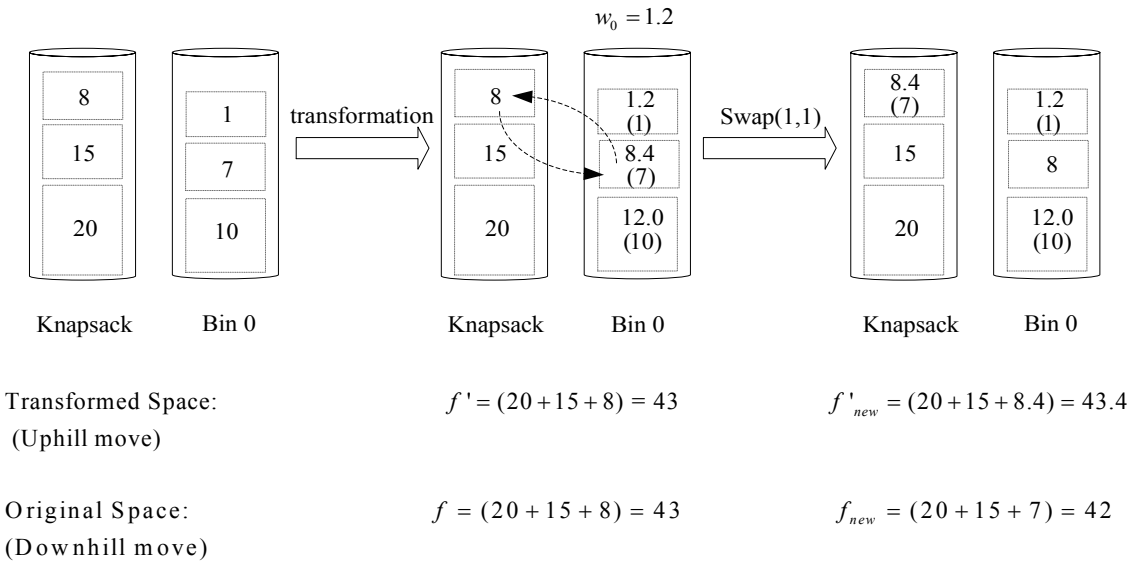


Figure 6.1 An example of an uphill move in the transformed space associated with a downhill move in the original space in MDKP.

6.2.2 Weight Assignments

For WAMDKP, we distort the profit of item *i* in bin 0 by assigning weight according to $w_0 = (1 + K)$, where *K* is the scaling constant. The profit of an item in the knapsack remains unchanged. In Figure 6.1, we show an instance that has an one-dimensional constraint. The optimal solution has the three items of highest efficiencies $(e_1 = 15/5, e_2 = 20/10, e_3 = 8/4)$ in the knapsack. Since weight annealing increases the efficiencies of items in bin 0, we have a move uphill in the transformed space (swapping the item that has a profit of 8 in the knapsack with the item that has a distorted profit of 8.4 in bin 0). This is associated with a move downhill in the original space.

6.2.3 Weight Annealing Algorithm for the Multidimensional Knapsack Problem (WAMDKP)

In Table 6.1, we show our weight annealing algorithm for the multidimensional knapsack problem. The inputs include the knapsack constraints and profits of the items. First, we compute the efficiencies of the items, sort them according to non-increasing efficiency, and store them in an ordered list. We want to generate different initial solutions for different iterations if necessary. To do this, we randomly select the first item from the first $0.2n$ items with the highest efficiencies. If feasible, we insert the item into the knapsack; if not, we insert the item into bin 0. We then pack the second item in the same manner, and so on, until the ordered list is empty.

We improve the current solution by carrying out exchanges of items between the knapsack and bin 0 with weight annealing. We start the first iteration of the j loop by computing the weight for all items in bin 0 according to $w_0^T = (1+K)^T$, and apply the weight to their profits. We then carry out swapping operations. At every iteration, we randomly select one of four swapping operations, Swap(1,0), Swap(1,1), Swap(1,2) and Swap (2,2).

For every swap operation, we need to check that none of the constraints are violated in the original space, and that the swap results in a non-negative change in the objective function value. The algorithm exits after *nloop1* iterations. The total profit, the list of items in the knapsack, and the computation time are the outputs of the algorithm.

Step 0. Initialization
Parameters are K, *nloop1, nloop2, T, Tred*
Set $K = 0.05$, *nloop1* = 20, *nloop2* = 50, $T = 1$, *Tred* = 0.95
Inputs are number of items, item sizes and profits, and knapsack constraints.

Step 1. Optimization runs
for $k = 1$: *nloop1* **do**
Step 1.1 Construct initial solution
Compute efficiencies of items
Sort the items according to non-increasing efficiency into an ordered list
Do while (the ordered list is not empty){
Randomly select an item from the first $0.2n$ items on the ordered list.
Evaluate feasibility
if feasible
Insert the item into the knapsack
else
Insert the item into bin 0
}
Step 1.2. Improve current solution
T:=1
For j =1 to *nloop2* {
Compute weights for items in knapsack: $w_i^T = (1+K)^T$
Perform one of the four swaps with equal probability
for items in the knapsack and bin 0 {

Perform Swap (1,0)
Evaluate feasibility and Δf
if $\Delta f \geq 0$
Move the item
Exit Swap(1,0)
Perform Swap (1,1)
Evaluate feasibility and Δf
if $\Delta f \geq 0$
Swap the items
Exit Swap(1,1)
Perform Swap (1,2)
Evaluate feasibility and Δf
if $\Delta f \geq 0$
Swap the items
Exit Swap(1,2)
Perform Swap (2,2)
Evaluate feasibility and Δf
if $\Delta f \geq 0$
Swap the items
Exit Swap(2,2)

$T := T \times T_{red}$
}
Step 2. Outputs are the total profit and the final selection of items in the knapsack

Table 6.1 Weight annealing algorithm for the multidimensional knapsack problem.

6.3 Computational Experiments

We coded the algorithm in C/C++, and used a 3 GHz Pentium 4 computer with 256 MB of RAM to solve well-known benchmark problems found in the literature.

6.3.1 Benchmark Problems

We use the following benchmark problems.

- Two capital budgeting problems from Weingartner and Ness (1967)
- Two project selection problems from Petersen (1967)
- Two resource planning problems from Senju and Toyoda (1968)
- 270 problem instances from Chu and Beasley (1998)

The capital budgeting, project selection, and resource planning problems are based on real-world application, and are easy to solve. The 270 problems proposed by Chu and Beasley (1998) are large and more difficult to solve. The problems are generated randomly using three parameters as proposed by Fréville and Plateau (1994): number of items (n = 100, 250, 500), number of constraints (d = 5, 10, 15), and tightness ratio (λ = 0.25, 0.50, 0.75).

The coefficients of the knapsack constraint matrix, t_{ij}, were integers randomly generated from the discrete uniform distribution $U(0,1000)$. For each n-d combination, Chu and Beasley generated 30 instances, and the right-hand side coefficients c_i were computed by $c_i = \lambda \sum_{j=1}^{n} t_{ij}$. For the first 10 instances, λ = 0.25; for the next 10 instances, λ = 0.50; and for the last 10 instances, λ = 0.75. The profit of item j, p_j, was generated by

$$p_j = \sum_{i=1}^{d} t_{ij} \Big/ d + 500 q_j \qquad j = 1, \ldots, n$$

where q_j is selected from $U(0,1)$. Past experiments (Pirkul 1987; Martello and Toth 1990; Freville and Plateau 1994) have shown that a strong correlation between p_j and t_{ij} contributes to problem difficulty. The data on instances and best-known solution/optimal results are available at http://people.brunel.ac.uk/~mastjjb/jeb/orlib/ mknapinfo.html and http://elib.zib.de/pub /Packages /mp-testdata/ip/sac94-suite/.

6.3.2 Results for WAMDKP

In Table 6.2, we show the results produced by WAMDKP, the genetic algorithm of Chu and Beasley (1998) and the procedures of Senju and Toyoda (1968), Toyoda (1975), Loulou and Michaelides (1979), Magazine and Oguz (1984), and Pirkul (1987). We see that only WAMDKP and the genetic algorithm of Chu and Beasley (1998) produce optimal solutions for all six instances.

In Table 6.3, we show the results of WAMDKP and GA to the 270 instances of Chu and Beasley (1998). The quality of results are measured by comparing them with the optimal value obtained by LP relaxation or 100(optimal LP value – best solution value)/(optimal LP value). We see that the WAMDKP and GA produce results that are within 1.49% and 0.54% of the LP relaxation, respectively. The average computation times for WAMDKP and GA are 32.0 seconds and 1267.4 seconds, respectively. We note that the two algorithms were tested on different machines.

The WAMDKP is not as accurate as the GA in solving MDKP, because the item exchanges have been confined to only two bins - the knapsack and bin 0. Distortion is applied to bin 0 always as long as $T>0$. This does not exploit fully the main advantage of weight annealing which is to focus computational efforts on poorly solved regions of the search space (poorly packed bins in 1BP) by re-weighting at every iteration.

	Weingartner and Ness (1967)		Petersen(1967)		Senju and Toyoda (1968)	
	Instance (1)	Instance (2)	Instance (1)	Instance (2)	Instance (1)	Instance (2)
n	28	105	39	50	60	60
d	2	2	5	5	30	30
Z^* (optimal value)	141,278	1,095,445	10,618	16,537	7,772	8,722
Z_{WA} (Weight Annealing)	141,278	1,095,445	10,618	16,537	7,772	8,722
Z_{GA} (Chu and Beasley)	141,278	1,095,445	10,618	16,537	7,772	8,722
Z_{SEN} (Senju and Toyoda)	138,888	1,093,181	9,548	16,237	7,590	8,638
Z_{TOY} (Toyoda)	139,278	1,088,365	10,320	15,897	7,719	8,709
Z_{LOU} (Loulou and Michaelides)	135,673	1,083,086	9,290	14,553	7,675	8,563
Z_{MAG} (Magazine and Orguz)	139,418	1,092,971	10,354	16,261	7,719	8,623
Z_{PIR} (Pirkul)	140,477	1,094,575	10,547	16,436	7,728	8,697

Table 6.2 Results generated by seven procedures to the capital budgeting, project selection, and resource allocation instances.

6.4 Conclusions

We developed a weight annealing algorithm (WAMDKP) to solve the multidimensional knapsack problem. When applied to the real-world capital budgeting, project selection, and resource allocation problems, WAMDKP and GA produced optimal solutions to all six instances. When applied to the 270 instances of Chu and Beasley, WAMDKP produced a set of results that are, on average, within 1.49% of the LP relaxation solution in less than 32 seconds. The GA was more accurate; it produced results that are within 0.49% of the LP relaxation on average. The key advantage of WAMDKP here is its simplicity in implementation, and the ability to quickly produce quality results for the multidimensional knapsack problem.

Instances			GA		Weight Annealing	
m	*n*	α	Average Gap below LP(%)	Time(s)	Average Gap below LP(%)	Time(s)
5	100	0.25	0.99	345.9	1.36	3.2
		0.50	0.45	347.3	0.84	2.3
		0.75	0.32	361.7	0.54	1.6
5	250	0.25	0.23	682.0	1.30	17.3
		0.50	0.12	709.4	0.85	11.9
		0.75	0.08	763.3	0.55	8.4
5	500	0.25	0.09	384.1	1.24	47.3
		0.50	0.04	418.9	0.90	31.5
		0.75	0.03	462.6	0.55	22.5
10	100	0.25	1.56	1271.9	2.84	7.1
		0.50	0.79	1345.9	1.45	5.9
		0.75	0.48	1412.6	0.92	3.4
10	250	0.25	0.51	870.9	1.61	50.4
		0.50	0.25	931.5	1.16	40.5
		0.75	0.15	1011.2	0.65	24.0
10	500	0.25	0.24	1504.9	2.72	29.6
		0.50	0.11	1728.8	1.86	19.3
		0.75	0.07	1931.7	1.20	11.0
30	100	0.25	2.91	604.5	3.79	20.4
		0.50	1.34	782.1	1.97	17.5
		0.75	0.83	904.2	1.13	11.4
30	250	0.25	1.19	1499.5	2.27	93.6
		0.50	0.53	1980.0	1.23	100.6
		0.75	0.31	2441.4	0.78	65.0
30	500	0.25	0.61	2437.7	3.33	94.3
		0.50	0.26	3198.9	2.06	88.2
		0.75	0.17	3888.2	1.14	37.0
Average			0.54	1267.4	1.49	32.0

GA	Silicon Graphics Indigo workstation (R4000, 100 MHz)
WAMDKP	Pentium 4 (3 GHz)

Table 6.3 Results of weight annealing and genetic algorithms to the 270 instances by Chu and Beasley (1998).

Chapter 7

The Multidimensional Multiple Knapsack Problem

7.1 Introduction

In the multidimensional multiple knapsack problem (MDMKP), we are given a set of items $N = \{1,\ldots,n\}$, where item j has profit p_j. Unlike the one dimensional bin packing problem where the number of bins is unlimited, we are now given a fixed number of knapsacks $M = \{1,\ldots,m\}$. Each knapsack i (i=1,…,m) has a capacity c_{ik} for resource k, and each item j has a demand t_{ijk} for resource k (k =1,..., d) in knapsack i. A subset $\mathcal{N} \subseteq N$ is feasible if we can assign items of $\mathcal{N}$ to the knapsacks without violating the resource capacity c_{ik}. The objective is to select a feasible subset $\mathcal{N}$ such that the total profit of $\mathcal{N}$ is maximized. The IP formulation of MDMKP is as follows.

$$\text{maximize} \quad \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} \tag{7.1}$$

subject to

$$\sum_{j=1}^{n} t_{ijk} x_{ij} \le c_{ik} \qquad i = 1,\ldots,m, \;\; k = 1,\ldots,d$$

$$\sum_{i=1}^{m} x_{ij} \le 1 \qquad j = 1,\ldots,n$$

$$x_{ij} \in \{0,1\} \qquad i = 1,\ldots,m, \quad j = 1,\ldots,n$$

where variable x_{ij} = 1 if item j is assigned to knapsack i and 0 otherwise.

The thesis by Romaine (1999) introduced the MDMKP. Romaine modeled the military aircraft load-scheduling problem of the US Military Airlift Command as a MDMKP with packing constraints. Specifically, the airlift loading problem involves a

heterogeneous fleet of m transport aircraft each with different weight and volume capacities available for shipping n items. For example, an air tasking order may call for an air transport squadron which is equipped with C-17s (first knapsack type) and C-5s (the second knapsack type), to execute an air lift mission to evacuate high value cargos. Each aircraft type has two technological constraints - a weight carrying constraint and a space (volume) capacity constraint. The mission objective is to maximize the total cargo values without exceeding the weight and volume capacities of the heterogeneous fleet of aircraft. Let b_j be the physical weight of item j, v_j be the volume of item j, c_{ib} be the physical weight aircraft i is capable of holding, and c_{iv} be the volume aircraft i is capable of holding. For this two-dimensional multiple knapsack problem, the formulation is given by.

$$\text{maximize} \quad \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} \tag{7.2}$$

subject to

$$\sum_{j=1}^{n} b_j x_{ij} \le c_{ib} \qquad i = 1,...,m$$

$$\sum_{j=1}^{n} v_j x_{ij} \le c_{iv} \qquad i = 1,...,m$$

$$\sum_{i=1}^{m} x_{ij} \le 1 \qquad j = 1,...,n$$

$$x_{ij} \in \{0,1\} \qquad i = 1,...,m, \quad j = 1,...,n$$

where variable $x_{ij} = 1$ if item j is assigned to aircraft i and 0 otherwise.

Another potential real world application of MDMKP could be in the scheduling of operating theaters in hospitals. Here we are given a fixed number of theaters (or knapsacks) each with a set of technical constraints such as the number of operating hours available per day and the types of surgeries permitted. On any given day, the medical

staff need to perform a subset of a given number of surgeries (or items), each with different priority or profit. The objective function is to maximize the total profits without violating the resource constraints, in terms of operating hours and surgery types, of the operating theaters.

7.2 Generalization of the Multidimensional Multiple Knapsack Problem

The multidimensional multiple knapsack problem is a generalization of the following knapsack problems.

- Multidimensional knapsack problem. If formulation (7.1) has only one knapsack, it becomes a multidimensional knapsack problem.
- Multiple knapsack problem. If each knapsack has only one constraint, formulation (7.1) becomes a multiple knapsack problem.

7.3 Weight Annealing Algorithm for the Multidimensional Multiple Knapsack Problem (WAMDMKP)

The weight annealing algorithm for solving MDMKP, denoted by WAMDMKP, is an extension of WAMDKP that we have developed to solve the multidimensional knapsack problem.

7.3.1 Initial Solution

To construct an initial solution, we propose a greedy procedure.

- Compute the *efficiency* e_j of each item j.

$$e_j = \frac{p_j}{\sum_{i=1}^{m}\sum_{k=1}^{d}\frac{t_{ijk}}{c_{ik}}}$$

- Sort the efficiencies in non-increasing order.

- Insert feasible items into the knapsacks following first-fit decreasing and insert infeasible items into bin 0

7.3.2 Local Search with Weight Annealing

For the local search, we carry out exchanges of items between the knapsacks and bin 0 using Swap(1,0), Swap(1,1), Swap(1,2) and Swap(2,2) with weight annealing. For the MDKP, the exchanges of items were always between bin 0 and the knapsack, with maximizing the total profit as its objective function. However, in MDMKP, we need two objective functions for the two distinct kinds of item exchanges – those between bin 0 and a knapsack and those between two knapsacks. An exchange of items between two knapsacks will not alter the total profit, but it may increase the residual capacities of the knapsacks. Thus, for the two types of exchanges, we are proposing two different objective functions and weight assignments.

For swapping of items between a knapsack and bin 0, the objective function is to maximum the total profit, or $f = \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij}$. We distort the profit of item i in bin 0 by assigning weight according to $w_0 = (1+K)$, where K is the scaling constant. The profit of an item in the knapsack remains unchanged. In Figure 6.1 of Chapter 6, we show an example of an uphill move in the transformed space associated with a downhill move in the original space.

For swapping of items between a pair of knapsacks, the objective function maximizes the sum of the residual resource capacities of the two knapsacks, or maximizes the sum of slacks of the two sets of multidimensional constraints.

$$\text{maximize} \quad f = \sum_{k=1}^{d} c_{ik} - \sum_{j=1}^{n} \sum_{k=1}^{d} t_{ijk} x_{ij} \tag{7.3}$$

Since the first term of (7.3) is a constant, we can rewrite the formulation (7.3) as follows.

$$\text{minimize} \quad f = \sum_{j=1}^{n} \sum_{k=1}^{d} t_{ijk} x_{ij} \tag{7.4}$$

We apply weights to distort the resource demands t_{ijk}. The weights for all items in knapsack i are proportional to r_i which is the sum of residual resource capacities in the knapsack, or $w_i = (1 + Kr_i)$. Given a packing solution $\hat{x}_{ij}$ for knapsack i, where $\hat{x}_{ij} = 1$ if item j is currently in knapsack i and 0 otherwise, we have

$$r_i = \frac{\sum_{k=1}^{d} c_{ik} - \sum_{j=1}^{n} \sum_{k=1}^{d} t_{ijk} \hat{x}_{ij}}{\sum_{k=1}^{d} c_{ik}}$$

or,

$$w_i = \left(1 + K \frac{\sum_{k=1}^{d} c_{ik} - \sum_{j=1}^{n} \sum_{k=1}^{d} t_{ijk} \hat{x}_{ij}}{\sum_{k=1}^{d} c_{ik}} \right) \tag{7.5}$$

In Figure 7.1, we show an MDMKP instance that has three items (n = 3), two knapsacks (m = 2) and each knapsack has one resource constraint (d = 1). The constraint matrix is

$$5.00x_{11} + 4.00x_{12} + 5.00x_{13} \le 10.00$$
$$5.00x_{21} + 4.10x_{22} + 5.00x_{23} \le 10.00.$$

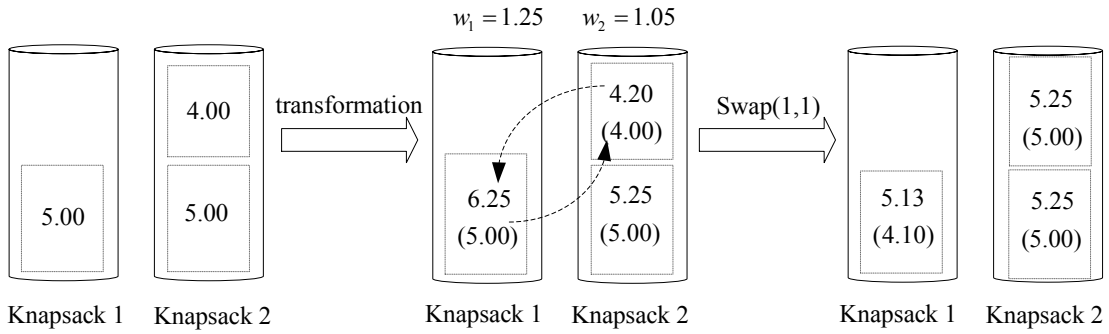
After applying weights to the items (K=0.5), we have a move downhill in the transformed space to minimize the objective function value as stated in (7.4), which is associated with

$d = 1, \quad m = 2, \quad n = 3,$

$c_{11} = 10.00, \quad c_{21} = 10.00, \quad K = 0.5,$

$(t_{1j1}) = (5.00,\ 4.10,\ 5.00),$

$(t_{2j1}) = (5.00,\ 4.00,\ 5.00)$



Transformed Space: $f' = (6.25 + 4.20 + 5.25)$ $f'_{new} = (5.13 + 5.25 + 5.25)$

(downhill move) $= 15.70$ $= 15.63$

Original Space: $f = (5.00 + 4.00 + 5.00)$ $f_{new} = (5.00 + 4.10 + 5.00)$

(Uphill move) $= 14.00$ $= 14.10$

Figure 7.1 An example of a downhill move in the transformed space associated with an uphill move in the original space in MDMKP.

a move uphill in the original space. The swap is carried out as it is feasible in the original space.

7.3.3 Weight Annealing Algorithm

In Table 7.1, we show our weight annealing algorithm for the multidimensional multiple knapsack problem. The inputs are the number of items, item profits, the number of knapsacks, and the knapsack constraint matrix. We first compute the efficiencies of the items, sort them according to non-increasing efficiency, and insert them into an ordered list. We insert the items on the ordered list into the knapsacks and bin 0 following the first-fit decreasing strategy. At each step, we insert the item at the top of the ordered list

into the knapsack with the smallest index, provided none of the *d*-dimensional constraints have been violated, and insert an infeasible item into bin 0, and so on until the item list is empty. Given an initial solution, we compute the sum of residual resource capacity r_i for each knapsack *i*.

We improve the current solution by randomly selecting with equal probability one of four exchanges of items (i.e., Swap(1,0), Swap(1,1), Swap(1,2) and Swap (2,2)) between the knapsacks and bin 0.

We start each iteration of *j* loop by computing the weights (7.5) for all items in the knapsacks and apply the weights to their resource demands $t_{ijk.}$ For every swap operation between two knapsacks, we need to check that none of the *d*-dimensional constraints for the knapsack are violated in the original space. We also need to check that the swap results in a non-positive change in the objective function value as stated in (7.4) in the minimization problem.

We compute the weight, $w_0 = (1+K)$, for all items in bin 0 to distort the profits. For every swap operation between a knapsack and bin 0, we need to check that it does not violate any knapsack constraint in the original space, and results in a non-negative change in the transformed profit sum in the maximization problem.

The algorithm exits after *nloop* iterations, and outputs the total profit, the list of items in the knapsacks, and the computation time.

7.4 Computational Experiments

We coded the algorithm in C/C++ and used a 3 GHz Pentium 4 computer. We point out that there are no benchmark problems available in the open literature, so we generated problem instances to test our algorithm.

Step 0. Initialization
Parameters are *K, nloop, T, Tred*
Set $K = 0.05$, $nloop = 200$, $T = 1$, $Tred = 0.995$
Inputs are number of items, item profits, the number of knapsacks, and the resource constraint matrix

Step 1. Construct initial solution
Compute efficiencies of items
Sort the items according to non-increasing efficiency into an ordered list
Do while (the ordered list is not empty){
Insert the first item on the list into the lowest numbered feasible knapsack
Insert the item into bin 0 if it cannot fit into any knapsack
Remove the item from the ordered list
}
Compute residual capacity r_i

Step2. Improve current solution
T:=1
For j =1 to *nloop* {
Compute weights for items in the knapsacks and bin 0
Perform one of the four swaps with equal probability
for items in the knapsack and bin 0 {

Perform Swap (1,0)
Evaluate feasibility and Δf
if Δf is non-deteriorating
Move the item
Exit Swap(1,0)
Perform Swap (1,1)
Evaluate feasibility and Δf
if Δf is non-deteriorating
Swap the items
Exit Swap(1,1)
Perform Swap (1,2)
Evaluate feasibility and Δf
if Δf is non-deteriorating
Swap the items
Exit Swap(1,2)
Perform Swap (2,2)
Evaluate feasibility and Δf
if Δf is non-deteriorating
Swap the items
Exit Swap(2,2)

$T := T \times T_{red}$

Step 3. Outputs are the total profit and the final selection of items in the knapsack

Table 7.1 Weight annealing algorithm for the multidimensional multiple knapsack problem.

7.4.1 Benchmark Problems

We develop a procedure that is similar to the one used by Fréville and Plateau (1994) to generate a set of 360 problem instances. Our procedure has four parameters: number of items (n = 100, 250, 500), number of knapsacks (m = 2, 5), number of constraints (d = 2, 5), and tightness ratio (λ = 0.4, 0.5, 0.6).

The coefficients of the knapsack constraint matrix, t_{ijk}, are integers randomly generated from the discrete uniform distribution $U(0,1000)$. For each n-m-d combination, we generate 30 instances and the right-hand side coefficients c_{ik} are computed by having $c_{ik} = \lambda \sum_{j=1}^{n} \left(t_{ijk} / m \right)$. We apply the number of knapsacks m as the scaling factor to control the degree of difficulty for a problem instance. For the first 10 instances, λ = 0.4; for the next 10 instances, λ = 0.5; and for the last 10 instances, λ = 0.6. The profit of item j, p_j is given by

$$p_j = \sum_{i=1}^{m} \sum_{k=1}^{d} t_{ijk} \Big/ md + 500 q_j \qquad j = 1, \ldots, n$$

where q_j is drawn from $U(0,1)$.

7.4.2 Results for WAMDMKP

In Table 7.2, we show the results produced by WAMDMKP to 360 problems. We compute the average gap below the LP relaxation solution, (WAMDMKP is a maximization problem), which is defined as 100(optimal LP value – best solution value)/(optimal LP value). We see that the average gap tends to be larger for those problem instances with the following characteristics.

- Low tightness ratios (or tight resource constraints), as observed by the past experimental analysis (Hill and Reilly 2000; Pirkul 1987) for the multidimensional knapsack problem.
- Combination of a small number of items, a large number of constraints, and large number of knapsacks.

Overall, the algorithm managed to obtain solutions within 1.33% of the optimal LP value on average in less than 86.9 seconds on average.

7.5 Conclusions

We developed a weight annealing algorithm to solve the multidimensional multiple knapsack problem. There are currently no benchmark problems available in the literature for the MDMKP. We developed a set of 360 benchmark problems for algorithm testing.

When applied to the 360 instances, WAMDMKP produced a set of results that are, on average, within 1.33% of the LP relaxation solution in less than 86.9 seconds on average.

Number of Items n	Number of knapsacks m	Number of constraints d	Tightness ratio λ	Average gap below LP (%)	Average Time(s)
100	2	2	0.4	0.56	8.3
	2	2	0.5	0.46	10.8
	2	2	0.6	0.47	13.8
	2	5	0.4	1.71	9.9
	2	5	0.5	1.42	12.2
	2	5	0.6	1.19	15.9
	5	2	0.4	2.66	5.4
	5	2	0.5	2.77	2.7
	5	2	0.6	0.15	2.1
	5	5	0.4	4.89	17.2
	5	5	0.5	4.38	13.1
	5	5	0.6	4.37	8.4
250	2	2	0.4	0.26	23.8
	2	2	0.5	0.21	32.2
	2	2	0.6	0.20	41.4
	2	5	0.4	0.86	35.7
	2	5	0.5	0.70	49.7
	2	5	0.6	0.64	66.4
	5	2	0.4	0.95	20.1
	5	2	0.5	0.79	29.2
	5	2	0.6	0.20	59.7
	5	5	0.4	2.31	18.9
	5	5	0.5	2.13	25.3
	5	5	0.6	1.87	35.0
500	2	2	0.4	0.39	186.5
	2	2	0.5	0.36	213.2
	2	2	0.6	0.40	175.4
	2	5	0.4	0.47	402.7
	2	5	0.5	0.51	444.8
	2	5	0.6	0.44	490.3
	5	2	0.4	1.87	70.6
	5	2	0.5	2.07	64.7
	5	2	0.6	0.10	51.2
	5	5	0.4	1.59	184.3
	5	5	0.5	1.77	169.3
	5	5	0.6	1.86	116.8
Average				1.33	86.9

Table 7.2 Results generated by WAMDMKP to the 360 problems

Chapter 8

Conclusions and Future Research

In this dissertation, we developed a weight annealing algorithm for solving some important classes of bin packing and knapsack problems.

We developed a powerful new procedure (WA1BP) that implements the concept of weight annealing to solve the classic one-dimensional bin packing problem. WA1BP is easy to understand and simple to implement, and it generated very high-quality solutions very quickly. When applied to well-known benchmark problems, WA1BP was very competitive with current solutions procedures, and generated new optimal solutions. We also developed WA1DPB that solved the one-dimensional dual bin packing problem. Our computational experiments showed that WA1DPB was competitive with WA1BP in terms of speed and accuracy.

We developed an algorithm based on weight annealing that solved four variants of the two-dimensional bin packing problem, namely problems with guillotine cuts (2BP|O|G and 2BP|R|G) and free cuts (2BP|O|F and 2BP|R|F). Overall, our algorithm produced high-quality results quickly. Specifically, with respect to oriented items and free cutting, our weight annealing algorithm WA2BPF generated results that were very comparable in terms of accuracy and computational speed to the best results found in the literature.

To solve the maximum cardinality bin packing problem, we developed a new algorithm (WAMCBP) that was able to produce high-quality solutions very quickly. Our

computational experiments showed that WAMCBP clearly outperformed all existing procedures when applied to benchmark problems. WAMCBP solved more instances to optimality, especially those large instances with a large average number of items per bin, and was very quick in producing these results.

We developed weight annealing algorithms to solve the multidimensional knapsack problem and the multidimensional multiple knapsack problem. There are currently no benchmark problems available in the literature for the multidimensional multiple knapsack problem. We developed a set of 360 benchmark problems for algorithm testing. When applied to the 360 instances, WAMDMKP quickly produced high-quality results.

In this dissertation, the main contributions have been to describe the concept of weight annealing, develop the weight annealing-based algorithms to solve bin packing, and knapsack problems, and show the high-quality results produced. Future research could go into greater theoretical depth comparing weight annealing and other similar heuristics such as search space smoothing, noising and the genetic algorithm. Future work could also be in identifying the classes of combinatorial optimization problems that could be solved efficiently by weight annealing-based algorithm, and solving current large scale problems in the related industries.

Appendix A

Sample Outputs of the Weight Annealing Algorithm for 1BP

A1 The optimal solution produced by WA1BP for TEST 0044 from Gau1

Bin Size = 10000, Computation Time: 0.16 sec

Bin 1	Bin 2	Bin 3	Bin 4	Bin 5	Bin 6	Bin 7	Bin 8	Bin 9	Bin 10
2443	2410	2341	2251	1893	1944	2491	2325	1254	2204
2247	2204	1768	2197	1877	1721	1901	2197	1254	1042
1893	1710	1768	2197	1877	1551	1901	1721	1235	1042
1651	1399	1651	1893	1254	1185	869	1254	1235	854
682	813	1064	1254	1042	1100	860	818	1185	813
504	411	1042	156	1042	1042	854	419	869	813
394	366	366	41	1015	869	419	354	818	682
186	366				321	394	267	504	483
	321				267	311	267	419	419
							186	384	411
							151	366	384
							41	321	366
								156	354
									133

Bin 11	Bin 12	Bin 13	Bin 14
1651	2341	901	818
1399	1005	869	813
1254	882	869	813
1015	854	818	682
860	854	818	504
860	818	712	483
682	682	682	419
682	504	682	419
419	433	682	419
419	433	682	417
411	366	384	411
307	354	384	394
41	185	366	394
	156	366	384
	133	321	384
		267	366
		156	354
		41	354
			354
			321
			185
			156
			156

Residual Capacities

Bin 1	Bin 2	Bin 3	Bin 4	Bin 5	Bin 6	Bin 7	Bin 8	Bin 9	Bin 10
0	0	0	11	0	0	0	0	0	0

Bin 11	Bin 12	Bin 13	Bin 14
0	0	0	0

A2 The optimal solution produced by WA1DPB for TEST 0044 from Gau1

Bin Size: 10000, Computation Time**: 0.05 sec**

Bin 1	Bin 2	Bin 3	Bin 4	Bin 5	Bin 6	Bin 7	Bin 8	Bin 9	Bin 10
2410	2341	1710	2341	2325	2443	2251	2204	2197	2204
869	1100	1651	1877	1768	1651	2247	1901	1877	1901
712	1042	1235	869	1185	1235	1721	1254	1721	1651
433	1015	1015	869	1042	1064	1254	1042	1254	1399
419	682	813	818	1042	813	901	869	1254	860
419	682	813	504	854	682	813	818	1042	854
419	504	419	419	504	682	813	682	504	818
419	433	394	419	483	417		682	151	311
411	411	394	394	411	384		321		
411	384	384	394	384	321		186		
366	366	384	366		267		41		
354	366	366	366		41				
354	321	156	321						
354	156	133	41						
354	156	133							
354	41								
354									
321									
267									

Bin 11	Bin 12	Bin 13	Bin 14
2197	1944	2491	2197
1893	1893	1768	1893
1254	1551	1399	1254
1005	1042	882	1185
854	818	869	860
854	682	860	818
682	682	818	483
682	384	419	419
267	366	307	366
156	267	186	366
156	185		156
	185		

Residual Capacities

Bin 1	Bin 2	Bin 3	Bin 4	Bin 5	Bin 6	Bin 7	Bin 8	Bin 9	Bin 10
0	0	0	2	2	0	0	0	0	2

Bin 11	Bin 12	Bin 13	Bin 14
0	1	1	3

Appendix B

One Dimensional Bin Packing Problem Code

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//*    WA1BP.cpp is program for solving the one-dimensional bin packing problem.
//*    It is to be complied with the following files (ANSI version) from Press et al.(2002)
//          1)    nrutil.cpp
//          2)    nrutil.h
//          3)    nr.h
//*   The header file sll_node.h is appended at the end of the program.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////


#include <io.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "nrutil.h"
#include "nr.h"
#include <sys\timeb.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include "sll_node.h"
#include <string.h>

Node * bin_root[MAX_NUMBER_BIN];
Node * obj_size;
Node * obj_size_clone;
FILE * fp;
FILE * op;
float binload[MAX_NUMBER_BIN];
float curcap [MAX_NUMBER_BIN];
int nitembin[MAX_NUMBER_BIN];
float bound = 0;
float binsize = 0;
double comptime=0.0;
double totaltime=0.0;
int done =0;
int lowbound =0;
```

```
double TimeElapsed(struct _timeb *begin, struct _timeb *end);
int irbit1(unsigned long *iseed);
void PrintNode(Node * item);
void PrintAllNode(Node * root);
void PrintFileNode(Node * item);
void PrintFile(Node * root);
Node * AddNode(float data, Node * bin);
Node * DeleteNode( Node **linkp, int item_num );
int InsertNode( register Node **linkp, Node * inserted_node);
float CompBinLoad(Node * root);
int CompBinItem(Node * root);
int swap11(Node ** rootp1, Node ** rootp2, int i,int p, float weight[], int numbin);
int swap12(Node ** rootp1, Node ** rootp2, int i,int p, float weight[], int numbin);
int swap22(Node ** rootp1, Node ** rootp2, int i,int p, float weight[], int numbin);
int swap10(Node ** rootp1, Node ** rootp2, int i,int p, float weight[], int numbin);

int
main(void)
{

    struct _timeb time1, time2;
    unsigned long iseed;
    Node * deleted_node;
     Node * deleted_node1;
    Node * deleted_node2;
    Node * inserted_node;
     Node * current_node;
    int i,j,p,m,n;
    int freq = 0;
    int numbitem=0;
    int improved=0;
    int numtype=0;
    float weight [MAX_NUMBER_BIN];
    float itemlist[MAX_NUMBER_ITEM];
    float obinsize = 0;
    float T = 1;
    float Tred =0.95;
    float temp=0.0;
    int numbin =66;
    int numtest =10;
    int test=0;
    char *name;

    if((op=fopen("out.txt", "w")) == NULL) {
        printf("Cannot open file.\n");
        exit(1);
```

```
    }

    /* Input file format – instances of Scholl et al.(1997)*/
    name="HARD0.BPP";
    lowbound=56;

    iseed=111;

    if((fp=fopen( name, "r")) == NULL) {
        printf("Cannot open file.\n");
        exit(1);
    }

    fscanf(fp,"%d", &numbitem);
    fscanf(fp, "%f", &obinsize);

    for(i =(numbitem-1); i >=0;i--){
        fscanf(fp, "%f", &itemlist[i]);
    }


    /* Initialisation*/
    obj_size=NULL;
    obj_size_clone=NULL;
    deleted_node  =NULL;
    deleted_node1 =NULL;
    deleted_node2 =NULL;
    inserted_node =NULL;
    current_node  =NULL;

    for (i=0;i<numbin;i++)
        bin_root[i] = NULL;
    for( i=0;i<numbin;i++)
        curcap[i]=binsize;
    for( i=0;i<numbin;i++){
        binload[i]=0;
        nitembin[i]=0;
    }

/* set bin size to original size*/
    binsize = obinsize;
    bound = 0.0;

    _ftime(&time1);

    for (n=0;n<1;n++){
```

```
/*read data file*/
for (i=0;i<numbitem;i++){
    obj_size = AddNode(itemlist[i], obj_size);
    obj_size_clone=AddNode(itemlist[i], obj_size_clone);
}

/*first-fit decreasing algorithm*/
current_node=obj_size_clone;

for (i=0;i<numbitem;i++){

    for (j=0;j<numbin;j++)
        if (curcap[j] >= current_node->value){
            curcap[j]=curcap[j]- (current_node->value);
            current_node=current_node->link;
            deleted_node= DeleteNode(&obj_size,0);
            InsertNode(&bin_root[j],deleted_node);
            break;
        }
}

/*compute the total load and number of items in each bin*/
for (i=0; i<numbin;i++) {
    binload[i]=CompBinLoad(bin_root[i]);
    nitembin[i]=CompBinItem(bin_root[i]);
}

for (m=0;m<50;m++){

    /* update weights setting K = 0.05*/
    for (i=0;i<numbin ; i++){
        weight[i]= pow((1.0+(curcap[i]/binsize)*0.05), T);
    }

    /*swapping items for all pairs of bins)*/

    for (i=0;i<numbin;i++) {
        improved=0;

        for (p=0;p<numbin;p++){
            if (p!=i){
                improved = swap11(&bin_root[i], &bin_root[p], i, p, weight,
                                                    numbin);
                if (done) break;
```

```
                    improved = swap12(&bin_root[i], &bin_root[p], i, p, weight,
                                                numbin);
                    if (done) break;

                    improved = swap22(&bin_root[i], &bin_root[p], i, p, weight,
                                                numbin);
                    if (done) break;

                    improved = swap10(&bin_root[i], &bin_root[p], i, p, weight,
                                                numbin);
                    if (done) break;
                }
            }
            if (done) break;
        }
        if(done) break;
        T=T*Tred;
    }

    if (done){
        fprintf(op, "Test  %d   done! \n", test);
        printf("Test  %d   done! \n", test);
        break;
    }

    printf("loop %d\n",n);

    obj_size=NULL;
    obj_size_clone=NULL;
    deleted_node  =NULL;
    deleted_node1 =NULL;
    deleted_node2 =NULL;
    inserted_node =NULL;
    current_node  =NULL;

    for (i=0;i<numbin;i++)
        bin_root[i] = NULL;

}

_ftime(&time2);

comptime= TimeElapsed(&time1,&time2);
fprintf(op, "Test  %d     %f \n", test, comptime);

/*print output to files*/
```

```
    fprintf(op, "after swap  \n");
    for(i =0; i <numbin; i++){
        fprintf(op, "%3d  ", i);
        PrintFile(bin_root[i]);
        fprintf(op, " \n");
    }

    fclose(fp);
    fclose(op);
    return EXIT_SUCCESS;

}


/* functions*/
int swap22(Node ** rootp1, Node **rootp2, int i, int p, float weight[], int numbin)
{

  float deltaf;
  int count=0;
  int m=0;
  int n=0;
  int j=0;
  int k=0;
  int q=0;
  int r=0;

 Node * deleted_node1;
 Node * deleted_node2;
 Node * deleted_node3;
 Node * deleted_node4;
 Node * current_node1;
 Node * current_node2;
 Node * current_node3;
 Node * current_node4;

    current_node1=*rootp1;

    for(m=0;m<(nitembin[i]-1);m++){

        k=j+1;
        current_node2=current_node1->link;

        while (current_node2 != NULL){
```

```
q=0;
current_node3=*rootp2;

for(n=0;n<(nitembin[p]-1);n++){

    r=q+1;
    current_node4=current_node3->link;

    while (current_node4!=NULL){

        if (( (current_node1->value + current_node2->value >= current_node3-
                >value +current_node4->value)&&( curcap[p] >= (current_node1-
                >value+current_node2->value -current_node3->value -
                current_node4->value)) )||
          ( ((current_node3->value + current_node4->value) >= current_node1
            >value + current_node2->value) &&( curcap[i] >= (current_node3-
            >value + current_node4->value -current_node1->value-current_node2-
            >value)) )){

            /* short cut equations for change in objective function values*/

            deltaf= SQ(1.0*(binload[i]*weight[i]-(current_node1->value)*
                    weight[i]-(current_node2->value)*weight[i]+(current_node3-
                    >value)*weight[p]+ (current_node4->value)*weight[p]))
                    +SQ(1.0*(binload[p]*weight[p]- (current_node3->value)*
                    weight[p]-(current_node4->value)*weight[p]+(current_node1-
                    >value)*weight[i]+(current_node2->value)*weight[i]))-
                    SQ(1.0*(binload[i]*weight[i]))- SQ(1.0*(binload[p]*weight[p]));

            if (deltaf>=0) {

                binload[i]=binload[i]-(current_node1->value)-
                            (current_node2->value)+(current_node3->
                            value)+(current_node4->value);

                binload[p]=binload[p]-(current_node3->value)-(current_node4-
                            >value)+(current_node1->value)+(current_node2-
                            >value);

                deleted_node2= DeleteNode(rootp1,k);
                deleted_node4= DeleteNode(rootp2,r);
                deleted_node1= DeleteNode(rootp1,j);
                deleted_node3= DeleteNode(rootp2,q);

                InsertNode(rootp1,deleted_node3);
                InsertNode(rootp1,deleted_node4);
```

```
                        InsertNode(rootp2,deleted_node1);
                        InsertNode(rootp2,deleted_node2);

                        /*update curcap after swaps*/
                        done = 0;
                        for (i=0;i<numbin ; i++){
                            curcap[i]= binsize-binload[i];

                            if (nitembin[i]> 0){
                                count++;
                            }
                        }
                        if (count<=lowbound) done=1;
                        count=0;

                        return TRUE;

                    }

                }
                r++;
                current_node4=current_node4->link;
            }
            q++;
            current_node3=current_node3->link;
        }
        k++;
        current_node2=current_node2->link;
    }
    j++;
    current_node1=current_node1->link;
}

return FALSE;
}


int swap12(Node ** rootp1, Node **rootp2, int i, int p, float weight[], int numbin)
{

    float deltaf;
    int count=0;
    int n=0;
    int j=0;
    int q=0;
    int r=0;
```

```
Node * deleted_node1;
Node * deleted_node2;
Node * deleted_node3;
Node * current_node1;
Node * current_node2;
Node * current_node3;

current_node1=*rootp1;

while (current_node1!=NULL){

    q=0;
    current_node2=*rootp2;

    for(n=0;n<(nitembin[p]-1);n++){

            r=q+1;
            current_node3=current_node2->link;

            while (current_node3!=NULL){

                if (( (current_node1->value >= current_node2->value + current_node3-
                    >value) && ( curcap[p] >= (current_node1->value - current_node2-
                    >value - current_node3->value)) )|| ( ((current_node2->value +
                    current_node3->value) >= current_node1->value) &&
                    ( curcap[i] >= (current_node2->value + current_node3->value -
                    current_node1->value)) )){

                    /* short cut equations for change in objective function values*/

                    deltaf= SQ(1.0*(binload[i]*weight[i]-(current_node1-
                            >value)*weight[i]+(current_node2->value)*weight[p]
                            +(current_node3->value)*weight[p]))+
                            SQ(1.0*(binload[p]*weight[p]-(current_node2->value)
                            *weight[p]-(current_node3->value)*weight[p]+
                            (current_node1->value)*weight[i]))-SQ(1.0*(binload[i]*
                            weight[i]))-SQ(1.0*(binload[p]*weight[p]));

                    if (deltaf>=0) {

                        binload[i]=binload[i]-(current_node1->value)+(current_node2-
                        >value)+(current_node3->value);
                        binload[p]=binload[p]-(current_node2->value)-(current_node3-
                        >value)+(current_node1->value);
```

```
                        deleted_node3= DeleteNode(rootp2,r);
                        deleted_node1= DeleteNode(rootp1,j);
                        deleted_node2= DeleteNode(rootp2,q);

                        InsertNode(rootp1,deleted_node2);
                        InsertNode(rootp1,deleted_node3);
                        InsertNode(rootp2,deleted_node1);

                        /*update number of items in bins*/
                        nitembin[p]--;
                        nitembin[i]++;

                        /*update curcap after swaps*/

                        done = 0;
                        for (i=0;i<numbin ; i++){
                            curcap[i]= binsize-binload[i];

                            if (nitembin[i]> 0){
                                count++;
                            }
                        }
                        if (count<=lowbound) done=1;
                        count=0;

                        return TRUE;
                    }
                }
                r++;
                current_node3=current_node3->link;
            }
        q++;
        current_node2=current_node2->link;
    }
    j++;
    current_node1=current_node1->link;
}

return FALSE;
}


int swap11(Node ** rootp1, Node **rootp2, int i, int p, float weight[], int numbin)
{

    float deltaf;
```

```
int count =0;
int j=0;
int q=0;

Node * deleted_node1;
Node * deleted_node2;
Node * current_node1;
Node * current_node2;

current_node1=*rootp1;
current_node2=*rootp2;

while (current_node1!=NULL){

    q=0;
    current_node2=*rootp2;

    while (current_node2!=NULL){


        if (( (current_node1->value >= current_node2->value) &&
            ( curcap[p] >= (current_node1->value - current_node2->value)) )||
          ( (current_node2->value >= current_node1->value) &&
          ( curcap[i] >= (current_node2->value - current_node1->value)) )){

            /* short cut equations for change in objective function values*/

            deltaf=SQ((binload[i]*weight[i]-(current_node1-
                    >value)*weight[i]+(current_node2->value)*weight[p]))+
                    SQ((binload[p]*weight[p]-(current_node2-
                    >value)*weight[p]+(current_node1->value)*weight[i]))-
                    SQ((binload[i]*weight[i]))-SQ((binload[p]*weight[p]));

            if (deltaf>=0) {
                binload[i]=binload[i]-(current_node1->value) +(current_node2->value);
                binload[p]=binload[p]-(current_node2->value) +(current_node1->value);

                deleted_node1= DeleteNode(rootp1,j);
                deleted_node2= DeleteNode(rootp2,q);

                InsertNode(rootp1,deleted_node2);
                InsertNode(rootp2,deleted_node1);

                /*update curcap after swaps*/

                done=0;
```

```
                    for (i=0;i<numbin ; i++){
                        curcap[i]= binsize-binload[i];

                        if (nitembin[i]> 0){
                                count++;

                        }
                    }
                    if (count<=lowbound) done=1;
                    count=0;

                    return TRUE;

                }

            }
            q++;
            current_node2=current_node2->link;
        }
        j++;
        current_node1=current_node1->link;
    }

return FALSE;
}


int swap10(Node ** rootp1, Node **rootp2, int i, int p, float weight[], int numbin)
{

    int j=0;
    int count=0;
    double deltaf;

    Node * deleted_node1;
    Node * current_node1;
    Node * current_node2;

    current_node1=*rootp1;
    current_node2=*rootp2;

    while (current_node1!=NULL){

        if (curcap[p]>= current_node1->value){
```

```
            /* short cut equations for change in objective function values*/

            deltaf=SQ(1.0*(binload[i]*weight[i]-(current_node1->value)*weight[i]))+
                    SQ(1.0*(binload[p]*weight[p]+(current_node1->value)*weight[i]))-
                    SQ(1.0*(binload[i]*weight[i]))-SQ(1.0*(binload[p]*weight[p]));

                if (deltaf>=0) {

                binload[i]=binload[i]-(current_node1->value);
                binload[p]=binload[p]+(current_node1->value);

                deleted_node1= DeleteNode(rootp1,j);
                InsertNode(rootp2,deleted_node1);

                /*update number of items in bins*/
                nitembin[p]++;
                nitembin[i]--;

                /*update curcap after swaps*/

                done=0;
                for (i=0;i<numbin ; i++){
                    curcap[i]= binsize-binload[i];
                        if (nitembin[i]> 0){
                                count++;

                        }
                    }
                if (count<=lowbound) done=1;
                count=0;

                return TRUE;

            }

        }

        j++;
        current_node1=current_node1->link;
    }

return FALSE;
}

/*Count the number of items in a bin*/
int CompBinItem(Node * root)
```

```
{
    int count=0;
    if (root != NULL) {
        while (root != NULL) {
            count++;
            root = root->link;
        }
    } else {
        /*printf("Empty Bin\n");*/
    }
    return count;
}

/*Compute the bin load in a bin*/
float CompBinLoad(Node * root)
{
    float load=0;
    if (root != NULL) {
        while (root != NULL) {
            load = load+(root->value);
            root = root->link;
        }
    } else {
        /*printf(" Empty bin \n");*/
    }
    return load;
}


void PrintFileNode(Node * item)
{
    fprintf(op, "%5.0f  ", item->value);
}


void PrintFile(Node * root)
{
    if (root != NULL) {
        while (root != NULL) {
            PrintFileNode(root);
            root = root->link;
        }
    } else {
        fprintf(op, "No nodes have been entered yet\n");
    }
}
```

```
void PrintNode(Node * item)
{
    printf("\tvalue: %f\n", item->value);
}

void PrintAllNode(Node * root)
{
    if (root != NULL) {
        while (root != NULL) {
            PrintNode(root);
            root = root->link;
        }
    } else {
        printf("Error: No nodes have been entered yet!\n");
    }
}



Node * AddNode(float data, Node * bin )
{
    struct NODE * temp = (struct NODE*) malloc(sizeof(struct NODE));
    temp->value = data;
    temp->link = bin;
    bin = temp;
    return bin;
}

/*Insert nodes in the order of non-increasing size*/
int InsertNode( register Node **linkp, Node * inserted_node)
{
    register Node   *current;
    while( ( current = *linkp ) != NULL &&
        current->value > inserted_node->value )
         linkp = &current->link;

    inserted_node->link = current;
    *linkp = inserted_node;

    return TRUE;
}


Node * DeleteNode( Node **linkp, int item_num )
```

```
{
    Node *current;
    Node *previous;
    Node *delnode;
    int count =0;

    current = *linkp;
    previous = NULL;

    while( current != NULL && count < item_num ){
        count++;
        previous = current;
        current = current->link;
    }

    delnode = current;

    if(previous==NULL)
        *linkp=current->link;
    else
        previous->link=delnode->link;

    if(current!=NULL)
        current = current->link;
    else
        current->link=NULL;

    return  delnode;
}

/*random bit generator for modified first-fit decreasing*/
int irbit1(unsigned long *iseed)
{
    unsigned long newbit;

    newbit =  (*iseed >> 17) & 1
        ^ (*iseed >> 4) & 1
        ^ (*iseed >> 1) & 1
        ^ (*iseed & 1);
    *iseed=(*iseed << 1) | newbit;
    return (int) newbit;
}

double TimeElapsed(struct _timeb *begin, struct _timeb *end)
{float e=end->millitm, b=begin->millitm;
return (end->time+(e/1000))-(begin->time +(b/1000));}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//     sll_node.h is the header file for WA1BP.cpp
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
typedef  struct NODE   {
    struct NODE   *link;
    float value;
} Node;


#define SQ(a) (a*a)
#define  FALSE  0
#define TRUE 1
#define MAX_NUMBER_BIN 100
#define MAX_NUMBER_ITEM 300
```

Appendix C

Two-Dimensional Bin Packing Problem Code
(Guillotine Cuts, Oriented Items)

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//*    WA2BPOG.cpp is the program for solving the two-dimensional bin packing
//problem with guillotine cuts and oriented items.
//*    It is to be complied with the following files (ANSI version) from Press et al.(2002)
//          1)     nrutil.cpp
//          2)     nrutil.h
//          3)     nr.h
//*   The header file 2bp_o_g.h is appended at the end of the program.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////


#include <io.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "nrutil.h"
#include "nr.h"
#include <sys\timeb.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include "2bp_o_g.h"
#include <string.h>
Node * bin_root[MAX_NUMBER_BIN];
Node * vbin_root[MAX_NUMBER_BIN];
Node * hbin_root[MAX_NUMBER_BIN];
Node * obj_size;
Node * strip_size;
FILE * fp;
FILE * op;
int binload[MAX_NUMBER_BIN];
int vbinload[MAX_NUMBER_BIN];
int hbinload[MAX_NUMBER_BIN];
int vbinarea[MAX_NUMBER_BIN];
int vareacap[MAX_NUMBER_BIN];
int hbinarea[MAX_NUMBER_BIN];
int hareacap[MAX_NUMBER_BIN];
```

```
int hmaxht[MAX_NUMBER_BIN];
int hstripsize[MAX_NUMBER_BIN];
int curcap[MAX_NUMBER_BIN];
int vcurcap[MAX_NUMBER_BIN];
int hcurcap[MAX_NUMBER_BIN];
int nitembin[MAX_NUMBER_BIN];
int vnitembin[MAX_NUMBER_BIN];
int hnitembin[MAX_NUMBER_BIN];
int numcell[MAX_NUMBER_BIN];
int ceilheight [MAX_NUMBER_BIN][MAX_NUMBER_HOR];
int ceilwidth  [MAX_NUMBER_BIN][MAX_NUMBER_HOR];
int ceilcap     [MAX_NUMBER_BIN][MAX_NUMBER_HOR];
int floorheight[MAX_NUMBER_BIN][MAX_NUMBER_HOR];
int floorwidth [MAX_NUMBER_BIN][MAX_NUMBER_HOR];
int floorcap   [MAX_NUMBER_BIN][MAX_NUMBER_HOR];
int binsize=0;
int bound=0;
int vbinsize=0;
int hbinsize=0;
int binarea=0;
int lowerbound=19;
int vnumbin=25;
int maxhnumbin=100;
int hnumbin=0;    /* (vnumbin*vbinsize)/average height of item)*/
double comptime=0.0;
double totaltime=0.0;
int done =0;
double TimeElapsed(struct _timeb *begin, struct _timeb *end);
int irbit1(unsigned long *iseed);
void PrintNode(Node * item);
void PrintAllNode(Node * root);
void PrintFileNode(Node * item);
void PrintFile(Node * root);
void removebin (Node **linkp);
void UpdateHposition(Node **rootp);
void UpdateVposition(Node **rootp);
Node * AddNode(int high, int wide, int num1, int num2, Node * bin);
Node * CreatNode(int high, int wide);
Node * DeleteNode( Node **linkp, int item_num );
int HorInsertNode( register Node **linkp, Node * inserted_node);
int VerInsertNode( register Node **linkp, Node * inserted_node);
int CompVbinLoad(Node * root);
int CompHbinLoad(Node * root);
int CompBinItem(Node * root);
int vswap10(Node ** rootp1, Node **rootp2, int x, int y, double vweight[]);
int vswap11(Node ** rootp1, Node **rootp2, int x, int y, double vweight[]);
```

```
int vswap12(Node ** rootp1, Node **rootp2, int x, int y, double vweight[]);
int vswap22(Node ** rootp1, Node **rootp2, int x, int y, double vweight[]);
int hswap10(Node ** rootp1, Node **rootp2, Node * temp_node1, Node * temp_node2,
        int x, int y, int xbin, int ybin, double hweight[]);
int hswap11(Node ** rootp1, Node **rootp2, Node * temp_node1, Node * temp_node2,
        int x, int y, int xbin, int ybin, double hweight[]);
int hswap12(Node ** rootp1, Node **rootp2, Node * temp_node1, Node * temp_node2,
        int x, int y, int xbin, int ybin, double hweight[]);
int hswap22(Node ** rootp1, Node **rootp2, Node * temp_node1, Node * temp_node2,
        int x, int y, int xbin, int ybin, double hweight[]);
int levelpack(Node ** rootp1, Node **rootp2, int s, int t, double hweight[]);



int
main(void)
{
    struct _timeb time1, time2;
    unsigned long iseed;

    Node * deleted_node;
    Node * deleted_node1;
    Node * deleted_node2;
    Node * inserted_node;
    Node * current_node;
    Node * temp_node;
    Node * temp_node1;
    Node * temp_node2;

    int i,j,k,idec,p,q,m,n,s,loop,preht;
    int numbitem=0;
    int improved=0;
    int numtype=0;
    int temp1=0;
    int numstrip=0;
    int empty=0;
    int data=0;
    int count=0;
    int binheight=0;
    int culwidth=0;

    double hweight [MAX_NUMBER_BIN];
    double vweight [MAX_NUMBER_BIN];
    double bweight [MAX_NUMBER_BIN];
    int height[MAX_NUMBER_ITEM];
    int width[MAX_NUMBER_ITEM];
```

```
    double T = 1.0;
    double Tred =0.95;
    float temp=0.0;
    char str[10];

    done=0;

    iseed=111;

    if((op=fopen("out.txt", "w")) == NULL) {
        printf("Cannot open file.\n");
        exit(1);
    }

    /*Input file format – instances from Martello and Toth (1998)*/
    if((fp=fopen( "C1_60_2.txt", "r")) == NULL)
    {       printf("Cannot open file.\n");
        exit(1);
    }

    fscanf(fp, "%d %s %s", &data, &str, &str);
    fscanf(fp, "%d %s %s %s", &numbitem, &str, &str, &str);
    fscanf(fp, "%d %d %s %s %s %s %s %s", &data, &data, &str,
              &str, &str, &str, &str, &str);
    fscanf(fp, "%d %d %s", &vbinsize, &hbinsize, &str);

    for(i =0; i <numbitem;i++){
        if (i==0)
            fscanf(fp, "%d  %d %s", &height[i], &width[i], &str);
        else
            fscanf(fp, "%d  %d ", &height[i], &width[i]);

    }

    bound=0;
    hbinsize=hbinsize+bound;
    vbinsize=vbinsize+bound;
    binarea=vbinsize*hbinsize;

_ftime(&time1);

    for (loop=0;loop<20;loop++){
    printf("\n");
    printf("loop = %d\n", loop);
    printf("\n");
```

```
fprintf(op,"loop = %d\n", loop);

/* Initialisation*/
obj_size=NULL;
strip_size=NULL;
deleted_node  =NULL;
deleted_node1 =NULL;
deleted_node2 =NULL;
inserted_node =NULL;
current_node  =NULL;
temp_node=NULL;
temp_node1=NULL;
temp_node2=NULL;

for (i=0;i<vnumbin;i++)
    vbin_root[i] = NULL;
for (i=0;i<maxhnumbin;i++)
    hbin_root[i] = NULL;
for( i=0;i<vnumbin;i++)
    vcurcap[i]=vbinsize;
for( i=0;i<maxhnumbin;i++)
    hcurcap[i]=hbinsize;
for( i=0;i<vnumbin;i++){
    vbinload[i]=0;
    vnitembin[i]=0;
}
for( i=0;i<maxhnumbin;i++){
    hbinload[i]=0;
    hnitembin[i]=0;
}
for (i=0;i< MAX_NUMBER_BIN;i++){
    for(j=0;j<MAX_NUMBER_HOR; j++){
        ceilheight [i][j]=0;
        ceilwidth  [i][j]=0;
        ceilcap  [i][j]=0;
        floorheight[i][j]=0;
        floorwidth [i][j]=0;
        floorcap   [i][j]=0;

    }
}

for (i=0;i<MAX_NUMBER_BIN;i++)
    numcell[i]=0;

T=1.0;
```

```
Tred=0.95;
numstrip=0;
count=0;

    /*read data file*/
    for (i=0;i<numbitem;i++){
        current_node=CreatNode(height[i], width[i]);
        current_node->area=height[i]*width[i];
        HorInsertNode(&obj_size, current_node);
    }

    /*generate random bit */
    idec=irbit1(&iseed);

    /* modified first-fit decreasing algorithm for horizontal bins*/
    for (i=0;i<numbitem;i++){
        idec=irbit1(&iseed);
        if (i== (numbitem-1)) idec=0;

        deleted_node= DeleteNode(&obj_size,idec);
        for (j=0;j<maxhnumbin;j++)
            if (hcurcap[j] >= deleted_node->width){
                hcurcap[j]=hcurcap[j]- (deleted_node->width);
                deleted_node->stripnum=j;
                HorInsertNode(&hbin_root[j],deleted_node);

                break;
            }
    }

    /*linked list for vertical bins*/
    for (i=(maxhnumbin-1);i>=0;i--){
        if (hbin_root[i] !=NULL){
            numstrip++;
            strip_size = AddNode(hbin_root[i]->height, hbin_root[i]->width,
            hbin_root[i]->stripnum, hbin_root[i]->binnum, strip_size);
        }
    }

    hnumbin=numstrip;

    /*compute the total load and number of items in each bin*/
    for (i=0; i<hnumbin;i++) {
        hbinload[i]=CompHbinLoad(hbin_root[i]);
        hnitembin[i]=CompBinItem(hbin_root[i]);
    }
```

```
/* first-fit decreasing for vertical bins*/
for (i=0;i<hnumbin;i++){

    deleted_node= DeleteNode(&strip_size,0);
    for (j=0;j<vnumbin;j++)
        if (vcurcap[j] >= deleted_node->height){
            vcurcap[j]=vcurcap[j]- (deleted_node->height);

            temp_node=hbin_root[deleted_node->stripnum];
            for (k=0;(k<hnitembin[(deleted_node->stripnum)]);k++){
                temp_node->binnum=j;
                if (temp_node->link != NULL)
                temp_node=temp_node->link;
            }
            temp_node=NULL;

            deleted_node ->binnum=j;
            VerInsertNode(&vbin_root[j],deleted_node);
            break;
        }
}

/*compute the total load and number of items in each bin*/
for (i=0; i<vnumbin;i++) {
    vbinload[i]=CompVbinLoad(vbin_root[i]);
    vnitembin[i]=CompBinItem(vbin_root[i]);
}


/*compute utilised areas in each horizontal bin*/

for (i=0;i<hnumbin;i++)
        hbinarea[i]=0;

for (i=0; i<hnumbin;i++){
    temp_node=hbin_root[i];
    for (k=0;(k<hnitembin[i]);k++){
        hbinarea[i]=hbinarea[i]+temp_node->area;
        temp_node=temp_node->link;
    }
}
```

```
/*Phase 1 of hybrid first-fit*/
for (m=0;m<50;m++){

/* update weights, set K = 0.05*/
for (i=0;i<hnumbin ; i++){
    hweight[i]= pow((1.0+((double)hcurcap[i]/(double)hbinsize)*0.05), T);
}

/*swap item amongst all pairs of levels*/
for (i=0;i<vnumbin;i++) {
    improved=0;

    temp_node1=vbin_root[i];
    for(j=0; j<vnitembin[i];j++){

        for (p=0;p<vnumbin;p++){

            temp_node2=vbin_root[p];
            for(q=0;q<vnitembin[p];q++){

                if  ((temp_node1->stripnum)!=(temp_node2->stripnum)){

                    improved = hswap10(&hbin_root[temp_node1->stripnum],
                        &hbin_root[temp_node2->stripnum], temp_node1,
                        temp_node2,(temp_node1->stripnum),
                        (temp_node2->stripnum),i, p, hweight);

                    if (hbin_root[temp_node1->stripnum]==NULL){
                        removebin(&vbin_root[i]);
                        vnitembin[i]--;
                        break;
                    }
                    if (done) break;

                    improved = hswap11(&hbin_root[temp_node1->stripnum],
                        &hbin_root[temp_node2->stripnum], temp_node1,
                        temp_node2,(temp_node1->stripnum),
                        (temp_node2->stripnum), i, p, hweight);

                    if (done) break;

                    improved = hswap12(&hbin_root[temp_node1->stripnum],
                        &hbin_root[temp_node2->stripnum],  temp_node1,
                        temp_node2,(temp_node1->stripnum),
                        (temp_node2->stripnum), i, p, hweight);
```

```
                        if (done) break;

                        improved = hswap22(&hbin_root[temp_node1->stripnum],
                            &hbin_root[temp_node2->stripnum],  temp_node1,
                            temp_node2,(temp_node1->stripnum),
                            (temp_node2->stripnum), i, p, hweight);

                        if (done) break;

                    }

                    temp_node2=temp_node2->link;

                }
                if (hbin_root[temp_node1->stripnum]==NULL) break;
                if (done) break;
            }
            if (done) break;
            if(temp_node1 !=NULL)
            temp_node1=temp_node1->link;
        }
        if (done) break;
    }
    if(done) break;
    T=T*Tred;

}

   /*Phase 2 of hybrid first-fit*/
   for (m=0;m<20;m++){

      /* update weights, set K=0.05*/

      for (i=0;i<vnumbin ; i++)
            vweight[i]= pow((1.0+((double)vcurcap[i]/(double)vbinsize)*0.05), T);

      /*swap item amongst vertical bins*/
      for (i=0;i<vnumbin;i++) {
            improved=0;

         for (p=0;p<vnumbin;p++){

            if  (p!=i){

               improved = vswap10(&vbin_root[i], &vbin_root[p],i,p, vweight);
               if (done) break;
```

```
                improved = vswap11(&vbin_root[i], &vbin_root[p],i,p,vweight);
                if (done) break;

                improved = vswap12(&vbin_root[i], &vbin_root[p],i,p,vweight);
                if (done) break;

                improved = vswap22(&vbin_root[i], &vbin_root[p],i,p,vweight);
                if (done) break;

            }

        }

        if (done) break;
    }
    if (done) break;
}

/*initialise hposition, vposition, floornum, ceilnum for horizontal levels  after hswap*/

    for (i=0;i<vnumbin;i++){
        temp_node1=vbin_root[i];

        for (j=0;(j<vnitembin[i]);j++){
            s=temp_node1->stripnum;
            temp_node2=hbin_root[s];

            for (k=0;(k<hnitembin[s]);k++){

                temp_node2->binnum=i;
                temp_node2->hposition = k;
                temp_node2->vposition = j;
                temp_node2->floornum = k;
                temp_node2->ceilnum  =-1;
                temp_node2=temp_node2->link;
            }

            temp_node1->binnum=i;
            temp_node1->hposition=0;
            temp_node1->vposition = j;
            temp_node1->floornum = 0;
            temp_node1->ceilnum  =-1;
            temp_node1->area=temp_node1->height * temp_node1->width;
            temp_node1=temp_node1->link;
        }
```

```
        }

/* determine space occupied by items, and residual areas in each level */

        for (i=0;i<hnumbin;i++){

            if (hbin_root[i]!=NULL){

                numcell[i]=hnitembin[i];
                temp_node1=hbin_root[i];
                preht = 0;
                culwidth=0;

                for (j=0;(j<hnitembin[i]);j++){

                    if (j==0){
                        culwidth=temp_node1->width;
                        if (temp_node1->vposition == (vnitembin[temp_node1->binnum]-1)){
                            ceilheight [i][0]=temp_node1->height + vcurcap[temp_node1-
                                        >binnum];
                            preht=temp_node1->height+ vcurcap[temp_node1->binnum];
                            ceilheight [i][numcell[i]]=vcurcap[temp_node1->binnum];
                            ceilwidth  [i][numcell[i]]=temp_node1->width;
                            ceilcap    [i][numcell[i]]=ceilwidth [i][numcell[i]];
                            floorheight[i][numcell[i]]=0;
                            floorwidth [i][numcell[i]]=0;
                            floorcap   [i][numcell[i]]=0;
                            numcell[i]++;
                        }
                        else{
                            ceilheight [i][0]=temp_node1->height;
                            preht=temp_node1->height;
                        }
                        ceilwidth  [i][0]=hbinsize-culwidth;
                        ceilcap    [i][0]=ceilwidth[i][0];
                        floorheight[i][0]=temp_node1->height;
                        floorwidth [i][0]=temp_node1->width;
                        floorcap   [i][0]=0;
                    }

                    else if (j==1){
                        ceilheight [i][0]=preht - temp_node1->height;
                        culwidth        =culwidth+temp_node1->width;
                        ceilheight [i][1]=temp_node1->height;
                        ceilwidth  [i][1]=hbinsize-culwidth;
                        ceilcap    [i][1]=ceilwidth[i][1];
```

```
                    floorheight[i][1]=temp_node1->height;
                    floorwidth [i][1]=temp_node1->width;;
                    floorcap   [i][1]=0;
                    preht=temp_node1->height;

                }

                else {
                    ceilheight [i][j-1]=preht - temp_node1->height;
                    ceilwidth  [i][j-1]=temp_node1->width;
                    ceilcap    [i][j-1]=ceilwidth[i][j-1];
                    culwidth           =culwidth+temp_node1->width;
                    ceilheight [i][j]  =preht;
                    ceilwidth  [i][j]  =hbinsize-culwidth;
                    ceilcap    [i][j]  =ceilwidth[i][j];
                    floorheight[i][j]  =temp_node1->height;
                    floorwidth [i][j]  =temp_node1->width;;
                    floorcap   [i][j]  =0;

                }
                temp_node1=temp_node1->link;

            }

        }

    }


/*compute utilised areas in each horizontal bin*/

    for (i=0;i<hnumbin;i++){
            hbinarea[i]=0;
            hareacap[i]=0;
    }
    for (i=0; i<hnumbin;i++){
        if (hbin_root[i]!=NULL){
            temp_node=hbin_root[i];
            for (k=0;(k<hnitembin[i]);k++){
                if (k==0){
                    hmaxht[i]=temp_node->height;
                    hstripsize[i]=hmaxht[i]*hbinsize;
            }

                hbinarea[i]=hbinarea[i]+temp_node->area;
                temp_node=temp_node->link;
```

```
            }
            hareacap[i]=hstripsize[i]-hbinarea[i];
        }

    }

    for (i=0;i<vnumbin;i++){
        vbinarea[i]=0;
        vareacap[i]=binarea;
    }
    for (i=0; i<vnumbin;i++){
        if (vbin_root[i]!=NULL){

        temp_node1=vbin_root[i];
        for (j=0;(j<vnitembin[i]);j++){

            vbinarea[i]=vbinarea[i]+hbinarea[temp_node1->stripnum];
            temp_node1=temp_node1->link;
        }
        vareacap[i]=binarea-vbinarea[i];

        }
    }



/*filling residual space in horizontal levels */
T=1.0;

for (n=0;n<20;n++){

    for (i=0;i<vnumbin ; i++)
        bweight[i]= pow( (1.0+((double)vareacap[i]/(double)binarea)*0.05), T);

    for (i=0;i<hnumbin;i++){

        if (hbin_root[i]!=NULL)
            vweight[i]=bweight[hbin_root[i]->binnum];
        else
            vweight[i]=0.0;

    }
    for (i=0;i<hnumbin;i++)
        hweight[i]= pow( (1.0+((double)hareacap[i]/(double)hbinarea[i])*0.05), T);
```

```
    /*filling the residual space within each level */
    for (i=0;i<hnumbin;i++) {
        improved=0;

        if (hbin_root[i]!=NULL){

            for(j=0; j<hnumbin;j++){

                if (hbin_root[j]!=NULL){

                    if  (i!=j){

                        if (hbin_root[i]->binnum == hbin_root[j]->binnum)
                            improved = levelpack(&hbin_root[i], &hbin_root[j], i, j, hweight);
                        else
                            improved = levelpack(&hbin_root[i], &hbin_root[j], i, j, vweight);

                        if (hbin_root[i]==NULL)
                        break;
                        if (done) break;

                    }

                }

            }
            if (done) break;
        }

    }

    T=T*Tred;

}

count=0;
for (i=0;i<hnumbin;i++){
    count=count+hnitembin[i];
}

count=0;
for (i=0;i<vnumbin;i++){
    if (vbin_root[i]!=NULL)
        count++;
}
```

```
if (count==lowerbound){
    done=1;
}

if(done)
    break;
}   /* end of loop*/


_ftime(&time2);

    comptime= TimeElapsed(&time1,&time2);
    totaltime=totaltime+comptime;
    fprintf(op,"\n");
    fprintf(op, "Subroutine Elapsed time:%f \n", TimeElapsed(&time1,&time2));
    fprintf(op, "Program Elapsed time:  %f\n", (float)clock()/CLOCKS_PER_SEC);
    printf("Subroutine Elapsed time:%f \n", TimeElapsed(&time1,&time2));

    fclose(fp);
    fclose(op);
    return EXIT_SUCCESS;

}


/* functions*/

/*filling the residual space within each level */
int levelpack(Node ** rootp1, Node **rootp2, int s, int t, double weight[])
{
    int k,n,i,j,p,q;
    int placement =0;
    int preposn = 0;
    double deltaf;
    int count=0;

    Node * deleted_node1;
    Node * deleted_node2;
    Node * current_node1;
    Node * temp_node1;
    Node * temp_node2;

    current_node1=*rootp1;

    p=(*rootp1)->binnum;
```

```
q=(*rootp2)->binnum;

for (k=0;(k<hnitembin[s]);k++){

    for (n=0;(n<numcell[t]);n++){

        if ((current_node1->width <= ceilcap[t][n]) &&
            (current_node1->height<= ceilheight[t][n])){
            placement =1;

        }
        else if ((current_node1->width <= floorcap[t][n]) &&
                (current_node1->height<= floorheight[t][n])){
            placement=2;

        }

        if (placement){

            if ((*rootp1)->binnum == (*rootp2)->binnum)
                deltaf=SQ(1.0*(hbinarea[s]*weight[s]-(current_node1->area)*weight[s]))
                  +SQ(1.0*(hbinarea[t]*weight[t]+(current_node1->area)*weight[s]))
                  -SQ(1.0*(hbinarea[s])*weight[s])-SQ(1.0*(hbinarea[t])*weight[t]);
            else
                deltaf=SQ(1.0*(vbinarea[p]*weight[s]-(current_node1->area)*weight[s]))
                  +SQ(1.0*(vbinarea[q]*weight[t]+(current_node1->area)*weight[s]))
                  -SQ(1.0*(vbinarea[p])*weight[s])-SQ(1.0*(vbinarea[q])*weight[t]);

            if (deltaf>0){


                vbinarea[p]=vbinarea[p]-(current_node1->area);
                vbinarea[q]=vbinarea[q]+(current_node1->area);
                vareacap[p]=binarea-vbinarea[p];
                vareacap[q]=binarea-vbinarea[q];

                hbinarea[s]=hbinarea[s]-(current_node1->area);
                hbinarea[t]=hbinarea[t]+(current_node1->area);
                hareacap[s]=hstripsize[s]-hbinarea[s];
                hareacap[t]=hstripsize[t]-hbinarea[t];

                /* update node data for bins*/
                if (placement == 1){

                    ceilcap[t][n]=ceilcap[t][n]- current_node1->width;
```

```
    if (current_node1 ->floornum>=0)
        floorcap[s][current_node1->floornum]=floorcap[s][current_node1-
            >floornum] + current_node1->width;
    else if (current_node1 ->ceilnum>=0)
        ceilcap[s][current_node1->ceilnum]=ceilcap[s][current_node1-
            >ceilnum] + current_node1->width;

    current_node1->stripnum = t;
    current_node1->ceilnum  = n;
    current_node1->floornum =-1;

}
else if (placement == 2){

    floorcap[t][n]=floorcap[t][n]- current_node1->width;

    if (current_node1 ->floornum>=0)
        floorcap[s][current_node1->floornum]=floorcap[s][current_node1-
            >floornum] + current_node1->width;
    else if (current_node1 ->ceilnum>=0)
        ceilcap[s][current_node1->ceilnum]=ceilcap[s][current_node1-
            >ceilnum] + current_node1->width;

    current_node1->stripnum = t;
    current_node1->ceilnum  =-1;
    current_node1->floornum = n;
}

deleted_node1= DeleteNode(rootp1,k);
UpdateHposition(rootp1);
hnitembin[s]--;

preposn = deleted_node1->vposition;
deleted_node1->binnum    =(*rootp2)->binnum;
deleted_node1->vposition=(*rootp2)->vposition;

HorInsertNode(rootp2,deleted_node1);
UpdateHposition(rootp2);
hnitembin[t]++;

if (((*rootp1)==NULL)&&(deleted_node1!=NULL)){

    deleted_node2= DeleteNode(&vbin_root[p], preposn);
    UpdateVposition(&vbin_root[p]);
    vnitembin[p]--;
```

```
                    temp_node1=vbin_root[p];
                    for (i=0;(i<vnitembin[p]);i++){

                        temp_node2=hbin_root[temp_node1->stripnum];

                        for(j=0;j<hnitembin[temp_node1->stripnum];j++){
                            temp_node2->vposition=i;
                            temp_node2=temp_node2->link;
                        }

                    temp_node1=temp_node1->link;
                    }

                }

                for (i=0;i<vnumbin ; i++){
                    if (vbin_root[i]!=NULL)
                        count++;
                }
                if (count == lowerbound){
                    done=1;
                }
                return TRUE;
            }

        }
        placement = 0;

    }
    current_node1=current_node1->link;

}

return FALSE;
}

/*swapping items between vertical bins*/
int vswap22(Node ** rootp1, Node **rootp2, int x, int y, double  vweight[])
{

    double  deltaf=0.0;
    int m=0;
    int n=0;
    int j=0;
    int k=0;
    int q=0;
```

```
int r=0;
int i=0;
int count=0;

Node * deleted_node1;
Node * deleted_node2;
Node * deleted_node3;
Node * deleted_node4;
Node * current_node1;
Node * current_node2;
Node * current_node3;
Node * current_node4;
Node * temp_node;

current_node1=*rootp1;

for(m=0;m<(nitembin[x]-1);m++){

    k=j+1;
    current_node2=current_node1->link;

    while (current_node2 != NULL){

        q=0;
        current_node3=*rootp2;

        for(n=0;n<(nitembin[y]-1);n++){

            r=q+1;
            current_node4=current_node3->link;

            while (current_node4!=NULL){

                if ( ( vcurcap[y] >= (current_node1->height+current_node2->height -
                        current_node3->height-current_node4->height))
                  &&( vcurcap[x] >= (current_node3->height+current_node4->height -
                        current_node1->height-current_node2->height)) ){

                    deltaf= SQ(1.0*(vbinload[x]*vweight[x]-(current_node1-
                            >height)*vweight[x]-(current_node2->height)*vweight[x]
                            +(current_node3->height)*vweight[y]+(current_node4-
                            >height)*vweight[y]))+SQ(1.0*(vbinload[y]*vweight[y]-
                            (current_node3->height)*vweight[y]-(current_node4-
                            >height)*vweight[y]+(current_node1->height)
                            *vweight[x]+(current_node2->height)*vweight[x]))-
                            SQ(1.0*(vbinload[x]*vweight[x]))- SQ(1.0*
```

```
            (vbinload[y]*vweight[y]));

if (deltaf>=0) {

    vbinload[x]=vbinload[x]-(current_node1->height)-(current_node2-
        >height)+(current_node3->height)+(current_node4->height);

    vbinload[y]=vbinload[y]-(current_node3->height)-(current_node4-
        >height)+(current_node1->height)+(current_node2->height);

    deleted_node2= DeleteNode(rootp1,k);
    deleted_node4= DeleteNode(rootp2,r);
    deleted_node1= DeleteNode(rootp1,j);
    deleted_node3= DeleteNode(rootp2,q);

    temp_node=hbin_root[deleted_node1->stripnum];
    for (j=0;(j<hnitembin[(deleted_node1->stripnum)]);j++){
        temp_node->binnum=y;
        if (temp_node->link != NULL)
            temp_node=temp_node->link;
    }

    temp_node=hbin_root[deleted_node2->stripnum];
    for (j=0;(j<hnitembin[(deleted_node2->stripnum)]);j++){
        temp_node->binnum=y;
        if (temp_node->link != NULL)
        temp_node=temp_node->link;
    }

    temp_node=hbin_root[deleted_node3->stripnum];
    for (j=0;(j<hnitembin[(deleted_node3->stripnum)]);j++){
        temp_node->binnum=x;
        if (temp_node->link != NULL)
        temp_node=temp_node->link;
    }

    temp_node=hbin_root[deleted_node4->stripnum];
    for (j=0;(j<hnitembin[(deleted_node4->stripnum)]);j++){
        temp_node->binnum=x;
        if (temp_node->link != NULL)
        temp_node=temp_node->link;
    }

    deleted_node1->binnum=y;
    deleted_node2->binnum=y;
    deleted_node3->binnum=x;
```

```
                    deleted_node4->binnum=x;

                    VerInsertNode(rootp1,deleted_node3);
                    VerInsertNode(rootp1,deleted_node4);
                    VerInsertNode(rootp2,deleted_node1);
                    VerInsertNode(rootp2,deleted_node2);

                    /*update curcap after swaps*/
                    for (i=0;i<vnumbin ; i++){
                        if (vbinload[i]>0)
                            count++;
                        vcurcap[i]= vbinsize-vbinload[i];
                    }
                    if (count == lowerbound)
                        done=1;
                    if (done){
                        printf("\n");
                        printf(" Done! vswap11. \n");
                        printf("\n");
                    }
                    return TRUE;

                }

            }
            r++;
            current_node4=current_node4->link;
        }
        q++;
        current_node3=current_node3->link;
    }
    k++;
    current_node2=current_node2->link;
}
j++;
current_node1=current_node1->link;
}

return FALSE;
}

/*swapping items between vertical bins*/
int vswap12(Node ** rootp1, Node **rootp2, int x, int y, double vweight[])
{
    double deltaf=0.0;
    int i=0;
```

```
int n=0;
int j=0;
int q=0;
int r=0;
int count=0;

Node * deleted_node1;
Node * deleted_node2;
Node * deleted_node3;
Node * current_node1;
Node * current_node2;
Node * current_node3;
Node * temp_node;

current_node1=*rootp1;


while (current_node1!=NULL){

    q=0;
    current_node2=*rootp2;

    for(n=0;n<(vnitembin[y]-1);n++){

            r=q+1;
            current_node3=current_node2->link;

            while (current_node3!=NULL){

                if (  (vcurcap[y] >= (current_node1->height - current_node2->height –
                        current_node3->height))
                   &&(vcurcap[x] >= (current_node2->height + current_node3->height –
                        current_node1->height)) ){

                  deltaf=SQ(1.0*(vbinload[x]*vweight[x]-(current_node1->height)*
                          vweight[x]+(current_node2->height)*vweight[y]+(current_node3-
                          >height)*vweight[y]))+   SQ(1.0*(vbinload[y]*vweight[y]-
                          (current_node2->height)*vweight[y]-(current_node3-
                          >height)*vweight[y]+(current_node1->height)*vweight[x]))-
                          SQ(1.0*(vbinload[x]*vweight[x]))-
                          SQ(1.0*(vbinload[y]*vweight[y]));

                   if (deltaf>=0) {
                      vbinload[x]=vbinload[x]-(current_node1->height)+(current_node2-
                         >height)+(current_node3->height);
                      vbinload[y]=vbinload[y]-(current_node2->height)-(current_node3-
```

```
    >height)+(current_node1->height);

deleted_node3= DeleteNode(rootp2,r);
deleted_node1= DeleteNode(rootp1,j);
deleted_node2= DeleteNode(rootp2,q);

temp_node=hbin_root[deleted_node1->stripnum];
for (j=0;(j<hnitembin[(deleted_node1->stripnum)]);j++){
    temp_node->binnum=y;
    if (temp_node->link != NULL)
    temp_node=temp_node->link;
}

temp_node=hbin_root[deleted_node2->stripnum];
for (j=0;(j<hnitembin[(deleted_node2->stripnum)]);j++){
    temp_node->binnum=x;
    if (temp_node->link != NULL)
    temp_node=temp_node->link;
}

temp_node=hbin_root[deleted_node3->stripnum];
for (j=0;(j<hnitembin[(deleted_node3->stripnum)]);j++){
    temp_node->binnum=x;
    if (temp_node->link != NULL)
    temp_node=temp_node->link;
}

deleted_node1->binnum=y;
deleted_node2->binnum=x;
deleted_node3->binnum=x;

VerInsertNode(rootp1,deleted_node2);
VerInsertNode(rootp1,deleted_node3);
VerInsertNode(rootp2,deleted_node1);

/*update number of items in bin*/
vnitembin[y]--;
vnitembin[x]++;

/*update curcap after swaps*/
for (i=0;i<vnumbin ; i++){
    if (vbinload[i]>0)
        count++;
    vcurcap[i]= vbinsize-vbinload[i];
}
if (count == lowerbound)
```

```
                        done=1;
                    if (done){
                        printf("\n");
                        printf(" Done! vswap12. \n");
                        printf("\n");
                    }

                        return TRUE;

                    }

                }
                r++;
                current_node3=current_node3->link;
            }
        q++;
        current_node2=current_node2->link;
    }
    j++;
    current_node1=current_node1->link;
}

return FALSE;
}

/*swapping items between vertical bins*/
int vswap11(Node ** rootp1, Node **rootp2, int x, int y, double vweight[])
{

    double deltaf=0.0;
    int i=0;
    int j=0;
    int q=0;
    int count=0;

    Node * deleted_node1;
    Node * deleted_node2;
    Node * current_node1;
    Node * current_node2;
    Node * temp_node;

    current_node1=*rootp1;
    current_node2=*rootp2;

    while (current_node1!=NULL){
```

```
q=0;
current_node2=*rootp2;

while (current_node2!=NULL){
    if (    (vcurcap[y] >= ((current_node1->height)- current_node2->height))
        && (vcurcap[x] >= ((current_node2->height)- current_node1->height)) ){


        deltaf=SQ(1.0*(vbinload[x]*vweight[x]-(current_node1->height)*vweight[x]
              +(current_node2-height)*vweight[y]))+SQ(1.0*(vbinload[y]*vweight[y]
              -(current_node2->height)*vweight[y]+(current_node1->height)*
              vweight[x])) -SQ(1.0*(vbinload[x]*vweight[x]))-
              SQ(1.0*(vbinload[y])*vweight[y]);

        if (deltaf>=0 ) {

            vbinload[x]=vbinload[x]-(current_node1->height) +(current_node2-
                >height);
            vbinload[y]=vbinload[y]-(current_node2->height) +(current_node1-
                >height);

            deleted_node1= DeleteNode(rootp1,j);
            deleted_node2= DeleteNode(rootp2,q);

            temp_node=hbin_root[deleted_node1->stripnum];
            for (j=0;(j<hnitembin[(deleted_node1->stripnum)]);j++){
                temp_node->binnum=y;
                if (temp_node->link != NULL)
                temp_node=temp_node->link;
            }

            temp_node=hbin_root[deleted_node2->stripnum];
            for (j=0;(j<hnitembin[(deleted_node2->stripnum)]);j++){
                temp_node->binnum=x;
                if (temp_node->link != NULL)
                temp_node=temp_node->link;
            }

            deleted_node1 ->binnum =y;
            deleted_node2 ->binnum =x;

            VerInsertNode(rootp1,deleted_node2);
            VerInsertNode(rootp2,deleted_node1);

            /*update curcap after swaps*/
            for (i=0;i<vnumbin ; i++){
```

```
                    if (vbinload[i]>0)
                        count++;
                    vcurcap[i]= vbinsize-vbinload[i];
                }
                if (count == lowerbound)
                    done=1;
                if (done){
                    printf("\n");
                    printf(" Done! vswap11. \n");
                    printf("\n");
                }

                return TRUE;

            }
        }
        q++;
        current_node2=current_node2->link;
    }
    j++;
    current_node1=current_node1->link;
}

return FALSE;
}


/*swapping items between vertical bins*/
int vswap10(Node ** rootp1, Node **rootp2, int x, int y, double vweight[])
{
    int i=0;
    int j=0;
    int count=0;
    double deltaf=0.0;

    Node * deleted_node1;
    Node * current_node1;
    Node * current_node2;
    Node * temp_node;

    current_node1=*rootp1;
    current_node2=*rootp2;

    while (current_node1!=NULL){

        if ( vcurcap[y] >= current_node1->height){
```

```
    deltaf=SQ(1.0*(vbinload[x]*vweight[x]-(current_node1->height)*vweight[x]))
          +SQ(1.0*(vbinload[y]*vweight[y]+(current_node1->height)*vweight[x]))
          -SQ(1.0*(vbinload[x])*vweight[x])-SQ(1.0*(vbinload[y])*vweight[y]);

    if ((deltaf >=0) ){

        vbinload[x]=vbinload[x]-(current_node1->height);
        vbinload[y]=vbinload[y]+(current_node1->height);

        deleted_node1= DeleteNode(rootp1,j);
        temp_node=hbin_root[deleted_node1->stripnum];
        for (j=0;(j<hnitembin[(deleted_node1->stripnum)]);j++){
            temp_node->binnum=y;
            if (temp_node->link != NULL)
            temp_node=temp_node->link;
        }

        deleted_node1 ->binnum =y;
        VerInsertNode(rootp2,deleted_node1);

        /*update number of items in bins*/
        vnitembin[y]++;
        vnitembin[x]--;

        /*update curcap after swaps*/

            for (i=0;i<vnumbin ; i++){
                if (vbinload[i]>0)
                    count++;
                vcurcap[i]= vbinsize-vbinload[i];
            }
            if (count == lowerbound)
                done=1;
            if (done){
                printf("\n");
                printf(" Done! vswap10. \n");
                printf("\n");
            }

        return TRUE;

    }
}
```

```
        j++;
        current_node1=current_node1->link;
    }

return FALSE;
}

/*swapping items between horizontal levels*/
int hswap22(Node ** rootp1, Node **rootp2, Node * temp_node1, Node * temp_node2,
int x, int y, int xbin, int ybin, double hweight[])
{

    double  deltaf;
    int m=0;
    int n=0;
    int j=0;
    int k=0;
    int q=0;
    int r=0;
    int i;
    int h1=0;
    int h2=0;
    int h3=0;
    int h4=0;

    Node * deleted_node1;
    Node * deleted_node2;
    Node * deleted_node3;
    Node * deleted_node4;
    Node * current_node1;
    Node * current_node2;
    Node * current_node3;
    Node * current_node4;

    current_node1=*rootp1;

    for(m=0;m<(nitembin[x]-1);m++){

        k=j+1;
        current_node2=current_node1->link;

        while (current_node2 != NULL){

            q=0;
            current_node3=*rootp2;
```

```
for(n=0;n<(nitembin[y]-1);n++){

    r=q+1;
    current_node4=current_node3->link;

    while (current_node4!=NULL){

        if (  (vcurcap[ybin] >= ((current_node1->height)-(*rootp2)->height))
            &&(vcurcap[xbin] >= ((current_node3->height)-(*rootp1)->height))
            &&(hcurcap[y] >= (current_node1->width+current_node2->width -
                current_node3->width-current_node4->width))
           &&(hcurcap[x] >= (current_node3->width+current_node4->width -
                current_node1->width-current_node2->width)) ){

            deltaf= SQ(1.0*(hbinload[x]*hweight[x]-(current_node1-
                >width)*hweight[x]-(current_node2->width)*hweight[x]+
                (current_node3->width)*hweight[y]+(current_node4->width)
                *hweight[y]))+SQ(1.0*(hbinload[y]*hweight[y]-(current_node3
                ->width)*hweight[y]-(current_node4->width)*hweight[y]+
                (current_node1->width)*hweight[x]+(current_node2-
                >width)*hweight[x]))-SQ(1.0*(hbinload[x]*hweight[x]))-
                SQ(1.0*(hbinload[y]*hweight[y]));

        if (deltaf>=0) {
                hbinload[x]=hbinload[x]-(current_node1->width)-(current_node2-
                    >width)+(current_node3->width)+(current_node4->width);
                hbinload[y]=hbinload[y]-(current_node3->width)-(current_node4-
                    >width)+(current_node1->width)+(current_node2->width);

                h1=(*rootp1)->height;
                h2=(*rootp2)->height;

                deleted_node2= DeleteNode(rootp1,k);
                deleted_node4= DeleteNode(rootp2,r);
                deleted_node1= DeleteNode(rootp1,j);
                deleted_node3= DeleteNode(rootp2,q);

                deleted_node1->stripnum=y;
                deleted_node1->binnum=ybin;

                deleted_node2->stripnum=y;
                deleted_node2->binnum=ybin;

                deleted_node3->stripnum=x;
                deleted_node3->binnum=xbin;
```

```
                    deleted_node4->stripnum=x;
                    deleted_node4->binnum=xbin;

                    HorInsertNode(rootp1,deleted_node3);
                    HorInsertNode(rootp1,deleted_node4);
                    temp_node1 ->height = (*rootp1)->height;
                    temp_node1 ->width  = (*rootp1)->width;

                    HorInsertNode(rootp2,deleted_node1);
                    HorInsertNode(rootp2,deleted_node2);
                    temp_node2 ->height = (*rootp2)->height;
                    temp_node2 ->width  = (*rootp2)->width;

                    h3=(*rootp1)->height;
                    h4=(*rootp2)->height;
                    vbinload[xbin]=vbinload[xbin]+ h3 - h1;
                    vbinload[ybin]=vbinload[ybin]+ h4 - h2;

                    /*update curcap after swaps*/
                    done = 1;
                    for (i=0;i<hnumbin ; i++){
                        hcurcap[i]= hbinsize-hbinload[i];
                    }

                    for (i=0;i<vnumbin ; i++){
                        vcurcap[i]= vbinsize-vbinload[i];
                        if (vcurcap[i]-bound < 0)
                        done = 0;
                    }

                    done=0 ;   /*to remove*/
                    return TRUE;

                }

            }
            r++;
            current_node4=current_node4->link;
        }
        q++;
        current_node3=current_node3->link;
    }
    k++;
    current_node2=current_node2->link;
}
j++;
```

```
        current_node1=current_node1->link;
    }

return FALSE;
}


/*swapping items between horizontal levels*/
int hswap12(Node ** rootp1, Node **rootp2, Node * temp_node1, Node * temp_node2,
int x, int y,  int xbin, int ybin, double hweight[])
{
    double deltaf;
    double deltaf1=0.0;

    int deltaf2=0;
    int deltaf21=0;
    int deltaf22=0;
    int deltaf23=0;

    int i;
    int n=0;
    int j=0;
    int q=0;
    int r=0;
    int h1=0;
    int h2=0;
    int h3=0;
    int h4=0;
    int hnext1=0;
    int hnext2=0;

    Node * deleted_node1;
    Node * deleted_node2;
    Node * deleted_node3;
    Node * current_node1;
    Node * current_node2;
    Node * current_node3;
    Node * current_node4;
    Node * current_node5;

    if ((*rootp1)!=NULL)
        h1=(*rootp1)->height;
    if ((*rootp2)!=NULL)
        h2=(*rootp2)->height;

    current_node1=*rootp1;
```

```
current_node4=*rootp1;
current_node5=*rootp2;

if (*rootp1!=NULL){
    current_node4=current_node4->link;

    if (current_node4!=NULL)
        hnext1=current_node4->height;
}

if (*rootp2!=NULL){
    current_node5=current_node5->link;

    if (current_node5!=NULL)
        hnext2=current_node5->height;

}

while (current_node1!=NULL){

    q=0;
    current_node2=*rootp2;

    for(n=0;n<(hnitembin[y]-1);n++){

            r=q+1;
            current_node3=current_node2->link;

            while (current_node3!=NULL){


                if (  (vcurcap[ybin] >= ((current_node1->height)-h2))
                    &&(vcurcap[xbin] >= ((current_node2->height)-h1))
                    &&(hcurcap[y] >= (current_node1->width - current_node2->width –
                        current_node3->width))
                    &&(hcurcap[x] >= (current_node2->width + current_node3->width –
                        current_node1->width)) ){

                    deltaf1= SQ(1.0*(hbinload[x]*hweight[x]-(current_node1->width)*
                        hweight[x]+(current_node2->width)*hweight[y]+(current_node3-
                        >width)*hweight[y]))+ SQ(1.0*(hbinload[y]*hweight[y]-
                        (current_node2->width)*hweight[y]-(current_node3-
                        >width)*hweight[y]+(current_node1->width)*hweight[x]))-
                        SQ(1.0*(hbinload[x]*hweight[x]))-
                        SQ(1.0*(hbinload[y]*hweight[y]));
```

```
if (j==0){
    if(current_node2->height < hnext1)
    deltaf21=((h1-hnext1))*(hbinsize);
else
    deltaf21=(h1 - current_node2->height )*(hbinsize);
}
else {

    if(current_node2->height < h1)
        deltaf21=0;
    else
        deltaf21=(h1 - current_node2->height )*(hbinsize);

}

if (q==0){
    if (current_node1->height < hnext2)
        deltaf22=((h2-hnext2))*(hbinsize);
    else
        deltaf22=(h2 - current_node1->height )*(hbinsize);
}
else {
    if(current_node1->height < h2)
        deltaf22=0;
    else
        deltaf22=(h2 - current_node1->height )*(hbinsize);
}

deltaf2=deltaf21+deltaf22;
deltaf=(double)deltaf1+(double)deltaf2;

if (deltaf>=0) {
    hbinload[x]=hbinload[x]-(current_node1->width)+(current_node2-
        >width)+(current_node3->width);
    hbinload[y]=hbinload[y]-(current_node2->width)-(current_node3-
        >width)+(current_node1->width);

    deleted_node3= DeleteNode(rootp2,r);
    deleted_node1= DeleteNode(rootp1,j);
    deleted_node2= DeleteNode(rootp2,q);

    deleted_node1->stripnum=y;
    deleted_node1->binnum=ybin;

    deleted_node2->stripnum=x;
```

```
deleted_node2->binnum=xbin;

deleted_node3->stripnum=x;
deleted_node3->binnum=xbin;

HorInsertNode(rootp1,deleted_node2);
HorInsertNode(rootp1,deleted_node3);

if((*rootp1)!=NULL){
    temp_node1 ->height = (*rootp1)->height;
    temp_node1 ->width  = (*rootp1)->width;
}
else{
    temp_node1 ->height = 0;
    temp_node1 ->width  = 0;
}

HorInsertNode(rootp2,deleted_node1);

if((*rootp2)!=NULL){
    temp_node2 ->height = (*rootp2)->height;
    temp_node2 ->width  = (*rootp2)->width;
}
else{
    temp_node2 ->height = 0;
    temp_node2 ->width  = 0;
}
if ((*rootp1)!=NULL)
    h3=(*rootp1)->height;

if ((*rootp2)!=NULL)
    h4=(*rootp2)->height;

vbinload[xbin]=vbinload[xbin]+ h3 - h1;
vbinload[ybin]=vbinload[ybin]+ h4 - h2;

/*update number of items in bins*/
hnitembin[y]--;
hnitembin[x]++;

/*update curcap after swaps*/
done = 1;
for (i=0;i<hnumbin ; i++){
    hcurcap[i]= hbinsize-hbinload[i];
    if (hcurcap[i]-bound < 0)
        done = 0;
```

```
                        }

                        for (i=0;i<vnumbin ; i++){
                            vcurcap[i]= vbinsize-vbinload[i];
                            if (vcurcap[i]-bound < 0)
                            done = 0;
                        }

                        return TRUE;

                    }

                }
                r++;
                current_node3=current_node3->link;
            }
        q++;
        current_node2=current_node2->link;
    }
    j++;
    current_node1=current_node1->link;
}

return FALSE;
}


/*swapping items between horizontal levels*/
int hswap11(Node ** rootp1, Node **rootp2, Node * temp_node1, Node * temp_node2,
int x, int y,  int xbin, int ybin, double hweight[])
{
    double deltaf=0.0;
    double deltaf1=0.0;
    double deltaf3=0.0;

    int deltaf2=0;
    int deltaf21=0;
    int deltaf22=0;

    int i=0;
    int j=0;
    int q=0;
    int h1=0;
    int h2=0;
    int h3=0;
    int h4=0;
```

```
int hnext1=0;
int hnext2=0;

Node * deleted_node1;
Node * deleted_node2;
Node * current_node1;
Node * current_node2;
Node * current_node3;
Node * current_node4;

current_node1=*rootp1;
current_node2=*rootp2;
current_node3=*rootp1;
current_node4=*rootp2;


if ((*rootp1)!=NULL)
    h1=(*rootp1)->height;

if ((*rootp2)!=NULL)
    h2=(*rootp2)->height;

if (*rootp1!=NULL){
    current_node3=current_node3->link;

    if (current_node3!=NULL)
        hnext1=current_node3->height;
}

if (*rootp2!=NULL){
    current_node4=current_node4->link;

    if (current_node4!=NULL)
        hnext2=current_node4->height;

}

while (current_node1!=NULL){

    q=0;
    current_node2=*rootp2;

    while (current_node2!=NULL){

        if( (vcurcap[ybin] >= ((current_node1->height)-h2))
            && (vcurcap[xbin] >= ((current_node2->height)-h1))
```

```
&& (hcurcap[y] >= (current_node1->width - current_node2->width))
&& (hcurcap[x] >= (current_node2->width - current_node1->width)) ){

deltaf1=SQ(1.0*(hbinload[x]*hweight[x]-(current_node1->width)*
    hweight[x]+ (current_node2->width)*hweight[y]))+SQ(1.0*(hbinload[y]
    *hweight[y]- (current_node2->width)*hweight[y]+(current_node1-
    >width)*hweight[x]))-SQ(1.0*(hbinload[x]*hweight[x]))-
    SQ(1.0*(hbinload[y])*hweight[y]);

if (j==0){
    if(current_node2->height < hnext1)
        deltaf21=((h1-hnext1))*(hbinsize);
    else
        deltaf21=(h1 - current_node2->height )*(hbinsize);
}
else {

    if(current_node2->height < h1)
        deltaf21=0;
    else
        deltaf21=(h1 - current_node2->height )*(hbinsize);

}

if (q==0){
    if (current_node1->height < hnext2)
        deltaf22=((h2-hnext2))*(hbinsize);
    else
        deltaf22=(h2 - current_node1->height )*(hbinsize);
}
else {
    if(current_node1->height < h2)
        deltaf22=0;
    else
        deltaf22=(h2 - current_node1->height )*(hbinsize);
}

deltaf2=deltaf21+deltaf22;

deltaf3=SQ(1.0*(hbinarea[x]-(current_node1->area)+(current_node2-
    >area))) +SQ(1.0*(hbinarea[y]+(current_node1->area)-(current_node2-
    >area))) -SQ(1.0*(hbinarea[x]))-SQ(1.0*(hbinarea[y]));

if (deltaf3 >=0)
    deltaf=(double)deltaf1+(double)deltaf2 + pow((double)deltaf3, 0.1);
else
```

```
    deltaf=(double)deltaf1+(double)deltaf2 - pow((-(double)deltaf3), 0.1);

if (deltaf>=0 ) {

    hbinload[x]=hbinload[x]-(current_node1->width) +(current_node2-
        >width);
    hbinload[y]=hbinload[y]-(current_node2->width) +(current_node1-
        >width);

    hbinarea[x]=hbinarea[x]-(current_node1->area)+(current_node2->area);
    hbinarea[y]=hbinarea[y]+(current_node1->area)-(current_node2->area);

    deleted_node1= DeleteNode(rootp1,j);
    deleted_node2= DeleteNode(rootp2,q);

    deleted_node1->stripnum=y;
    deleted_node1->binnum=ybin;

    deleted_node2->stripnum=x;
    deleted_node2->binnum=xbin;

    HorInsertNode(rootp1,deleted_node2);

    if((*rootp1)!=NULL){
        temp_node1 ->height = (*rootp1)->height;
        temp_node1 ->width  = (*rootp1)->width;
    }
    else{
    temp_node1 ->height = 0;
    temp_node1 ->width  = 0;
    }

    HorInsertNode(rootp2,deleted_node1);

    if((*rootp2)!=NULL){
        temp_node2 ->height = (*rootp2)->height;
        temp_node2 ->width  = (*rootp2)->width;
    }
    else{
    temp_node2 ->height = 0;
    temp_node2 ->width  = 0;
    }

    if ((*rootp1)!=NULL)
        h3=(*rootp1)->height;
```

```
                    if ((*rootp2)!=NULL)
                        h4=(*rootp2)->height;

                    vbinload[xbin]=vbinload[xbin]+ h3 - h1;
                    vbinload[ybin]=vbinload[ybin]+ h4 - h2;

                    /*update curcap after swaps*/
                    done = 1;
                    for (i=0;i<hnumbin ; i++){
                        hcurcap[i]= hbinsize-hbinload[i];
                        if (hcurcap[i]-bound < 0)
                            done = 0;
                    }

                    for (i=0;i<vnumbin ; i++){
                        vcurcap[i]= vbinsize-vbinload[i];
                        if (vcurcap[i]-bound < 0)
                            done = 0;
                    }
                    return TRUE;

                }

            }
            q++;
            current_node2=current_node2->link;
        }
        j++;
        current_node1=current_node1->link;
    }

return FALSE;
}

/*swapping items between horizontal levels*/
int hswap10(Node ** rootp1, Node **rootp2, Node * temp_node1, Node * temp_node2,
int x, int y,
            int xbin, int ybin, double hweight[])
{
    int i=0;
    int j=0;
    double deltaf=0.0;
    double deltaf1=0.0;
    int deltaf2=0;
    int deltaf21=0;
    int deltaf22=0;
```

```
double deltaf3=0.0;
int h1=0;
int h2=0;
int h3=0;
int h4=0;
int hnext=0;

if ((*rootp1)!=NULL)
    h1=(*rootp1)->height;


if ((*rootp2)!=NULL)
    h2=(*rootp2)->height;

Node * deleted_node1;
Node * current_node1;
Node * current_node2;
Node * current_node3;

current_node1=*rootp1;
current_node2=*rootp2;
current_node3=*rootp1;

if (*rootp1!=NULL){
    current_node3=current_node3->link;

    if (current_node3!=NULL)
        hnext=current_node3->height;

}

while (current_node1!=NULL){

    if ((vcurcap[ybin] >= ((current_node1->height)-h2))&&
         (hcurcap[y]>=(current_node1->width)) ){


        deltaf1=SQ(1.0*(hbinload[x]-(current_node1->width))*hweight[x])+
            SQ(1.0*(hbinload[y]*hweight[y]+(current_node1->width)*hweight[x]))-
            SQ(1.0*(hbinload[x])*hweight[x])-SQ(1.0*(hbinload[y])*hweight[y]);

        if (j==0)
            deltaf21=((h1-hnext))*(hbinsize);

        if (current_node1->height >h2)
            deltaf22=(current_node1->height - h2)*hbinsize;
```

```
deltaf2=deltaf21-deltaf22;

deltaf3=SQ(1.0*(hbinarea[x]-(current_node1->area)))
       +SQ(1.0*(hbinarea[y]+(current_node1->area)))
       -SQ(1.0*(hbinarea[x]))-SQ(1.0*(hbinarea[y]));

if (deltaf3 >=0)
    deltaf=(double)deltaf1+(double)deltaf2 + pow((double)deltaf3, 0.1);
else
    deltaf=(double)deltaf1+(double)deltaf2 - pow((-(double)deltaf3), 0.1);

if ( (deltaf>=0)  ){

    hbinload[x]=hbinload[x]-(current_node1->width);
    hbinload[y]=hbinload[y]+(current_node1->width);

    hbinarea[x]=hbinarea[x]-(current_node1->area);
    hbinarea[y]=hbinarea[y]+(current_node1->area);

    deleted_node1= DeleteNode(rootp1,j);
    deleted_node1->stripnum=y;
    deleted_node1->binnum=ybin;

    HorInsertNode(rootp2,deleted_node1);

    if((*rootp1)!=NULL){
        temp_node1 ->height = (*rootp1)->height;
        temp_node1 ->width  = (*rootp1)->width;
    }
    else{
        temp_node1 ->height = 0;
        temp_node1 ->width  = 0;
    }

    if((*rootp2)!=NULL){
        temp_node2 ->height = (*rootp2)->height;
        temp_node2 ->width  = (*rootp2)->width;}
    else{
        temp_node2 ->height = 0;
        temp_node2 ->width  = 0;
    }

    if ((*rootp1)!=NULL)
        h3=(*rootp1)->height;
```

```
                if ((*rootp2)!=NULL)
                    h4=(*rootp2)->height;

                vbinload[xbin]=vbinload[xbin]+ h3 - h1;
                vbinload[ybin]=vbinload[ybin]+ h4 - h2;

                /*update number of items in bins*/
                hnitembin[y]++;
                hnitembin[x]--;

                /*update curcap after swaps*/
                done = 1;
                for (i=0;i<hnumbin ; i++){
                    hcurcap[i]= hbinsize-hbinload[i];
                }

                for (i=0;i<vnumbin ; i++){
                    vcurcap[i]= vbinsize-vbinload[i];
                    if (vcurcap[i]-bound < 0)
                        done = 0;
                }

                return TRUE;

            }
        }

        j++;
        current_node1=current_node1->link;
    }

return FALSE;
}


void UpdateVposition(Node **rootp)
{
    int count=0;
    Node * current_node;

    current_node=*rootp;

    while (current_node!=NULL){
        current_node->vposition=count;
        count++;
        current_node=current_node->link;
```

```
    }

}

void UpdateHposition(Node **rootp)
{
    int count=0;
    Node * current_node;

    current_node=*rootp;

    while (current_node!=NULL){
        current_node->hposition=count;
        count++;
        current_node=current_node->link;
    }

}


int CompBinItem(Node * root)
{
    int count=0;
    if (root != NULL) {
        while (root != NULL) {
            count++;
            root = root->link;
        }
    } else {
        printf("This bin has no items!\n");
    }
    return count;
}


int CompVbinLoad(Node * root)
{
    int load=0;
    if (root != NULL) {
        while (root != NULL) {
            load = load+(root->height);
            root = root->link;
        }
    } else {
        printf("This vertical bin is empty!\n");
    }
```

```
    return load;
}


int CompHbinLoad(Node * root)
{
    int load=0;
    if (root != NULL) {
        while (root != NULL) {
            load = load+(root->width);
            root = root->link;
        }
    } else {
        printf(" This horisontal bin is empty!\n");
    }
    return load;
}



void PrintFileNode(Node * item)
{
    fprintf(op, "%8d %8d %8d %8d %8d %8d %8d %8d %8d \n", item->height,item-
>width,item->stripnum, item->hposition, item->binnum,item->vposition,item-
>floornum,item->ceilnum,item->area);

}


void PrintFile(Node * root)
{
    if (root != NULL) {
        while (root != NULL) {
            PrintFileNode(root);
            root = root->link;
        }
    } else {
        fprintf(op, " No nodes have been entered yet!\n");
    }
}
```

```
void PrintNode(Node * item)
{
    printf("%8d %8d %8d %8d %8d %8d %8d %8d %8d \n", item->height,item-
>width,item->stripnum, item->hposition,item->binnum,item->vposition,item-
>floornum,item->ceilnum,item->area);

}


void PrintAllNode(Node * root)
{
    if (root != NULL) {
        while (root != NULL) {
            PrintNode(root);
            root = root->link;
        }
    } else {
        printf("No nodes have been entered yet!\n");
    }
}


Node * CreatNode(int high, int wide)
{
    struct NODE * temp = (struct NODE*) malloc(sizeof(struct NODE));

    temp->height=high;
    temp->width=wide;
    temp->stripnum=0;
    temp->binnum=0;
    temp->hposition=0;
    temp->vposition=0;
    temp->ceilnum=0;
    temp->floornum=0;
    temp->area=0;

    return temp;
}



Node * AddNode(int high, int wide, int num1, int num2, Node * bin )
{
    struct NODE * temp = (struct NODE*) malloc(sizeof(struct NODE));
```

```
    temp->height = high;
    temp->width  = wide;
    temp->stripnum=num1;
    temp->binnum=num2;
    temp->vposition=0;
    temp->ceilnum=0;
    temp->floornum=0;
    temp->area=0;
    temp->link = bin;
    bin = temp;
    return bin;
}



int HorInsertNode( register Node **linkp, Node * inserted_node)
{
    register Node   *current;
    while( ( current = *linkp ) != NULL &&
        current->height >=inserted_node->height)
        linkp = &current->link;

    inserted_node->link = current;
    *linkp = inserted_node;

    return TRUE;
}



int VerInsertNode( register Node **linkp, Node * inserted_node)
{
    register Node   *current;
    while( ( current = *linkp ) != NULL &&
        hbinload[current->stripnum] >hbinload[inserted_node->stripnum])
        linkp = &current->link;

    inserted_node->link = current;
    *linkp = inserted_node;

    return TRUE;
}
```

```
Node * DeleteNode( Node **linkp, int item_num )
{
    Node *current;
    Node *previous;
    Node *delnode;
    int count =0;

    current = *linkp;
    previous = NULL;

    while( current != NULL && count < item_num ){
        count++;
        previous = current;
        current = current->link;
    }

    delnode = current;

    if(previous==NULL)
        *linkp=current->link;
    else
        previous->link=delnode->link;

    if(current!=NULL)
        current = current->link;
    else
        current->link=NULL;

    return  delnode;
}


void removebin (Node **linkp)
{
    Node *current;
    Node *previous;

    current = *linkp;
    previous = NULL;

    while( current->height != 0 ){
        previous = current;
        current = current->link;
    }

    if(previous==NULL)
```

```
        *linkp=current->link;
    else
        previous->link=current->link;

    if(current!=NULL)
        current = current->link;
    else
        current->link=NULL;

}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//  2bp_o_g.h  is the header file for WA2BPOG.cpp
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
typedef  struct NODE   {
    struct NODE   *link;
    int height;
    int width;
    int stripnum;
    int binnum;
    int hposition;
    int vposition;
    int floornum;
    int ceilnum;
    int resinum;
    int area;
} Node;

void sort(unsigned long n, double arr[]);
#define SQ(a) (a*a)
#define  FALSE  0
#define TRUE 1
#define MAX_NUMBER_BIN 100
#define MAX_NUMBER_ITEM 150
#define MAX_NUMBER_HOR  20
#define MAX_NUMBER_VER 10
```

Appendix D

Two-Dimensional Bin Packing Problem Code
(Guillotine Cuts, Non-Oriented Items)

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//*     WA2BPRG.cpp is the program for solving the two-dimensional bin packing
//problem  //with guillotine cuts and non_oriented item.
//*      It calls functions from WA2BPOG.cpp
//*      It is to be complied with the following files (ANSI version) from Press et al.(2002)
//          1)     nrutil.cpp
//          2)     nrutil.h
//          3)     nr.h
//*   The header file 2bp_r_g.h is appended at the end of the program.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#include <io.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "nrutil.h"
#include "nr.h"
#include <sys\timeb.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include "2bp_r_g.h"
#include <string.h>

Node * bin_root[MAX_NUMBER_BIN];
Node * vbin_root[MAX_NUMBER_BIN];
Node * hbin_root[MAX_NUMBER_BIN];

Node * obj_size;
Node * strip_size;

FILE * fp;
FILE * op;

int binload[MAX_NUMBER_BIN];
int vbinload[MAX_NUMBER_BIN];
int hbinload[MAX_NUMBER_BIN];
int vbinarea[MAX_NUMBER_BIN];
```

```
int vareacap[MAX_NUMBER_BIN];
int hbinarea[MAX_NUMBER_BIN];
int hareacap[MAX_NUMBER_BIN];
int hmaxht[MAX_NUMBER_BIN];
int hstripsize[MAX_NUMBER_BIN];
int curcap[MAX_NUMBER_BIN];
int vcurcap[MAX_NUMBER_BIN];
int hcurcap[MAX_NUMBER_BIN];
int nitembin[MAX_NUMBER_BIN];
int vnitembin[MAX_NUMBER_BIN];
int hnitembin[MAX_NUMBER_BIN];
int numcell[MAX_NUMBER_BIN];

int ceilheight [MAX_NUMBER_BIN][MAX_NUMBER_HOR];
int ceilwidth  [MAX_NUMBER_BIN][MAX_NUMBER_HOR];
int ceilcap    [MAX_NUMBER_BIN][MAX_NUMBER_HOR];
int floorheight[MAX_NUMBER_BIN][MAX_NUMBER_HOR];
int floorwidth [MAX_NUMBER_BIN][MAX_NUMBER_HOR];
int floorcap   [MAX_NUMBER_BIN][MAX_NUMBER_HOR];

int binsize=0;
int bound=0;
int vbinsize=0;
int hbinsize=0;
int binarea=0;
int vnumbin=32;
int maxhnumbin=100;
int hnumbin=0;

double comptime=0.0;
double totaltime=0.0;
int done =0;

double TimeElapsed(struct _timeb *begin, struct _timeb *end);
int irbit1(unsigned long *iseed);
void PrintNode(Node * item);
void PrintAllNode(Node * root);
void PrintFileNode(Node * item);
void PrintFile(Node * root);
void removebin (Node **linkp);
void UpdateHposition(Node **rootp);
void UpdateVposition(Node **rootp);

Node * AddNode(int high, int wide, int num1, int num2, Node * bin);
Node * CreatNode(int high, int wide);
Node * DeleteNode( Node **linkp, int item_num );
```

```
int HorInsertNode( register Node **linkp, Node * inserted_node);
int VerInsertNode( register Node **linkp, Node * inserted_node);

int CompVbinLoad(Node * root);
int CompHbinLoad(Node * root);
int CompBinItem(Node * root);

int vswap10(Node ** rootp1, Node **rootp2, int x, int y, double vweight[]);
int vswap11(Node ** rootp1, Node **rootp2, int x, int y, double vweight[]);
int vswap12(Node ** rootp1, Node **rootp2, int x, int y, double vweight[]);
int vswap22(Node ** rootp1, Node **rootp2, int x, int y, double vweight[]);

int hswap10(Node ** rootp1, Node **rootp2, Node * temp_node1, Node * temp_node2,
        int x, int y, int xbin, int ybin, double hweight[]);
int hswap11(Node ** rootp1, Node **rootp2, Node * temp_node1, Node * temp_node2,
        int x, int y, int xbin, int ybin, double hweight[]);
int hswap12(Node ** rootp1, Node **rootp2, Node * temp_node1, Node * temp_node2,
        int x, int y, int xbin, int ybin, double hweight[]);
int hswap22(Node ** rootp1, Node **rootp2, Node * temp_node1, Node * temp_node2,
        int x, int y, int xbin, int ybin, double hweight[]);
int rgpack(Node ** rootp1, Node **rootp2, int s, int t, double hweight[]);

int tightpack(Node **rootp);

int lowerbound=4;

int
main(void)
{
    struct _timeb time1, time2;
    unsigned long iseed;
    Node * deleted_node;
    Node * deleted_node1;
    Node * deleted_node2;
    Node * inserted_node;
    Node * current_node;
    Node * temp_node;
    Node * temp_node1;
    Node * temp_node2;
    int i,j,k,idec,p,q,m,n,s,loop,preht;
    int numbitem=0;
    int improved=0;
    int numtype=0;
    int temp1=0;
    int numstrip=0;
    int empty=0;
```

```
int data=0;
int count=0;
int count1=0;
int binheight=0;
int culwidth=0;
double hweight [MAX_NUMBER_BIN];
double vweight [MAX_NUMBER_BIN];
double bweight [MAX_NUMBER_BIN];
int height[MAX_NUMBER_ITEM];
int width[MAX_NUMBER_ITEM];
double T = 1.0;
double Tred =0.95;
float temp=0.0;
char str[10];
done=0;
iseed=111;

if((op=fopen("out.txt", "w")) == NULL) {
    printf("Cannot open file.\n");
    exit(1);
}

/*Input file format – instances from Berkey and Wang (1987)*/
if((fp=fopen( "C3_20_5.txt", "r")) == NULL)
{       printf("Cannot open file.\n");
    exit(1);
}

fscanf(fp, "%d %s %s", &data, &str, &str);
fscanf(fp, "%d %s %s %s", &numbitem, &str, &str, &str);
fscanf(fp, "%d %d %s %s %s %s %s %s", &data, &data, &str,
          &str, &str, &str, &str, &str);
fscanf(fp, "%d %d %s", &vbinsize, &hbinsize, &str);

for(i =0; i <numbitem;i++){
    if (i==0)
        fscanf(fp, "%d  %d %s", &height[i], &width[i], &str);
    else
        fscanf(fp, "%d  %d ", &height[i], &width[i]);

}

bound=0;
hbinsize=hbinsize+bound;
vbinsize=vbinsize+bound;
binarea=vbinsize*hbinsize;
```

```
_ftime(&time1);

    for (loop=0;loop<10;loop++){

    /* initialisation*/
    obj_size=NULL;
    strip_size=NULL;
    deleted_node  =NULL;
    deleted_node1 =NULL;
    deleted_node2 =NULL;
    inserted_node =NULL;
    current_node  =NULL;
    temp_node=NULL;
    temp_node1=NULL;
    temp_node2=NULL;

    for (i=0;i<vnumbin;i++)
        vbin_root[i] = NULL;
    for (i=0;i<maxhnumbin;i++)
        hbin_root[i] = NULL;

    for( i=0;i<vnumbin;i++)
        vcurcap[i]=vbinsize;
    for( i=0;i<maxhnumbin;i++)
        hcurcap[i]=hbinsize;

    for( i=0;i<vnumbin;i++){
        vbinload[i]=0;
        vnitembin[i]=0;
    }
    for( i=0;i<maxhnumbin;i++){
        hbinload[i]=0;
        hnitembin[i]=0;
    }
    for (i=0;i< MAX_NUMBER_BIN;i++){
        for(j=0;j<MAX_NUMBER_HOR; j++){
            ceilheight [i][j]=0;
            ceilwidth  [i][j]=0;
            ceilcap  [i][j]=0;
            floorheight[i][j]=0;
            floorwidth [i][j]=0;
            floorcap   [i][j]=0;

        }
    }
```

```
for (i=0;i<MAX_NUMBER_BIN;i++)
    numcell[i]=0;

T=1.0;
Tred=0.95;
numstrip=0;
count=0;

    /*read data file*/
    for (i=0;i<numbitem;i++){
        current_node=CreatNode(height[i], width[i]);
        current_node->area=height[i]*width[i];

        HorInsertNode(&obj_size, current_node);
    }

    idec=irbit1(&iseed);

    /* first-fit decreasing for horizontal bins*/
    for (i=0;i<numbitem;i++){
        idec=irbit1(&iseed);
        if (i== (numbitem-1)) idec=0;

        deleted_node= DeleteNode(&obj_size,idec);
        for (j=0;j<maxhnumbin;j++)
            if (hcurcap[j] >= deleted_node->width){
                hcurcap[j]=hcurcap[j]- (deleted_node->width);
                deleted_node->stripnum=j;
                HorInsertNode(&hbin_root[j],deleted_node);

                break;
            }
    }

    /*linked list for vertical bins*/
    for (i=(maxhnumbin-1);i>=0;i--){

        if (hbin_root[i] !=NULL){
            numstrip++;
            strip_size = AddNode(hbin_root[i]->height, hbin_root[i]->width,
            hbin_root[i]->stripnum, hbin_root[i]->binnum, strip_size);
        }
    }

    hnumbin=numstrip;
```

```
/*compute the total load and number of items in each bin*/
for (i=0; i<hnumbin;i++) {
    hbinload[i]=CompHbinLoad(hbin_root[i]);
    hnitembin[i]=CompBinItem(hbin_root[i]);
}



/* first-fit dereasing for vertical bins*/
for (i=0;i<hnumbin;i++){

    deleted_node= DeleteNode(&strip_size,0);
    for (j=0;j<vnumbin;j++)
        if (vcurcap[j] >= deleted_node->height){
            vcurcap[j]=vcurcap[j]- (deleted_node->height);

            temp_node=hbin_root[deleted_node->stripnum];
            for (k=0;(k<hnitembin[(deleted_node->stripnum)]);k++){
                temp_node->binnum=j;
                if (temp_node->link != NULL)
                temp_node=temp_node->link;
            }
            temp_node=NULL;

            deleted_node ->binnum=j;
            VerInsertNode(&vbin_root[j],deleted_node);
            break;
        }
}

/*compute the total load and number of items in each bin*/
for (i=0; i<vnumbin;i++) {
    vbinload[i]=CompVbinLoad(vbin_root[i]);
    vnitembin[i]=CompBinItem(vbin_root[i]);
}


/*compute utilised areas in each horizontal bin*/

for (i=0;i<hnumbin;i++)
        hbinarea[i]=0;

for (i=0; i<hnumbin;i++){
    temp_node=hbin_root[i];
    for (k=0;(k<hnitembin[i]);k++){
```

```
        hbinarea[i]=hbinarea[i]+temp_node->area;
        temp_node=temp_node->link;
    }
}

for (m=0;m<20;m++){


    /* update weights, set K = 0.05*/
    for (i=0;i<hnumbin ; i++){
        hweight[i]= pow((1.0+((double)hcurcap[i]/(double)hbinsize)*0.05), T);
    }

    /*swap item amongst levels */
    for (i=0;i<vnumbin;i++) {
        improved=0;

        temp_node1=vbin_root[i];
        for(j=0; j<vnitembin[i];j++){

            for (p=0;p<vnumbin;p++){

                temp_node2=vbin_root[p];
                for(q=0;q<vnitembin[p];q++){

                    if  ((temp_node1->stripnum)!=(temp_node2->stripnum)){

                        improved = hswap10(&hbin_root[temp_node1->stripnum],
                            &hbin_root[temp_node2->stripnum], temp_node1,
                            temp_node2,(temp_node1->stripnum),
                            (temp_node2->stripnum),i, p, hweight);

                        if (hbin_root[temp_node1->stripnum]==NULL){
                            removebin(&vbin_root[i]);
                            vnitembin[i]--;
                            break;
                        }
                        if (done) break;

                        improved = hswap11(&hbin_root[temp_node1->stripnum],
                            &hbin_root[temp_node2->stripnum], temp_node1,
                             temp_node2,(temp_node1->stripnum),
                            (temp_node2->stripnum), i, p, hweight);
                        if (done) break;

                        improved = hswap12(&hbin_root[temp_node1->stripnum],
```

```
                        &hbin_root[temp_node2->stripnum], temp_node1,
                        temp_node2,(temp_node1->stripnum),
                        (temp_node2->stripnum), i, p, hweight);
                    if (done) break;

                    improved = hswap22(&hbin_root[temp_node1->stripnum],
                        &hbin_root[temp_node2->stripnum],  temp_node1,
                        temp_node2,(temp_node1->stripnum),
                        (temp_node2->stripnum), i, p, hweight);

                    if (done) break;

                    /*rotating items to achieve a tighter packing solution*/

                    improved = tightpack(&hbin_root[temp_node1->stripnum]);
                    improved = tightpack(&hbin_root[temp_node2->stripnum]);

                }

                temp_node2=temp_node2->link;

            }
            if (hbin_root[temp_node1->stripnum]==NULL) break;
            if (done) break;
        }
        if (done) break;
        if(temp_node1 !=NULL)
        temp_node1=temp_node1->link;
    }
    if (done) break;
}
if(done) break;
T=T*Tred;

}

/*packing vertical bins*/
    T=1.0;
    for (m=0;m<20;m++){

        /* update weights, set K = 0.05*/

        for (i=0;i<vnumbin ; i++)
                vweight[i]= pow((1.0+((double)vcurcap[i]/(double)vbinsize)*0.05), T);

        /*swapping items between vertical bins*/
```

```
        for (i=0;i<vnumbin;i++) {
                improved=0;

            for (p=0;p<vnumbin;p++){

                if  (p!=i){

                    improved = vswap10(&vbin_root[i], &vbin_root[p],i,p, vweight);
                    if (done) break;

                    improved = vswap11(&vbin_root[i], &vbin_root[p],i,p,vweight);
                    if (done) break;

                    improved = vswap12(&vbin_root[i], &vbin_root[p],i,p,vweight);
                    if (done) break;

                    improved = vswap22(&vbin_root[i], &vbin_root[p],i,p,vweight);
                    if (done) break;

                }

            }

            if (done) break;
        }
        if (done) break;
        T=T*Tred;
    }

count1=0;
for (i=0;i<vnumbin;i++){
    if (vbin_root[i]!=NULL)
        count1++;
}

/*initialise hposition, vposition, floornum, ceilnum for horizontal levels  after hswap*/

        for (i=0;i<vnumbin;i++){
            temp_node1=vbin_root[i];

            for (j=0;(j<vnitembin[i]);j++){
                s=temp_node1->stripnum;
                temp_node2=hbin_root[s];

                for (k=0;(k<hnitembin[s]);k++){
```

```
            temp_node2->binnum=i;
            temp_node2->hposition = k;
            temp_node2->vposition = j;
            temp_node2->floornum = k;
            temp_node2->ceilnum  =-1;
            temp_node2=temp_node2->link;
        }

        temp_node1->binnum=i;
        temp_node1->hposition=0;
        temp_node1->vposition = j;
        temp_node1->floornum = 0;
        temp_node1->ceilnum  =-1;
        temp_node1->area=temp_node1->height * temp_node1->width;
        temp_node1=temp_node1->link;
    }
}


/* determine space occupied by items, and residual areas in each level */
for (i=0;i<hnumbin;i++){

    if (hbin_root[i]!=NULL){

        numcell[i]=hnitembin[i];
        temp_node1=hbin_root[i];
        preht = 0;
        culwidth=0;

        for (j=0;(j<hnitembin[i]);j++){

            if (j==0){
                culwidth=temp_node1->width;
                /* if the horizontal strip is on the top of veritcal bin */
                if (temp_node1->vposition == (vnitembin[temp_node1->binnum]-1)){
                    ceilheight [i][0]=temp_node1->height;
                    preht=temp_node1->height;

                   /* dimensions of the slot directly above the first item of the above
                    strip*/
                    ceilheight [i][numcell[i]]=vcurcap[temp_node1->binnum];
                    ceilwidth  [i][numcell[i]]=hbinsize;
                    ceilcap    [i][numcell[i]]=ceilwidth [i][numcell[i]];
                    floorheight[i][numcell[i]]=0;
                    floorwidth [i][numcell[i]]=0;
                    floorcap   [i][numcell[i]]=0;
```

```
                numcell[i]++;
            }
            else{
                ceilheight [i][0]=temp_node1->height;
                preht=temp_node1->height;
            }
            ceilwidth  [i][0]=hbinsize-culwidth;
            ceilcap    [i][0]=ceilwidth[i][0];
            floorheight[i][0]=temp_node1->height;
            floorwidth [i][0]=temp_node1->width;
            floorcap   [i][0]=0;

        }

        else {
            ceilheight [i][j-1]=preht - temp_node1->height;

            culwidth          =culwidth+temp_node1->width;
            ceilheight [i][j]  =temp_node1->height;
            ceilwidth  [i][j]  =hbinsize-culwidth;
            ceilcap    [i][j]  =ceilwidth[i][j];
            floorheight[i][j]  =temp_node1->height;
            floorwidth [i][j]  =temp_node1->width;;
            floorcap   [i][j]  =0;

            preht=temp_node1->height;

        }

        temp_node1=temp_node1->link;

    }

  }

}

for (i=0;i<hnumbin;i++){

  if (hbin_root[i]!=NULL){

    numcell[i]=hnitembin[i];
    temp_node1=hbin_root[i];
    preht = 0;
    culwidth=0;
```

```
for (j=0;(j<hnitembin[i]);j++){

    if (j==0){
        culwidth=temp_node1->width;
        if (temp_node1->vposition == (vnitembin[temp_node1->binnum]-1)){
            ceilheight [i][0]=temp_node1->height + vcurcap[temp_node1-
                >binnum];
            preht=temp_node1->height+ vcurcap[temp_node1->binnum];

            ceilheight [i][numcell[i]]=vcurcap[temp_node1->binnum];
            ceilwidth  [i][numcell[i]]=temp_node1->width;
            ceilcap    [i][numcell[i]]=ceilwidth [i][numcell[i]];
            floorheight[i][numcell[i]]=0;
            floorwidth [i][numcell[i]]=0;
            floorcap   [i][numcell[i]]=0;

            numcell[i]++;
        }
        else{
            ceilheight [i][0]=temp_node1->height;
            preht=temp_node1->height;
        }
        ceilwidth  [i][0]=hbinsize-culwidth;
        ceilcap    [i][0]=ceilwidth[i][0];
        floorheight[i][0]=temp_node1->height;
        floorwidth [i][0]=temp_node1->width;
        floorcap   [i][0]=0;

    }

    else if (j==1){
        ceilheight [i][0]=preht - temp_node1->height;

        culwidth        =culwidth+temp_node1->width;
        ceilheight [i][1]=temp_node1->height;
        ceilwidth  [i][1]=hbinsize-culwidth;
        ceilcap    [i][1]=ceilwidth[i][1];
        floorheight[i][1]=temp_node1->height;
        floorwidth [i][1]=temp_node1->width;;
        floorcap   [i][1]=0;

        preht=temp_node1->height;

    }
```

```
                else {
                    ceilheight [i][j-1]=preht - temp_node1->height;
                    ceilwidth  [i][j-1]=temp_node1->width;
                    ceilcap    [i][j-1]=ceilwidth[i][j-1];

                    culwidth          =culwidth+temp_node1->width;
                    ceilheight [i][j]  =preht;
                    ceilwidth  [i][j]  =hbinsize-culwidth;
                    ceilcap    [i][j]  =ceilwidth[i][j];
                    floorheight[i][j]  =temp_node1->height;
                    floorwidth [i][j]  =temp_node1->width;;
                    floorcap   [i][j]  =0;

                }
                temp_node1=temp_node1->link;

            }

        }

    }


/*compute utilised areas in each horizontal bin*/

    for (i=0;i<hnumbin;i++){
            hbinarea[i]=0;
            hareacap[i]=0;
    }
    for (i=0; i<hnumbin;i++){
        if (hbin_root[i]!=NULL){
            temp_node=hbin_root[i];
            for (k=0;(k<hnitembin[i]);k++){
                if (k==0){
                    hmaxht[i]=temp_node->height;
                    hstripsize[i]=hmaxht[i]*hbinsize;
                }

                hbinarea[i]=hbinarea[i]+temp_node->area;
                temp_node=temp_node->link;
            }
            hareacap[i]=hstripsize[i]-hbinarea[i];
        }

    }
```

```
    for (i=0;i<vnumbin;i++){
        vbinarea[i]=0;
        vareacap[i]=binarea;
    }
    for (i=0; i<vnumbin;i++){
        if (vbin_root[i]!=NULL){

        temp_node1=vbin_root[i];
        for (j=0;(j<vnitembin[i]);j++){

            vbinarea[i]=vbinarea[i]+hbinarea[temp_node1->stripnum];
            temp_node1=temp_node1->link;
        }
        vareacap[i]=binarea-vbinarea[i];

        }
    }


/*filling residual space in  horizontal levels */
T=1.0;

for (n=0;n<20;n++){

    for (i=0;i<vnumbin ; i++)
        bweight[i]= pow( (1.0+((double)vareacap[i]/(double)binarea)*0.05), T);

    for (i=0;i<hnumbin;i++){

        if (hbin_root[i]!=NULL)
            vweight[i]=bweight[hbin_root[i]->binnum];
        else
            vweight[i]=0.0;

    }
    for (i=0;i<hnumbin;i++)
        hweight[i]= pow( (1.0+((double)hareacap[i]/(double)hbinarea[i])*0.05), T);


    /*filling the residual space within each level */
    for (i=0;i<hnumbin;i++) {
        improved=0;

        if (hbin_root[i]!=NULL){

            for(j=0; j<hnumbin;j++){
```

```
                if (hbin_root[j]!=NULL){

                    if  (i!=j){

                        if (hbin_root[i]->binnum == hbin_root[j]->binnum)
                            improved = rgpack(&hbin_root[i], &hbin_root[j], i, j, hweight);
                        else
                            improved = rgpack(&hbin_root[i], &hbin_root[j], i, j, vweight);

                        if (hbin_root[i]==NULL)
                        break;
                        if (done) break;

                    }

                }

            }
            if (done) break;
        }

    }
    if (done) break;
    T=T*Tred;
}

count=0;
for (i=0;i<hnumbin;i++){
    count=count+hnitembin[i];
}

count=0;
for (i=0;i<vnumbin;i++){
    if (vbin_root[i]!=NULL)
        count++;
}

if (count==lowerbound){
    done=1;
}

if(done)
    break;
}   /* end of loop*/
```

```
_ftime(&time2);

    comptime= TimeElapsed(&time1,&time2);
    totaltime=totaltime+comptime;
    fprintf(op,"\n");
    fprintf(op, "Subroutine Elapsed time:%f \n", TimeElapsed(&time1,&time2));
    fprintf(op, "Program Elapsed time:  %f\n", (float)clock()/CLOCKS_PER_SEC);
    printf("Subroutine Elapsed time:%f \n", TimeElapsed(&time1,&time2));

    fclose(fp);
    fclose(op);
    return EXIT_SUCCESS;

}


/* functions*/

/*rotating items to achieve a tighter packing solution*/
int tightpack(Node **rootp)
{
    int i=0;
    int ht=0;
    int temp=0;
    int delta=0;

    Node *current_node;
    Node *deleted_node;

    current_node=*rootp;
    ht=current_node->height;
    current_node=current_node->link;
    i=1;

    while(current_node!=NULL){

        if (current_node->height< current_node->width){

            if(current_node->width < ht){
                temp=current_node->height;
                current_node->height = current_node->width;
                current_node->width  = temp;
                delta = current_node->height - current_node->width;

                if (current_node->rotation ==0)
```

```
                current_node->rotation=1;
            else
                current_node->rotation=0;

            hbinload[current_node->stripnum]=hbinload[current_node->stripnum]-delta;

            deleted_node=DeleteNode(rootp,i);
            HorInsertNode(rootp,deleted_node);
            return TRUE;
        }

    }


    current_node=current_node->link;
    i++;

}

return FALSE;

}


/*filling residual areas in each level*/
int rgpack(Node ** rootp1, Node **rootp2, int s, int t, double weight[])
{
    int k,n,i,j,p,q;
    int placement =0;
    int preposn = 0;
    double deltaf;
    int count=0;

    Node * deleted_node1;
    Node * deleted_node2;
    Node * current_node1;
    Node * temp_node1;
    Node * temp_node2;

    current_node1=*rootp1;

    p=(*rootp1)->binnum;
    q=(*rootp2)->binnum;


    for (k=0;(k<hnitembin[s]);k++){
```

```
for (n=0;(n<numcell[t]);n++){

    if  ((current_node1->width <= ceilcap[t][n]) &&
          (current_node1->height<= ceilheight[t][n])){

          if (current_node1->height>current_node1->width)
             placement =10;
          else{
              if(current_node1->width <=ceilheight[t][n])
                  placement=11;
              else
                  placement=10;
          }
       }

       else if ((current_node1->height <=  ceilcap[t][n]) &&
             (current_node1->width<= ceilheight[t][n])) {

          placement =11;

       }

       else if ((current_node1->width <= floorcap[t][n]) &&
             (current_node1->height<= floorheight[t][n])){

          if (current_node1->height>current_node1->width)
              placement=20;

          else{
              if(current_node1->width <=floorheight[t][n])
                  placement=21;
              else
                  placement=20;
              }

       }

       else if ((current_node1->height <= floorcap[t][n]) &&
             (current_node1->width<= floorheight[t][n])){

          placement=21;

       }

    if (placement){
```

```
if ((*rootp1)->binnum == (*rootp2)->binnum)
    deltaf=SQ(1.0*(hbinarea[s]*weight[s]-(current_node1->area)*weight[s]))
      +SQ(1.0*(hbinarea[t]*weight[t]+(current_node1->area)*weight[s]))
      -SQ(1.0*(hbinarea[s])*weight[s])-SQ(1.0*(hbinarea[t])*weight[t]);
else
    deltaf=SQ(1.0*(vbinarea[p]*weight[s]-(current_node1->area)*weight[s]))
      +SQ(1.0*(vbinarea[q]*weight[t]+(current_node1->area)*weight[s]))
      -SQ(1.0*(vbinarea[p])*weight[s])-SQ(1.0*(vbinarea[q])*weight[t]);

if (deltaf>0){


    vbinarea[p]=vbinarea[p]-(current_node1->area);
    vbinarea[q]=vbinarea[q]+(current_node1->area);
    vareacap[p]=binarea-vbinarea[p];
    vareacap[q]=binarea-vbinarea[q];

    hbinarea[s]=hbinarea[s]-(current_node1->area);
    hbinarea[t]=hbinarea[t]+(current_node1->area);
    hareacap[s]=hstripsize[s]-hbinarea[s];
    hareacap[t]=hstripsize[t]-hbinarea[t];

    /* update node data for bins*/
    if (placement == 10){

        ceilcap[t][n]=ceilcap[t][n]- current_node1->width;

        if (current_node1 ->floornum>=0){
            if (current_node1->orientation == 0)
                floorcap[s][current_node1->floornum]=floorcap[s]
                        [current_node1->floornum]+ current_node1->width;

            else if (current_node1->orientation == 1)
                floorcap[s][current_node1->floornum]=floorcap[s]
                        [current_node1->floornum] + current_node1->height;
        }

        else if (current_node1 ->ceilnum>=0){

            if (current_node1->orientation == 0)
                ceilcap[s][current_node1->ceilnum]=ceilcap[s][current_node1-
                    >ceilnum] + current_node1->width;
            else if (current_node1->orientation == 1)
                ceilcap[s][current_node1->ceilnum]=ceilcap[s][current_node1-
                    >ceilnum] + current_node1->height;
```

```
    }

    current_node1->orientation=0;
    current_node1->stripnum = t;
    current_node1->ceilnum  = n;
    current_node1->floornum =-1;

}

else if (placement == 11){

    ceilcap[t][n]=ceilcap[t][n]- current_node1->height;

    if (current_node1 ->floornum>=0) {

        if (current_node1->orientation == 0)
            floorcap[s][current_node1->floornum]=floorcap[s]
                [current_node1->floornum] + current_node1->width;

        else if (current_node1->orientation == 1)
            floorcap[s][current_node1->floornum]=floorcap[s]
                [current_node1->floornum] + current_node1->height;
    }


    else if (current_node1 ->ceilnum>=0){

        if (current_node1->orientation == 0)
            ceilcap[s][current_node1->ceilnum]=ceilcap[s][current_node1-
                >ceilnum] + current_node1->width;
        else if (current_node1->orientation == 1)
            ceilcap[s][current_node1->ceilnum]=ceilcap[s][current_node1-
                >ceilnum] + current_node1->height;

    }
    current_node1->orientation=1;
    current_node1->stripnum = t;
    current_node1->ceilnum  = n;
    current_node1->floornum =-1;

}

else if (placement == 20){

    floorcap[t][n]=floorcap[t][n]- current_node1->width;
```

```
    if (current_node1 ->floornum>=0){
        if (current_node1->orientation == 0)
            floorcap[s][current_node1->floornum]=floorcap[s]
                    [current_node1->floornum] + current_node1->width;

        else if (current_node1->orientation == 1)
            floorcap[s][current_node1->floornum]=floorcap[s]
                    [current_node1->floornum] + current_node1->height;
    }

    else if (current_node1 ->ceilnum>=0){

        if (current_node1->orientation == 0)
            ceilcap[s][current_node1->ceilnum]=ceilcap[s][current_node1-
                >ceilnum] + current_node1->width;
        else if (current_node1->orientation == 1)
            ceilcap[s][current_node1->ceilnum]=ceilcap[s][current_node1-
                >ceilnum] + current_node1->height;

    }
    current_node1->orientation=0;
    current_node1->stripnum = t;
    current_node1->ceilnum  =-1;
    current_node1->floornum = n;

}

else if (placement==21){

    floorcap[t][n]=floorcap[t][n]- current_node1->height;

    if (current_node1 ->floornum>=0){
        if (current_node1->orientation == 0)
            floorcap[s][current_node1->floornum]=floorcap[s]
                [current_node1->floornum] + current_node1->width;

        else if (current_node1->orientation == 1)
            floorcap[s][current_node1->floornum]=floorcap[s]
                [current_node1->floornum] + current_node1->height;
    }

    else if (current_node1 ->ceilnum>=0){

        if (current_node1->orientation == 0)
            ceilcap[s][current_node1->ceilnum]=ceilcap[s][current_node1-
```

```
                >ceilnum] + current_node1->width;
        else if (current_node1->orientation == 1)
            ceilcap[s][current_node1->ceilnum]=ceilcap[s][current_node1-
                >ceilnum] + current_node1->height;

    }
    current_node1->orientation=1;
    current_node1->stripnum = t;
    current_node1->ceilnum  =-1;
    current_node1->floornum = n;

}

deleted_node1= DeleteNode(rootp1,k);
UpdateHposition(rootp1);
hnitembin[s]--;

preposn = deleted_node1->vposition;
deleted_node1->binnum    =(*rootp2)->binnum;
deleted_node1->vposition=(*rootp2)->vposition;

HorInsertNode(rootp2,deleted_node1);
UpdateHposition(rootp2);
hnitembin[t]++;


if (((*rootp1)==NULL)&&(deleted_node1!=NULL)){

    deleted_node2= DeleteNode(&vbin_root[p], preposn);
    UpdateVposition(&vbin_root[p]);
    vnitembin[p]--;

    temp_node1=vbin_root[p];
    for (i=0;(i<vnitembin[p]);i++){

        temp_node2=hbin_root[temp_node1->stripnum];

        for(j=0;j<hnitembin[temp_node1->stripnum];j++){
            temp_node2->vposition=i;
            temp_node2=temp_node2->link;
        }

    temp_node1=temp_node1->link;
    }

}
```

```
                    for (i=0;i<vnumbin ; i++){
                        if (vbin_root[i]!=NULL)
                            count++;
                    }
                    if (count == lowerbound){
                        done=1;
                    }
                    return TRUE;
                }

            }
            placement = 0;

        }
        current_node1=current_node1->link;

    }

return FALSE;
}


void PrintFileNode(Node * item)
{
    fprintf(op, "%8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d \n", item-
>height,item->width,item->stripnum, item->hposition, item->binnum,item-
>vposition,item->floornum,item->ceilnum,item->area,item->orientation, item->rotation);

}


void PrintNode(Node * item)
{
    printf("%8d %8d %8d %8d %8d %8d %8d %8d %8d %8d  %8d \n", item-
>height,item->width,item->stripnum, item->hposition,item->binnum,item-
>vposition,item->floornum,item->ceilnum,item->area, item->orientation, item->
rotation);

}


Node * CreatNode(int high, int wide)
{
    struct NODE * temp = (struct NODE*) malloc(sizeof(struct NODE));
```

```
    temp->height=high;
    temp->width=wide;
    temp->stripnum=0;
    temp->binnum=0;
    temp->hposition=0;
    temp->vposition=0;
    temp->ceilnum=0;
    temp->floornum=0;
    temp->area=0;
    temp->rotation=0;
    temp->orientation=0;

    return temp;
}


Node * AddNode(int high, int wide, int num1, int num2, Node * bin )
{
    struct NODE * temp = (struct NODE*) malloc(sizeof(struct NODE));

    temp->height = high;
    temp->width  = wide;
    temp->stripnum=num1;
    temp->binnum=num2;
    temp->vposition=0;
    temp->ceilnum=0;
    temp->floornum=0;
    temp->area=0;
    temp->rotation=0;
    temp->orientation=0;
    temp->link = bin;
    bin = temp;
    return bin;
}



int HorInsertNode( register Node **linkp, Node * inserted_node)
{
    register Node   *current;
    while( ( current = *linkp ) != NULL &&
        current->height >=inserted_node->height)
        linkp = &current->link;

    inserted_node->link = current;
    *linkp = inserted_node;
```

```
    return TRUE;
}


int VerInsertNode( register Node **linkp, Node * inserted_node)
{
    register Node   *current;
    while( ( current = *linkp ) != NULL &&
        hbinload[current->stripnum] >hbinload[inserted_node->stripnum])
        linkp = &current->link;

    inserted_node->link = current;
    *linkp = inserted_node;

    return TRUE;
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//*  2bp_r_g.h  is the header file for WA2BPRG.cpp
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

typedef  struct NODE   {
    struct NODE   *link;
    int height;
    int width;
    int stripnum;
    int binnum;
    int hposition;
    int vposition;
    int floornum;
    int ceilnum;
    int resinum;
    int area;
    int orientation;
    int rotation;

} Node;

void sort(unsigned long n, double arr[]);
#define SQ(a) (a*a)
#define  FALSE  0
#define TRUE 1
#define MAX_NUMBER_BIN 100
#define MAX_NUMBER_ITEM 150
#define MAX_NUMBER_HOR  20
#define MAX_NUMBER_VER 10
```

Appendix E

Two-Dimensional Bin Packing Problem Code
(Free Cuts, Oriented Items)

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//*     WA2BPOF.cpp is the program for solving the two Dimensional Bin Packing
//with free cuts and oriented item.
//*     It calls functions from WA2BPOG.cpp
//*     It is to be complied with the following files (ANSI version) from Press et al.(2002)
//          1)     nrutil.cpp
//          2)     nrutil.h
//          3)     nr.h
//*   The header file 2bp_o_f.h is appended at the end of the program.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////


#include <io.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "nrutil.h"
#include "nr.h"
#include <sys\timeb.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include "2bp_o_f.h"
#include <string.h>

Node * bin_root[MAX_NUMBER_BIN];
Node * vbin_root[MAX_NUMBER_BIN];
Node * hbin_root[MAX_NUMBER_BIN];
Node * vertical_bin[MAX_NUMBER_BIN];
Node * space[MAX_NUMBER_BIN];

Node * obj_size;
Node * strip_size;

FILE * fp;
FILE * op;
```

```
int binload[MAX_NUMBER_BIN];
int vbinload[MAX_NUMBER_BIN];
int hbinload[MAX_NUMBER_BIN];
int vbinarea[MAX_NUMBER_BIN];
int vareacap[MAX_NUMBER_BIN];
int hbinarea[MAX_NUMBER_BIN];
int hareacap[MAX_NUMBER_BIN];
int hmaxht[MAX_NUMBER_BIN];
int hstripsize[MAX_NUMBER_BIN];
int curcap[MAX_NUMBER_BIN];
int vcurcap[MAX_NUMBER_BIN];
int hcurcap[MAX_NUMBER_BIN];
int nitembin[MAX_NUMBER_BIN];
int vnitembin[MAX_NUMBER_BIN];
int vernitembin[MAX_NUMBER_BIN];
int hnitembin[MAX_NUMBER_BIN];
int numcell[MAX_NUMBER_BIN];
int stackheight[MAX_NUMBER_BIN];
double evareacap[MAX_NUMBER_BIN];

int ceilheight [MAX_NUMBER_BIN][MAX_NUMBER_HOR];
int ceilwidth  [MAX_NUMBER_BIN][MAX_NUMBER_HOR];
int ceilcap    [MAX_NUMBER_BIN][MAX_NUMBER_HOR];
int floorheight[MAX_NUMBER_BIN][MAX_NUMBER_HOR];
int floorwidth [MAX_NUMBER_BIN][MAX_NUMBER_HOR];
int floorcap   [MAX_NUMBER_BIN][MAX_NUMBER_HOR];

int binsize=0;
int bound=0;
int vbinsize=0;
int hbinsize=0;
int binarea=0;

int vnumbin=30;
int maxhnumbin=100;
int hnumbin=0;

double comptime=0.0;
double totaltime=0.0;
double timelimit=300.0;
double ebinarea=0.0;
int done =0;
int loop =0;

double TimeElapsed(struct _timeb *begin, struct _timeb *end);
int irbit1(unsigned long *iseed);
```

```
void PrintNode(Node * item);
void PrintAllNode(Node * root);
void PrintFileNode(Node * item);
void PrintFile(Node * root);
void removebin (Node **linkp);
void UpdateHposition(Node **rootp);
void UpdateVposition(Node **rootp);

Node * AddNode(int high, int wide, int num1, int num2, Node * bin);
Node * CreatNode(int high, int wide);
Node * CopyNode(Node * root);
Node * DeleteNode( Node **linkp, int item_num );
Node * DeleteTagNode( Node **linkp);
int HorInsertNode( register Node **linkp, Node * inserted_node);
int VerInsertNode( register Node **linkp, Node * inserted_node);
int HorInsertOrderNode( register Node **linkp, Node * inserted_node);

int CompVbinLoad(Node * root);
int CompHbinLoad(Node * root);
int CompBinItem(Node * root);

int vswap10(Node ** rootp1, Node **rootp2, int x, int y, double vweight[]);
int vswap11(Node ** rootp1, Node **rootp2, int x, int y, double vweight[]);
int vswap12(Node ** rootp1, Node **rootp2, int x, int y, double vweight[]);
int vswap22(Node ** rootp1, Node **rootp2, int x, int y, double vweight[]);

int hswap10(Node ** rootp1, Node **rootp2, Node * temp_node1, Node * temp_node2,
        int x, int y, int xbin, int ybin, double hweight[]);
int hswap11(Node ** rootp1, Node **rootp2, Node * temp_node1, Node * temp_node2,
        int x, int y, int xbin, int ybin, double hweight[]);
int hswap12(Node ** rootp1, Node **rootp2, Node * temp_node1, Node * temp_node2,
        int x, int y, int xbin, int ybin, double hweight[]);
int hswap22(Node ** rootp1, Node **rootp2, Node * temp_node1, Node * temp_node2,
        int x, int y, int xbin, int ybin, double hweight[]);
int levelpack(Node ** rootp1, Node **rootp2, int s, int t, double hweight[]);

int alterdirection(Node ** rootp);

int fswap10(Node ** rootp1, Node **rootp2, int i, int p, double weight[]);
int fswap11(Node ** rootp1, Node **rootp2, int i, int p, double weight[]);
int fswap12(Node ** rootp1, Node **rootp2, int i, int p, double weight[]);
int fswap22(Node ** rootp1, Node **rootp2, int i, int p, double weight[]);

void ComputeSpace(Node ** rootp);

double binpara;
```

```
int lowerbound=16;
int
main(void)
{
    struct _timeb time1, time2;
    unsigned long iseed;
    Node * deleted_node;
    Node * deleted_node1;
    Node * deleted_node2;
    Node * inserted_node;
    Node * current_node;
    Node * temp_node;
    Node * temp_node1;
    Node * temp_node2;
    int i,j,k,idec,p,q,m,n,s,preht;
    int numbitem=0;
    int improved=0;
    int feasible=0;
    int numtype=0;
    int temp1=0;
    int temp2=0;
    int numstrip=0;
    int empty=0;
    int data=0;
    int count=0;
    int count1=0;
    int count2=0;
    int binheight=0;
    int culwidth=0;
    int maxind=0;
    int maxsize=0;
    int minind=0;
    int mincap=0;
    double maxcap=0;
    double hweight [MAX_NUMBER_BIN];
    double vweight [MAX_NUMBER_BIN];
    double bweight [MAX_NUMBER_BIN];
    int height[MAX_NUMBER_ITEM];
    int width[MAX_NUMBER_ITEM];
    double T = 1.0;
    double Tred =0.95;
    float temp=0.0;
    char str[10];
    done=0;
    iseed=111;
```

```
    if((op=fopen("out.txt", "w")) == NULL) {
        printf("Cannot open file.\n");
        exit(1);
    }
    /* Input file format – Berky and Wang (1996) */
    if((fp=fopen( "C5_60_9.txt", "r")) == NULL)
    {       printf("Cannot open file.\n");
        exit(1);
    }

    fscanf(fp, "%d %s %s", &data, &str, &str);
    fscanf(fp, "%d %s %s %s", &numbitem, &str, &str, &str);
    fscanf(fp, "%d %d %s %s %s %s %s %s", &data, &data, &str,
              &str, &str, &str, &str, &str);
    fscanf(fp, "%d %d %s", &vbinsize, &hbinsize, &str);

    for(i =0; i <numbitem;i++){
        if (i==0)
            fscanf(fp, "%d  %d %s", &height[i], &width[i], &str);
        else
            fscanf(fp, "%d  %d ", &height[i], &width[i]);

    }

    bound=0;
    hbinsize=hbinsize+bound;
    vbinsize=vbinsize+bound;
    binarea=vbinsize*hbinsize;

_ftime(&time1);

    for (loop=0;loop<20;loop++){

    /* initialisation*/
    obj_size=NULL;
    strip_size=NULL;
    deleted_node  =NULL;
    deleted_node1 =NULL;
    deleted_node2 =NULL;
    inserted_node =NULL;
    current_node  =NULL;
    temp_node=NULL;
    temp_node1=NULL;
    temp_node2=NULL;
```

```
vnumbin=lowerbound+5;

for (i=0;i<vnumbin;i++){
    vbin_root[i] = NULL;
    vertical_bin[i]=NULL;
    space[i]=NULL;
}
for (i=0;i<maxhnumbin;i++)
    hbin_root[i] = NULL;

for( i=0;i<vnumbin;i++)
    vcurcap[i]=vbinsize;
for( i=0;i<maxhnumbin;i++)
    hcurcap[i]=hbinsize;

for( i=0;i<vnumbin;i++){
    vbinload[i]=0;
    vnitembin[i]=0;
    vernitembin[i]=0;
}
for( i=0;i<maxhnumbin;i++){
    hbinload[i]=0;
    hnitembin[i]=0;
}
for (i=0;i< MAX_NUMBER_BIN;i++){
    for(j=0;j<MAX_NUMBER_HOR; j++){
        ceilheight [i][j]=0;
        ceilwidth  [i][j]=0;
        ceilcap  [i][j]=0;
        floorheight[i][j]=0;
        floorwidth [i][j]=0;
        floorcap   [i][j]=0;

    }
}

for (i=0;i<MAX_NUMBER_BIN;i++)
    numcell[i]=0;

T=1.0;
Tred=0.95;
numstrip=0;
count=0;

    /*read data file*/
    for (i=0;i<numbitem;i++){
```

```
    current_node=CreatNode(height[i], width[i]);
    current_node->area=height[i]*width[i];
    HorInsertNode(&obj_size, current_node);
}

idec=irbit1(&iseed);

/* first-fit decreasing for horizontal bins*/
for (i=0;i<numbitem;i++){
    idec=irbit1(&iseed);

    if ((i== (numbitem-1))||(i==(numbitem-2))) idec=0;

    deleted_node= DeleteNode(&obj_size,idec);
    for (j=0;j<maxhnumbin;j++)
        if (hcurcap[j] >= deleted_node->width){
            hcurcap[j]=hcurcap[j]- (deleted_node->width);
            deleted_node->stripnum=j;
            HorInsertNode(&hbin_root[j],deleted_node);

            break;
        }
}

/*linked list for vertical bins*/
for (i=(maxhnumbin-1);i>=0;i--){

    if (hbin_root[i] !=NULL){
        numstrip++;
        strip_size = AddNode(hbin_root[i]->height, hbin_root[i]->width,
        hbin_root[i]->stripnum, hbin_root[i]->binnum, strip_size);
    }
}

hnumbin=numstrip;

/*compute the total load and number of items in each bin*/
for (i=0; i<hnumbin;i++) {
    hbinload[i]=CompHbinLoad(hbin_root[i]);
    hnitembin[i]=CompBinItem(hbin_root[i]);
}



/* first-fit decreasing for vertical bins*/
for (i=0;i<hnumbin;i++){
```

```
    deleted_node= DeleteNode(&strip_size,0);
    for (j=0;j<vnumbin;j++)
        if (vcurcap[j] >= deleted_node->height){
            vcurcap[j]=vcurcap[j]- (deleted_node->height);

            temp_node=hbin_root[deleted_node->stripnum];
            for (k=0;(k<hnitembin[(deleted_node->stripnum)]);k++){
                temp_node->binnum=j;
                if (temp_node->link != NULL)
                temp_node=temp_node->link;
            }
            temp_node=NULL;

            deleted_node ->binnum=j;
            VerInsertNode(&vbin_root[j],deleted_node);
            break;
        }
}

/*compute the total load and number of items in each bin*/
for (i=0; i<vnumbin;i++) {
    vbinload[i]=CompVbinLoad(vbin_root[i]);
    vnitembin[i]=CompBinItem(vbin_root[i]);
}


/*compute utilised areas in each horizontal bin*/

for (i=0;i<hnumbin;i++)
        hbinarea[i]=0;

for (i=0; i<hnumbin;i++){
    temp_node=hbin_root[i];
    for (k=0;(k<hnitembin[i]);k++){
        hbinarea[i]=hbinarea[i]+temp_node->area;
        temp_node=temp_node->link;
    }
}

for (m=0;m<20;m++){

    /* update weights, set K = 0.05*/
    for (i=0;i<hnumbin ; i++){
        hweight[i]= pow((1.0+((double)hcurcap[i]/(double)hbinsize)*0.05), T);
    }
```

```
/*swap item amongst levels */
for (i=0;i<vnumbin;i++) {
    improved=0;

    temp_node1=vbin_root[i];
    for(j=0; j<vnitembin[i];j++){

        for (p=0;p<vnumbin;p++){

            temp_node2=vbin_root[p];
            for(q=0;q<vnitembin[p];q++){

                if  ((temp_node1->stripnum)!=(temp_node2->stripnum)){


                    improved = hswap10(&hbin_root[temp_node1->stripnum],
                        &hbin_root[temp_node2->stripnum], temp_node1,
                         temp_node2,(temp_node1->stripnum),
                        (temp_node2->stripnum),i, p, hweight);

                    if (hbin_root[temp_node1->stripnum]==NULL){
                        removebin(&vbin_root[i]);
                        vnitembin[i]--;
                        break;
                    }
                    if (done) break;

                    improved = hswap11(&hbin_root[temp_node1->stripnum],
                        &hbin_root[temp_node2->stripnum], temp_node1,
                        temp_node2,(temp_node1->stripnum),
                        (temp_node2->stripnum), i, p, hweight);
                    if (done) break;

                    improved = hswap12(&hbin_root[temp_node1->stripnum],
                        &hbin_root[temp_node2->stripnum], temp_node1,
                        temp_node2,(temp_node1->stripnum),
                        (temp_node2->stripnum), i, p, hweight);
                    if (done) break;

                    improved = hswap22(&hbin_root[temp_node1->stripnum],
                        &hbin_root[temp_node2->stripnum], temp_node1,
                         temp_node2,(temp_node1->stripnum),
                        (temp_node2->stripnum), i, p, hweight);
                    if (done) break;
```

```
                    }

                    temp_node2=temp_node2->link;

                }
                if (hbin_root[temp_node1->stripnum]==NULL) break;
                if (done) break;
            }
            if (done) break;
            if(temp_node1 !=NULL)
            temp_node1=temp_node1->link;
        }
        if (done) break;
    }
    if(done) break;
    T=T*Tred;

}

/*packing vertical bins*/
    T=1.0;
    for (m=0;m<20;m++){

        /* update weights, set K = 0.05*/

        for (i=0;i<vnumbin ; i++)
            vweight[i]= pow((1.0+((double)vcurcap[i]/(double)vbinsize)*0.05), T);

        /*swapping items between vertical bins*/
        for (i=0;i<vnumbin;i++) {
            improved=0;

            for (p=0;p<vnumbin;p++){

                if  (p!=i){

                    improved = vswap10(&vbin_root[i], &vbin_root[p],i,p, vweight);
                    if (done) break;

                    improved = vswap11(&vbin_root[i], &vbin_root[p],i,p,vweight);
                    if (done) break;

                    improved = vswap12(&vbin_root[i], &vbin_root[p],i,p,vweight);
                    if (done) break;

                    improved = vswap22(&vbin_root[i], &vbin_root[p],i,p,vweight);
```

```
                if (done) break;

            }

        }

        if (done) break;
    }
    if (done) break;
    T=T*Tred;
}


count1=0;
for (i=0;i<vnumbin;i++){
    if (vbin_root[i]!=NULL)
        count1++;
}

/*initialise hposition, vposition, floornum, ceilnum for horizontal levels  after hswap*/

        for (i=0;i<vnumbin;i++){
            temp_node1=vbin_root[i];

            for (j=0;(j<vnitembin[i]);j++){
                s=temp_node1->stripnum;
                temp_node2=hbin_root[s];

                for (k=0;(k<hnitembin[s]);k++){

                    temp_node2->binnum=i;
                    temp_node2->hposition = k;
                    temp_node2->vposition = j;
                    temp_node2->floornum = k;
                    temp_node2->ceilnum  =-1;
                    temp_node2=temp_node2->link;
                }

                temp_node1->binnum=i;
                temp_node1->hposition=0;
                temp_node1->vposition = j;
                temp_node1->floornum = 0;
                temp_node1->ceilnum  =-1;
                temp_node1->area=temp_node1->height * temp_node1->width;
                temp_node1=temp_node1->link;
            }
```

```
        }

/* determine space occupied by items, and residual areas in each level */

        for (i=0;i<hnumbin;i++){

            if (hbin_root[i]!=NULL){

                numcell[i]=hnitembin[i];
                temp_node1=hbin_root[i];
                preht = 0;
                culwidth=0;

                for (j=0;(j<hnitembin[i]);j++){

                    if (j==0){
                        culwidth=temp_node1->width;
                        if (temp_node1->vposition == (vnitembin[temp_node1->binnum]-1)){
                            ceilheight [i][0]=temp_node1->height + vcurcap[temp_node1-
                                >binnum];
                            preht=temp_node1->height+ vcurcap[temp_node1->binnum];
                            ceilheight [i][numcell[i]]=vcurcap[temp_node1->binnum];
                            ceilwidth  [i][numcell[i]]=temp_node1->width;
                            ceilcap    [i][numcell[i]]=ceilwidth [i][numcell[i]];
                            floorheight[i][numcell[i]]=0;
                            floorwidth [i][numcell[i]]=0;
                            floorcap   [i][numcell[i]]=0;

                            numcell[i]++;
                        }
                        else{
                            ceilheight [i][0]=temp_node1->height;
                            preht=temp_node1->height;
                        }
                        ceilwidth  [i][0]=hbinsize-culwidth;
                        ceilcap    [i][0]=ceilwidth[i][0];
                        floorheight[i][0]=temp_node1->height;
                        floorwidth [i][0]=temp_node1->width;
                        floorcap   [i][0]=0;

                    }

                    else if (j==1){
                        ceilheight [i][0]=preht - temp_node1->height;
                        culwidth        =culwidth+temp_node1->width;
                        ceilheight [i][1]=temp_node1->height;
```

```
                    ceilwidth  [i][1]=hbinsize-culwidth;
                    ceilcap    [i][1]=ceilwidth[i][1];
                    floorheight[i][1]=temp_node1->height;
                    floorwidth [i][1]=temp_node1->width;;
                    floorcap   [i][1]=0;

                    preht=temp_node1->height;

                }

                else {
                    ceilheight [i][j-1]=preht - temp_node1->height;
                    ceilwidth  [i][j-1]=temp_node1->width;
                    ceilcap    [i][j-1]=ceilwidth[i][j-1];

                    culwidth          =culwidth+temp_node1->width;
                    ceilheight [i][j]  =preht;
                    ceilwidth  [i][j]  =hbinsize-culwidth;
                    ceilcap    [i][j]  =ceilwidth[i][j];
                    floorheight[i][j]  =temp_node1->height;
                    floorwidth [i][j]  =temp_node1->width;;
                    floorcap   [i][j]  =0;

                }
                temp_node1=temp_node1->link;

            }

        }

    }

/*compute utilised areas in each horizontal bin*/

    for (i=0;i<hnumbin;i++){
        hbinarea[i]=0;
        hareacap[i]=0;
    }
    for (i=0; i<hnumbin;i++){
        if (hbin_root[i]!=NULL){
            temp_node=hbin_root[i];
            for (k=0;(k<hnitembin[i]);k++){
                if (k==0){
                    hmaxht[i]=temp_node->height;
                    hstripsize[i]=hmaxht[i]*hbinsize;
            }
```

```
                hbinarea[i]=hbinarea[i]+temp_node->area;
                temp_node=temp_node->link;
            }
            hareacap[i]=hstripsize[i]-hbinarea[i];
        }

    }

    for (i=0;i<vnumbin;i++){
        vbinarea[i]=0;
        vareacap[i]=binarea;
    }
    for (i=0; i<vnumbin;i++){
        if (vbin_root[i]!=NULL){

        temp_node1=vbin_root[i];
        for (j=0;(j<vnitembin[i]);j++){

            vbinarea[i]=vbinarea[i]+hbinarea[temp_node1->stripnum];
            temp_node1=temp_node1->link;
        }
        vareacap[i]=binarea-vbinarea[i];

        }
    }

/*filling residual space in  horizontal levels */
T=1.0;

for (n=0;n<20;n++){

    for (i=0;i<vnumbin ; i++)
        bweight[i]= pow( (1.0+((double)vareacap[i]/(double)binarea)*0.05), T);

    for (i=0;i<hnumbin;i++){

        if (hbin_root[i]!=NULL)
            vweight[i]=bweight[hbin_root[i]->binnum];
        else
            vweight[i]=0.0;

    }
    for (i=0;i<hnumbin;i++)
        hweight[i]= pow( (1.0+((double)hareacap[i]/(double)hbinarea[i])*0.05), T);
```

```
    /*filling the residual space within each level */
    for (i=0;i<hnumbin;i++) {
        improved=0;

        if (hbin_root[i]!=NULL){

            for(j=0; j<hnumbin;j++){

                if (hbin_root[j]!=NULL){

                    if  (i!=j){

                        if (hbin_root[i]->binnum == hbin_root[j]->binnum)
                            improved = levelpack(&hbin_root[i], &hbin_root[j], i, j, hweight);
                        else
                            improved = levelpack(&hbin_root[i], &hbin_root[j], i, j, vweight);

                        if (hbin_root[i]==NULL)
                        break;
                        if (done) break;

                    }

                }

            }
            if (done) break;
        }

    }
    if (done) break;
    T=T*Tred;

}

count=0;
for (i=0;i<hnumbin;i++){
    count=count+hnitembin[i];
}
count=0;
for (i=0;i<vnumbin;i++){
    if (vbin_root[i]!=NULL)
        count++;
}
```

```
/*Exit the loop if the 2BP|O|G solution conincide with the best lower bound*/
if (count==lowerbound){
    done=1;
}

if(done)
    break;


/* consolidate all items in a  vertical bin, remove empty vertical bins and re-number
bins*/
count1=0;
count2=vnumbin;

for (i=0;i<vnumbin;i++){
    temp_node1=vbin_root[i];
    if(vbin_root[i]!=NULL){
        for (j=0;(j<vnitembin[i]);j++){
            s=temp_node1->stripnum;
            temp_node2=hbin_root[s];
            if(hbin_root[s]!=NULL){
                for (k=0;(k<hnitembin[s]);k++){
                    deleted_node= DeleteNode(&hbin_root[s], 0);
                    deleted_node->binnum=count1;
                    deleted_node->stripnum=0;
                    deleted_node->hposition=0;
                    deleted_node->vposition=0;
                    deleted_node->floornum=0;
                    deleted_node->ceilnum=0;
                    HorInsertNode(&vertical_bin[count1],deleted_node);
                    vernitembin[count1]++;
                    if (temp_node2!=NULL)
                        temp_node2=temp_node2->link;
                }
            }
            temp_node1=temp_node1->link;
        }
        count1++;
    }
    else {
    count2--;

    }
}
vnumbin=count2;
```

```
/* recompute utilised bin areas after renumber*/

for (i=0;i<(vnumbin+2);i++){
    vbinarea[i]=0;
    vareacap[i]=binarea;
}
for (i=0; i<vnumbin;i++){

    temp_node1=vertical_bin[i];

    while (temp_node1!=NULL){

        vbinarea[i]=vbinarea[i]+ temp_node1->area;
        temp_node1=temp_node1->link;

    }
    vareacap[i]=binarea-vbinarea[i];

}

/*repack all bins using altenate direction algorithm*/

for(i=0;i<vnumbin;i++){
    stackheight[i]=alterdirection(&vertical_bin[i]);
}

feasible=0;
while(feasible==0){

    feasible=1;
    for (i=0;i<vnumbin;i++){

        /* repack infeasible bins*/
        if(stackheight[i]>vbinsize){
            deleted_node = DeleteNode(&vertical_bin[i], (vernitembin[i]-1));
            vernitembin[i]--;
            vbinarea[i]=vbinarea[i]-deleted_node->area;
            vareacap[i]=binarea-vbinarea[i];
            stackheight[i]=alterdirection(&vertical_bin[i]);

            if (vareacap[vnumbin]>=deleted_node->area){
                deleted_node->binnum=vnumbin;
                HorInsertNode(&vertical_bin[vnumbin],deleted_node);
                vernitembin[vnumbin]++;
                vbinarea[vnumbin]=vbinarea[vnumbin]+deleted_node->area;
                vareacap[vnumbin]=binarea-vbinarea[vnumbin];
```

```
                stackheight[vnumbin]=alterdirection(&vertical_bin[vnumbin]);
                feasible=0;
                break;
            }
            else{
                deleted_node->binnum=vnumbin+1;
                HorInsertNode(&vertical_bin[vnumbin+1],deleted_node);
                vernitembin[vnumbin+1]++;
                vbinarea[vnumbin+1]=vbinarea[vnumbin+1]+deleted_node->area;
                vareacap[vnumbin+1]=binarea-vbinarea[vnumbin+1];
                stackheight[vnumbin+1]=alterdirection(&vertical_bin[vnumbin+1]);
                feasible=0;
                vnumbin++;
                break;

            }
        }
    }
}

vnumbin++;
for(i=0;i<vnumbin;i++){
    /*pack the items in alternate directions, and compute the stack height*/
    stackheight[i]=alterdirection(&vertical_bin[i]);
}

T=1.0;
Tred=0.95;

/*swapping items between bins, and repacking using alternate direction algorithm*/
for (n=0;n<30;n++){

    for (i=0;i<vnumbin ; i++)
        bweight[i]= pow( (1.0-((1.0*stackheight[i])/(1.0*vbinsize))*0.05), T);

    for (i=0;i<vnumbin;i++) {

        for(j=0; j<vnumbin;j++){

            if  ( i!=j){

                        improved = fswap10(&vertical_bin[i], &vertical_bin[j], i, j,
                                bweight);
                        if (done) break;
```

```
                        improved = fswap11(&vertical_bin[i], &vertical_bin[j], i, j,
                                bweight);
                        if (done) break;

                        improved = fswap12(&vertical_bin[i], &vertical_bin[j], i, j,
                                bweight);
                        if (done) break;

                        improved = fswap22(&vertical_bin[i], &vertical_bin[j], i, j,
                                bweight);
                        if (done) break;

                }

            }
            if (done) break;

        }
        if (done) break;
        T=T*Tred;

}

if (done) break;

for (i=0;i<vnumbin;i++){

    /*pack the items in alternate directions, and compute the stack height*/
    stackheight[i]=alterdirection(&vertical_bin[i]);

    /* compute the coordinates of dead space*/
    if(vertical_bin[i]!=NULL)
    ComputeSpace(&vertical_bin[i]);

}

maxcap=0;
maxind=0;

for (i=0;i<vnumbin;i++){

    if ((vareacap[i]>maxcap)&&(vareacap[i]!=binarea)){
        maxcap=vareacap[i];
        maxind=i;
    }
}
```

```
count1=0;


/* move items to occupy dead space*/
for (i=0;i<vernitembin[maxind];i++){

    improved=0;

    for(j=0;j<vnumbin;j++){

        if ((j!=maxind)||(vareacap[j]!=binarea)){

            temp_node1=space[j];

            while(temp_node1!=NULL){

                if( ((temp_node1->height>=vertical_bin[maxind]->height) && (temp_node1-
                    >width >=vertical_bin[maxind]->width)) && (temp_node1->tag!=1) ){

                    deleted_node=DeleteNode(&vertical_bin[maxind],0);
                    temp_node1->tag=1;
                    deleted_node->tag=1;
                    count1++;
                    HorInsertNode(&vertical_bin[temp_node1->binnum], deleted_node);
                    vernitembin[temp_node1->binnum]++;
                    improved=1;

                    count=0;
                    done=0;
                    for (k=0;k<vnumbin ; k++)
                        if (vertical_bin[k]!=NULL)
                            count++;

                    if (count==lowerbound){
                        done = 1;
                        stackheight[maxind]=0;
                        vernitembin[maxind]=0;
                    }

                    break;
                }

                temp_node1=temp_node1->link;
            }
            if(improved==1)
```

```
                break;

        }
        if(done==1)
            break;

    }
    if(done==1)
        break;

}



if (done==1)
    break;
else
    vernitembin[maxind]=vernitembin[maxind]-count1;

_ftime(&time2);
totaltime=TimeElapsed(&time1,&time2);
printf("time = %f\n",totaltime);
if(totaltime>timelimit)
    break;

}   /* end of loop*/


    fprintf(op,"\n");
    fprintf(op, "Subroutine Elapsed time:%f \n", totaltime);
    fprintf(op, "Program Elapsed time:  %f\n", (float)clock()/CLOCKS_PER_SEC);
    printf("Subroutine Elapsed time:%f \n", totaltime);

    fclose(fp);
    fclose(op);
    return EXIT_SUCCESS;

}




/* functions*/

/*compute the coordinates of dead space of a bin*/
```

```
void ComputeSpace(Node ** rootp)
{

    int i,j,count;
    int spacecount;
    int item=0;
    int band=0;
    int delnum=0;
    int numfail=0;
    int succeed = 0;
    int culwidth=0;
    int startco_ord = 0;
    int current_max_ht =0;
    int current_min_depth =0;
    int current_max_right =0;
    int current_max_left =0;
    int current_min_left =0;
    int current_min_right =0;
    int current_top = 0;
    int leftcorner =0;
    int rightcorner =0;
    int itemheight =0;
    int maxband=0;
    int itemcover=0;
    int spacecover=0;
    int itemtop[MAX_NUMBER_ITEM];
    int itembottom[MAX_NUMBER_ITEM];
    int itemleft  [MAX_NUMBER_ITEM];
    int itemright [MAX_NUMBER_ITEM];
    int spacetop[MAX_NUMBER_ITEM];
    int spacebottom[MAX_NUMBER_ITEM];
    int spaceleft  [MAX_NUMBER_ITEM];
    int spaceright [MAX_NUMBER_ITEM];

    Node * current_node1=NULL;
    Node * current_node2=NULL;
    Node * current_node3=NULL;
    Node * new_node =NULL;
    Node * new_node1 =NULL;
    Node * new_node2 =NULL;
    Node * temp_node1 = NULL;
    Node * temp_node2=NULL;
    Node * deleted_node =NULL;

    for (i=0;i<MAX_NUMBER_ITEM;i++){
```

```
    itemtop     [i]=0;
    itembottom  [i]=0;
    itemleft    [i]=0;
    itemright   [i]=0;
    spacetop    [i]=0;
    spacebottom  [i]=0;
    spaceleft    [i]=0;
    spaceright   [i]=0;

}

while (numfail<3){

    delnum=0;
    culwidth=0;
    startco_ord=0;
    succeed=0;
    count=0;
    current_node1=*rootp;

    while (current_node1!=NULL){

        if ((startco_ord + current_node1->width) <= hbinsize){

                leftcorner=culwidth;
                culwidth=culwidth+current_node1->width;
                startco_ord=culwidth;
                current_max_ht=0;
                rightcorner=culwidth;

                for (i=0;i<item;i++){
                    if ( (itemtop[i]> current_max_ht)&&
                        ( (( leftcorner<itemright[i])&&( leftcorner >  itemleft[i]))||
                        ((rightcorner<itemright[i])&&(rightcorner >  itemleft[i]))||
                        ((rightcorner>=itemright[i])&&(leftcorner <= itemleft[i])) ))

                        current_max_ht=itemtop[i];
                }

                itemheight=current_node1->height + current_max_ht;

                itembottom[item]=current_max_ht;
                itemleft[item]=leftcorner;
                itemright[item]=rightcorner;
                itemtop[item]=itemheight;
```

```
            current_node1=current_node1->link;

            deleted_node=DeleteNode(rootp, delnum);

            deleted_node->vposition=band;
            deleted_node->hposition=count;
            deleted_node->floornum=item;
            HorInsertNode(&current_node2, deleted_node);

            item++;
            count++;
            succeed=1;


    }
    else{

        current_node1=current_node1->link;
        delnum++;
    }

}

if (succeed==0)
    numfail++;
else{
    numfail=0;
}

band++;

delnum=0;
culwidth=hbinsize;
startco_ord=hbinsize;
succeed=0;
count=0;
current_node1=*rootp;

while (current_node1!=NULL){

    if (startco_ord - current_node1->width >= 0){

        rightcorner=culwidth;
        culwidth = culwidth-current_node1->width;
        leftcorner=culwidth;
        startco_ord=culwidth;
```

```
        current_max_ht=0;

        for (i=0;i<item;i++){
            if ( (itemtop[i]> current_max_ht)&&
              ( (( leftcorner<itemright[i])&&( leftcorner>itemleft[i]))||
                ((rightcorner<itemright[i])&&(rightcorner>itemleft[i]))||
                ((rightcorner>=itemright[i])&&( leftcorner<=itemleft[i]))))

               current_max_ht=itemtop[i];
        }

        itemheight=current_node1->height + current_max_ht;

        itembottom[item]=current_max_ht;
        itemleft[item]=leftcorner;
        itemright[item]=rightcorner;
        itemtop[item]=itemheight;

        current_node1=current_node1->link;

        deleted_node=DeleteNode(rootp, delnum);
        deleted_node->vposition=band;
        deleted_node->hposition=count;
        deleted_node->floornum=item;

        HorInsertNode(&current_node2, deleted_node);

        item++;
        count++;
        succeed=1;

    }
    else{

        current_node1=current_node1->link;
        delnum++;
    }

}

if (succeed==0)
    numfail++;
else{
    numfail=0;
}
```

```
    band++;

}

*rootp=current_node2;
spacecount=0;
current_node1=*rootp;
current_node2=NULL;

while (current_node1!=NULL){

    if ((current_node1->vposition % 2) == 0){

        current_max_right=hbinsize;

        for (i=0;i<item;i++){
            if (i!=current_node1->floornum){
                if ((itemleft   [i] >=itemright[current_node1->floornum]) &&
                    (itemtop    [i] >=itemtop[current_node1->floornum]) &&
                    (itembottom [i]    < itemtop[current_node1->floornum]) &&
                    (itemleft   [i] < current_max_right) )

                    current_max_right=itemleft[i];

            }

        }

        for (i=0;i<spacecount;i++){
            if (i!=current_node1->floornum){
                if ((spaceleft   [i] >=itemright[current_node1->floornum]) &&
                    (spacetop    [i] >=itemtop[current_node1->floornum]) &&
                    (spacebottom [i] < itemtop[current_node1->floornum]) &&
                    (spaceleft   [i] < current_max_right) )

                    current_max_right=spaceleft[i];
            }
        }

        current_min_depth=0;

        for (i=0;i<item;i++){

            if (i!=current_node1->floornum){

                if ((itemtop   [i] <= itemtop[current_node1->floornum]) &&
```

```
            (itemleft  [i] <  current_max_right) &&
            (itemright [i] >= current_max_right) &&
            (itemtop   [i] >  current_min_depth) ){

                current_min_depth= itemtop[i];

        }

    }

}

for (i=0;i<spacecount;i++){

    if (i!=current_node1->floornum){

        if ((spacetop   [i] <= itemtop[current_node1->floornum]) &&
            (spaceleft  [i] <  current_max_right) &&
            (spaceright [i] >= current_max_right) &&
            (spacetop   [i] >  current_min_depth) ){

                current_min_depth=spacetop[i];

        }
    }
}


current_max_left=0;
current_top=0;

for (i=0;i<item;i++){

    if (i!=current_node1->floornum){

        if ((itemtop   [i] >= itemtop[current_node1->floornum]) &&
            (itembottom[i] <= itembottom[current_node1->floornum])&&
            (itemleft  [i] ==  itemright[current_node1->floornum]) ){

            current_max_left= itemleft[i];
            break;

        }

        if ((itemtop   [i] >  current_min_depth) &&
            (itembottom[i] <= current_min_depth) &&
```

```
            (itemtop   [i] <  itemtop[current_node1->floornum])&&
            (itemright [i] >  itemright[current_node1->floornum]) &&
            (itemright [i] <  current_max_right) &&
            (itemright [i] >  current_max_left)){

            current_max_left= itemright[i];
            if (itemtop[i]>current_top)
                current_top = itemtop[i];
        }

    }

}

for (i=0;i<spacecount;i++){

    if (i!=current_node1->floornum){

        if ((spacetop   [i] >= itemtop[current_node1->floornum]) &&
            (spacebottom[i] <= itembottom[current_node1->floornum])&&
            (spaceleft  [i] ==  itemright[current_node1->floornum]) ){

            current_max_left= spaceleft[i];
            break;

        }

        if ((spacetop   [i] >  current_min_depth) &&
            (spacebottom[i] <= current_min_depth) &&
            (spacetop   [i] <  itemtop[current_node1->floornum])&&
            (spaceright [i] >  itemright[current_node1->floornum]) &&
            (spaceright [i] <  current_max_right) &&
            (spaceright [i] >  current_max_left)){

            current_max_left= spaceright[i];
            if (spacetop[i]>current_top)
                current_top  = spacetop[i];
        }

    }

}

if (current_max_left<= itemright[current_node1->floornum]) {

    new_node=CopyNode(current_node1);
```

```
    new_node->width = current_max_right - itemright[current_node1-
                                >floornum];
    new_node->height = itemtop[current_node1->floornum]-current_min_depth;
    new_node->area = new_node->height * new_node->width;
    new_node->floornum=spacecount;

    spacetop[spacecount]    = itemtop[current_node1->floornum];
    spacebottom[spacecount] = current_min_depth;
    spaceleft[spacecount]     = itemright[current_node1->floornum];
    spaceright[spacecount]  = current_max_right;

    spacecount++;

    HorInsertOrderNode(&current_node2, new_node);

}
else {

    /*first space*/
    new_node1=CopyNode(current_node1);
    new_node1->width = current_max_right - current_max_left;
    new_node1->height = itemtop[current_node1->floornum]-
                                current_min_depth;
    new_node1->area = new_node1->height * new_node1->width;

    if (new_node1->area!=0){
        new_node1->floornum = spacecount;

        spacetop[spacecount]    = itemtop[current_node1->floornum];
        spacebottom[spacecount] = current_min_depth;
        spaceleft[spacecount]     = current_max_left;
        spaceright[spacecount]  = current_max_right;

        spacecount++;
        HorInsertOrderNode(&current_node2, new_node1);
    }
    /*second space*/
    new_node2=CopyNode(current_node1);
    new_node2->width = current_max_left - itemright[current_node1-
                                >floornum];
    new_node2->height = itemtop[current_node1->floornum]-current_top;
    new_node2->area = new_node2->height * new_node2->width;

    if (new_node2->area!=0){
        new_node2->floornum = spacecount;
```

```
            spacetop[spacecount]    = itemtop[current_node1->floornum];
            spacebottom[spacecount] = current_top;
            spaceleft[spacecount]     = itemright[current_node1->floornum];
            spaceright[spacecount]  = current_max_left;
            spacecount++;
            HorInsertOrderNode(&current_node2, new_node2);
        }
    }

}
else{

    current_min_left=0;

    for (i=0;i<item;i++){
        if (i!=current_node1->floornum){
            if ((itemright  [i] <=itemleft[current_node1->floornum]) &&
                (itemtop    [i] >=itemtop[current_node1->floornum]) &&
                (itembottom [i]    < itemtop[current_node1->floornum]) &&
                (itemright   [i] > current_min_left) )

                current_min_left=itemright[i];

        }

    }


    for (i=0;i<spacecount;i++){
        if (i!=current_node1->floornum){
            if ((spaceright  [i] <=itemleft[current_node1->floornum]) &&
                (spacetop    [i] >=itemtop[current_node1->floornum]) &&
                (spacebottom [i]  < itemtop[current_node1->floornum]) &&
                (spaceright   [i] > current_min_left) )

                current_min_left=spaceright[i];
        }

    }

    current_min_depth=0;

    for (i=0;i<item;i++){
        if (i!=current_node1->floornum){
            if ((itemtop   [i] <= itemtop[current_node1->floornum]) &&
                (itemright  [i]>  current_min_left) &&
```

```
            (itemleft   [i]<= current_min_left) &&
            (itemtop   [i] >  current_min_depth) )

                current_min_depth=itemtop[i];

    }
}

for (i=0;i<spacecount;i++){
    if (i!=current_node1->floornum){
        if ((spacetop   [i] <= itemtop[current_node1->floornum]) &&
            (spaceright  [i]>  current_min_left) &&
            (spaceleft   [i]<= current_min_left) &&
            (spacetop   [i] >  current_min_depth) )

                current_min_depth=spacetop[i];
    }
}

current_min_right=hbinsize;
current_top=0;

for (i=0;i<item;i++){
    if (i!=current_node1->floornum){

        if ((itemtop   [i] >= itemtop[current_node1->floornum]) &&
            (itembottom[i] <= itembottom[current_node1->floornum])&&
            (itemright [i] ==  itemleft[current_node1->floornum]) ){

            current_min_right= itemright[i];
            break;

        }

        if ((itemtop   [i] >   current_min_depth) &&
            (itembottom[i] <=  current_min_depth) &&
            (itemtop   [i] <  itemtop[current_node1->floornum])&&
            (itemleft  [i] <  itemleft[current_node1->floornum]) &&
            (itemleft  [i] >  current_min_left)&&
            (itemleft  [i] <  current_min_right)){

            current_min_right= itemleft[i];
            if (itemtop[i]>current_top)
                current_top = itemtop[i];

        }
```

```
    }
}

for (i=0;i<spacecount;i++){
    if (i!=current_node1->floornum){

        if ((spacetop   [i] >= itemtop[current_node1->floornum]) &&
            (spacebottom[i] <= itembottom[current_node1->floornum])&&
            (spaceright [i] ==  itemleft[current_node1->floornum]) ){

            current_min_right= spaceright[i];
            break;

        }

        if ((spacetop   [i] >   current_min_depth) &&
            (spacebottom[i] <=  current_min_depth) &&
            (spacetop   [i] <  itemtop[current_node1->floornum])&&
            (spaceleft  [i] <  itemleft[current_node1->floornum]) &&
            (spaceleft  [i] >  current_min_left)&&
            (spaceleft  [i] <  current_min_right)){

            current_min_right= spaceleft[i];
            if (spacetop[i]>current_top)
                current_top  = spacetop[i];

        }

    }
}

if (current_min_right >= itemleft[current_node1->floornum]) {
    new_node=CopyNode(current_node1);
    new_node->width = itemleft[current_node1->floornum] - current_min_left;
    new_node->height = itemtop[current_node1->floornum]-current_min_depth;
    new_node->area = new_node->height * new_node->width;
    new_node->floornum=spacecount;
    spacetop[spacecount]    = itemtop[current_node1->floornum];
    spacebottom[spacecount] = current_min_depth;
    spaceright[spacecount]  = itemleft[current_node1->floornum];
    spaceleft[spacecount]  = current_min_left;

    spacecount++;

    HorInsertOrderNode(&current_node2, new_node);
```

```
        }
        else {

            /*first space*/
            new_node1=CopyNode(current_node1);
            new_node1->width =  current_min_right - current_min_left;
            new_node1->height = itemtop[current_node1->floornum]-
                                    current_min_depth;
            new_node1->area = new_node1->height * new_node1->width;
            new_node1->floornum = spacecount;
            spacetop[spacecount]    = itemtop[current_node1->floornum];
            spacebottom[spacecount] = current_min_depth;
            spaceright[spacecount]  = current_min_right;
            spaceleft[spacecount]  = current_min_left;

            spacecount++;
            HorInsertOrderNode(&current_node2, new_node1);

            /*second space*/
            new_node2=CopyNode(current_node1);
            new_node2->width = itemleft[current_node1->floornum] –
                                current_min_right;
            new_node2->height = itemtop[current_node1->floornum]-current_top;
            new_node2->area = new_node2->height * new_node2->width;
            new_node2->floornum = spacecount;
            spacetop[spacecount]    = itemtop[current_node1->floornum];
            spacebottom[spacecount] = current_top;
            spaceright[spacecount] = itemleft[current_node1->floornum];
            spaceleft[spacecount]  = current_min_right;
            spacecount++;

            HorInsertOrderNode(&current_node2, new_node2);
        }

    }

    current_node1=current_node1->link;

}


temp_node1=*rootp;
while (temp_node1!= NULL){
    if (temp_node1->vposition>maxband)
        maxband = temp_node1->vposition;
```

```
        temp_node1=temp_node1->link;

    }
temp_node1=*rootp;


    /*item/space j cover item i*/
    for (i=0;i<item;i++){

        itemcover=0;
        spacecover=0;
        for (j=0;j<item;j++){
            if (i!=j) {
                if( ( (itembottom[j]>= itemtop[i])&&
                    ( (( itemleft[i]< itemright[j])&&( itemleft[i]>   itemleft[j])) ||
                      ((itemright[i]< itemright[j])&&(itemright[i]>   itemleft[j])) ||
                      ((itemright[i]>=itemright[j])&&( itemleft[i]<=  itemleft[j])) ) ) ){

                    itemcover=1;
                }
            }

        }

        for (j=0;j<spacecount;j++){
            if (i!=j) {
                if( ( (spacebottom[j]>= itemtop[i])&&
                    ( (( itemleft[i]< spaceright[j])&&( itemleft[i]>   spaceleft[j])) ||
                    ((itemright[i]< spaceright[j])&&(itemright[i]>   spaceleft[j])) ||
                    ((itemright[i]>=spaceright[j])&&( itemleft[i]<=  spaceleft[j])))  )){

                    spacecover=1;
                }

              }

        }

        if ((itemcover==0)&&(spacecover==0)){
                new_node=CreatNode((vbinsize-itemtop[i]),(itemright[i]-itemleft[i]));
                new_node->area = new_node->height * new_node->width;
                new_node->binnum= temp_node1->binnum;
                new_node->vposition=maxband+1;
                new_node->ceilnum=1;
                new_node->floornum=spacecount;
                HorInsertOrderNode(&current_node2, new_node);
```

```
                spacetop[spacecount]    =vbinsize;
                spacebottom[spacecount] =itemtop[i];
                spaceright[spacecount]  = itemright[i];
                spaceleft[spacecount]   =itemleft[i] ;

                spacecount++;

            }

        }

    /*item/space j cover space i*/
        for (i=0;i<spacecount;i++){

            itemcover=0;
            spacecover=0;
            for (j=0;j<item;j++){
                if (i!=j) {
                    if( ( (itembottom[i]>= spacetop[j])&&
                        ( (( itemleft[i]< spaceright[j])&&( itemleft[i]>   spaceleft[j])) ||
                          ((itemright[i]< spaceright[j])&&(itemright[i]>   spaceleft[j])) ||
                          ((itemright[i]>=spaceright[j])&&( itemleft[i]<=  spaceleft[j])) ) ) ){

                        itemcover=1;
                    }
                }

            }

            for (j=0;j<spacecount;j++){
                if (i!=j) {
                    if( ( (spacebottom[j]>= spacetop[i])&&
                        ( (( spaceleft[i]< spaceright[j])&&( spaceleft[i]>   spaceleft[j])) ||
                        ((spaceright[i]< spaceright[j])&&(spaceright[i]>   spaceleft[j])) ||
                        ((spaceright[i]>=spaceright[j])&&( spaceleft[i]<=  spaceleft[j])))  )){

                        spacecover=1;
                    }

                }

            }

            if ((itemcover==0)&&(spacecover==0)){
                    new_node=CreatNode((vbinsize-spacetop[i]),(spaceright[i]-spaceleft[i]));
```

```
                new_node->area = new_node->height * new_node->width;
                new_node->binnum= temp_node1->binnum;
                new_node->vposition=maxband+1;
                new_node->ceilnum=1;
                new_node->floornum=spacecount;

                HorInsertOrderNode(&current_node2, new_node);

                spacetop[spacecount]    =vbinsize;
                spacebottom[spacecount] =spacetop[i];
                spaceright[spacecount]  = spaceright[i];
                spaceleft[spacecount]   =spaceleft[i] ;

                spacecount++;

        }

    }


    space[current_node2->binnum] = current_node2;
}


/*swapping items between bins, and repacking using alternate direction algorithm*/

int fswap22(Node ** rootp1, Node **rootp2, int i, int p, double weight[] )
{

    double deltaf, deltaf1, deltaf2, deltaf3;

    int m=0;
    int n=0;
    int j=0;
    int k=0;
    int q=0;
    int r=0;

    int height1=0;
    int height2=0;
    int count=0;


    Node * deleted_node1;
    Node * deleted_node2;
    Node * deleted_node3;
```

```
Node * deleted_node4;
Node * current_node1;
Node * current_node2;
Node * current_node3;
Node * current_node4;

Node * duplicate1=NULL;
Node * duplicate2=NULL;
Node * temp_node;

deltaf=0.0;
deltaf1=0.0;
deltaf2=0.0;
deltaf3=0.0;

current_node1=*rootp1;
current_node2=*rootp2;
while (current_node1!=NULL){
    temp_node=CopyNode(current_node1);
    HorInsertNode(&duplicate1, temp_node);
    current_node1=current_node1->link;
}
while (current_node2!=NULL){
    temp_node=CopyNode(current_node2);
    HorInsertNode(&duplicate2, temp_node);
    current_node2=current_node2->link;
}
current_node1=*rootp1;
current_node2=*rootp2;

for(m=0;m<(vernitembin[i]-1);m++){

    k=j+1;
    current_node2=current_node1->link;

    while (current_node2 != NULL){

        q=0;
        current_node3=*rootp2;

        for(n=0;n<(vernitembin[p]-1);n++){

            r=q+1;
            current_node4=current_node3->link;

            while (current_node4!=NULL){
```

```
if ( ((vbinarea[i]- current_node1->area - current_node2->area+
        current_node3->area+ current_node4->area) <=binarea)&&
     ((vbinarea[p]- current_node3->area - current_node4->area+
        current_node1->area+ current_node2->area) <=binarea) ) {

    deleted_node2= DeleteNode(&duplicate1,k);
    deleted_node4= DeleteNode(&duplicate2,r);
    deleted_node1= DeleteNode(&duplicate1,j);
    deleted_node3= DeleteNode(&duplicate2,q);

    deleted_node1->tag=1;
    deleted_node2->tag=1;
    deleted_node3->tag=1;
    deleted_node4->tag=1;

    HorInsertNode(&duplicate1,deleted_node3);
    HorInsertNode(&duplicate1,deleted_node4);
    HorInsertNode(&duplicate2,deleted_node1);
    HorInsertNode(&duplicate2,deleted_node2);

    height1=alterdirection(&duplicate1);

    if  (height1<=vbinsize){

        height2=alterdirection(&duplicate2);

        if(height2<=vbinsize){

            deltaf3=SQ(1.0* (vbinarea[i]- current_node1->area –
                current_node2->area+current_node3->area+
                current_node4->area)*weight[i]*weight[i])+
                SQ(1.0*(vbinarea[p]- current_node3->area - current_node4-
                >area+ current_node1->area+ current_node2-
                >area)*weight[p]*weight[p])-SQ(1.0*(vbinarea[i]*
                weight[i]*weight[i]))-SQ(1.0*(vbinarea[p]*weight[p]
                *weight[p]));

            if (deltaf>=0){
                vbinarea[i]=vbinarea[i]-(current_node1->area)-(current_node2
                    ->area)+(current_node3->area)+(current_node4->area);
                vbinarea[p]=vbinarea[p]-(current_node3->area)-
                    (current_node4->area)+(current_node1->area)+
                    (current_node2->area);

                deleted_node2= DeleteNode(rootp1,k);
```

```
                deleted_node4= DeleteNode(rootp2,r);
                deleted_node1= DeleteNode(rootp1,j);
                deleted_node3= DeleteNode(rootp2,q);

                deleted_node1->binnum=p;
                deleted_node2->binnum=p;
                deleted_node3->binnum=i;
                deleted_node4->binnum=i;

                HorInsertNode(rootp1,deleted_node3);
                HorInsertNode(rootp1,deleted_node4);
                HorInsertNode(rootp2,deleted_node1);
                HorInsertNode(rootp2,deleted_node2);

                stackheight[i]=alterdirection(rootp1);
                stackheight[p]=alterdirection(rootp2);

                /*update curcap after swaps*/
                count=0;
                for (i=0;i<vnumbin ; i++){
                    vareacap[i]= binarea-vbinarea[i];
                    if (vbinarea[i] > 0)
                        count++;
                }
                if (count==lowerbound)
                    done = 1;
                for (; duplicate1!=NULL; duplicate1=temp_node){
                    temp_node=duplicate1->link;
                    free(duplicate1);
                }

                for (; duplicate2!=NULL; duplicate2=temp_node){
                    temp_node=duplicate2->link;
                    free(duplicate2);
                }

                return TRUE;
            }

        }

    }
deleted_node1= DeleteTagNode(&duplicate1);
deleted_node2= DeleteTagNode(&duplicate1);
deleted_node3= DeleteTagNode(&duplicate2);
deleted_node4= DeleteTagNode(&duplicate2);
```

```
                    deleted_node1->tag=0;
                    deleted_node2->tag=0;
                    deleted_node3->tag=0;
                    deleted_node4->tag=0;
                    HorInsertNode(&duplicate1,deleted_node3);
                    HorInsertNode(&duplicate1,deleted_node4);
                    HorInsertNode(&duplicate2,deleted_node2);
                    HorInsertNode(&duplicate2,deleted_node1);

                }
                r++;
                current_node4=current_node4->link;
            }
            q++;
            current_node3=current_node3->link;
        }
        k++;
        current_node2=current_node2->link;
    }
    j++;
    current_node1=current_node1->link;

}

for (; duplicate1!=NULL; duplicate1=temp_node){
    temp_node=duplicate1->link;
    free(duplicate1);
}

for (; duplicate2!=NULL; duplicate2=temp_node){
    temp_node=duplicate2->link;
    free(duplicate2);
}


return FALSE;
}


/*swapping items between bins, and repacking using alternate direction algorithm*/
int fswap12(Node ** rootp1, Node **rootp2, int i, int p, double weight[] )
{

    double deltaf, deltaf1, deltaf2, deltaf3;
    int n=0;
    int j=0;
```

```
int q=0;
int r=0;
int height1=0;
int height2=0;
int count=0;

Node * deleted_node1;
Node * deleted_node2;
Node * deleted_node3;
Node * current_node1;
Node * current_node2;
Node * current_node3;
Node * duplicate1=NULL;
Node * duplicate2=NULL;
Node * temp_node;

deltaf=0.0;
deltaf1=0.0;
deltaf2=0.0;
deltaf3=0.0;

current_node1=*rootp1;
current_node2=*rootp2;
while (current_node1!=NULL){
    temp_node=CopyNode(current_node1);
    HorInsertNode(&duplicate1, temp_node);
    current_node1=current_node1->link;
}
while (current_node2!=NULL){
    temp_node=CopyNode(current_node2);
    HorInsertNode(&duplicate2, temp_node);
    current_node2=current_node2->link;
}
current_node1=*rootp1;
current_node2=*rootp2;

while (current_node1!=NULL){

    q=0;
    current_node2=*rootp2;

    for(n=0;n<(vernitembin[p]-1);n++){

            r=q+1;
            current_node3=current_node2->link;
```

```
while (current_node3!=NULL){


    if ( ((vbinarea[i]- current_node1->area + current_node2->area+
            current_node3->area) <=binarea)&&((vbinarea[p]- current_node2-
            >area -current_node3->area+ current_node1->area) <=binarea) ) {

        deleted_node3= DeleteNode(&duplicate2,r);
        deleted_node1= DeleteNode(&duplicate1,j);
        deleted_node2= DeleteNode(&duplicate2,q);

        deleted_node1->tag=1;
        deleted_node2->tag=1;
        deleted_node3->tag=1;

        HorInsertNode(&duplicate1,deleted_node2);
        HorInsertNode(&duplicate1,deleted_node3);
        HorInsertNode(&duplicate2,deleted_node1);

        height1=alterdirection(&duplicate1);

        if (height1<=vbinsize){

            height2=alterdirection(&duplicate2);

            if (height2<=vbinsize){

                deltaf=SQ(1.0* (vbinarea[i]- current_node1->area +
                    current_node2->area+ current_node3->area)*weight[i]
                    *weight[i])+ SQ(1.0*(vbinarea[p]-current_node2->area
                     - current_node3->area+ current_node1->area)
                    *weight[p]*weight[p])- SQ(1.0*(vbinarea[i]*weight[i]
                    *weight[i]))-SQ(1.0*(vbinarea[p]*weight[p]*weight[p]));

                if (deltaf>=0){

                    vbinarea[i]=vbinarea[i]-(current_node1->area)+
                            (current_node2->area)+(current_node3->area);
                    vbinarea[p]=vbinarea[p]-(current_node2->area)-
                            (current_node3->area)+(current_node1->area);

                    deleted_node3= DeleteNode(rootp2,r);
                    deleted_node1= DeleteNode(rootp1,j);
                    deleted_node2= DeleteNode(rootp2,q);

                    deleted_node1->binnum=p;
```

```
            deleted_node2->binnum=i;
            deleted_node3->binnum=i;

            HorInsertNode(rootp1,deleted_node2);
            HorInsertNode(rootp1,deleted_node3);
            HorInsertNode(rootp2,deleted_node1);

            /*update number of items in bins*/
            vernitembin[p]--;
            vernitembin[i]++;

            stackheight[i]=alterdirection(rootp1);
            stackheight[p]=alterdirection(rootp2);

            /*update curcap after swaps*/
            count=0;
            for (i=0;i<vnumbin ; i++){
                vareacap[i]= binarea-vbinarea[i];
                if (vbinarea[i] > 0)
                    count++;
            }

            if (count==lowerbound)
                done=1;
            for (; duplicate1!=NULL; duplicate1=temp_node){
                temp_node=duplicate1->link;
                free(duplicate1);
            }

            for (; duplicate2!=NULL; duplicate2=temp_node){
                temp_node=duplicate2->link;
                free(duplicate2);
            }

            return TRUE;
        }

    }

}

deleted_node1= DeleteTagNode(&duplicate1);
deleted_node2= DeleteTagNode(&duplicate1);
deleted_node3= DeleteTagNode(&duplicate2);
deleted_node1->tag=0;
deleted_node2->tag=0;
```

```
                    deleted_node3->tag=0;
                    HorInsertNode(&duplicate1,deleted_node3);
                    HorInsertNode(&duplicate2,deleted_node2);
                    HorInsertNode(&duplicate2,deleted_node1);

                }

                r++;
                current_node3=current_node3->link;

            }

        q++;
        current_node2=current_node2->link;

    }

    j++;
    current_node1=current_node1->link;
}

for (; duplicate1!=NULL; duplicate1=temp_node){
    temp_node=duplicate1->link;
    free(duplicate1);
}

for (; duplicate2!=NULL; duplicate2=temp_node){
    temp_node=duplicate2->link;
    free(duplicate2);
}

return FALSE;
}

/*swapping items between bins, and repacking using alternate direction algorithm*/
int fswap11(Node ** rootp1, Node **rootp2, int i, int p, double weight[])
{

    double deltaf, deltaf1, deltaf2, deltaf3;
    int j=0;
    int q=0;
    int height1=0;
    int height2=0;
    int count=0;

    Node * deleted_node1;
```

```
Node * deleted_node2;
Node * current_node1;
Node * current_node2;
Node * duplicate1=NULL;
Node * duplicate2=NULL;
Node * temp_node;

deltaf=0.0;
deltaf1=0.0;
deltaf2=0.0;
deltaf3=0.0;

current_node1=*rootp1;
current_node2=*rootp2;
while (current_node1!=NULL){
    temp_node=CopyNode(current_node1);
    HorInsertNode(&duplicate1, temp_node);
    current_node1=current_node1->link;
}
while (current_node2!=NULL){
    temp_node=CopyNode(current_node2);
    HorInsertNode(&duplicate2, temp_node);
    current_node2=current_node2->link;
}
current_node1=*rootp1;
current_node2=*rootp2;

while (current_node1!=NULL){

    q=0;
    current_node2=*rootp2;

    while (current_node2!=NULL){

        if ( ((vbinarea[i]- current_node1->area + current_node2->area) <= binarea)&&
            ((vbinarea[p]- current_node2->area + current_node1->area) <= binarea) ) {

            deleted_node1= DeleteNode(&duplicate1,j);
            deleted_node2= DeleteNode(&duplicate2,q);
            deleted_node1->tag=1;
            deleted_node2->tag=1;
            HorInsertNode(&duplicate1,deleted_node2);
            HorInsertNode(&duplicate2,deleted_node1);

            height1=alterdirection(&duplicate1);
```

```
if (height1<=vbinsize){

    height2=alterdirection(&duplicate2);

    if (height2<=vbinsize){

        deltaf=SQ(1.0* (vbinarea[i]- current_node1->area + current_node2-
            >area)*weight[i]*weight[i])+ SQ(1.0*(vbinarea[p]- current_node2->
            area  + current_node1->area)*weight[p]*weight[p])-
            SQ(1.0*(vbinarea[i]*weight[i]*weight[i]))-
            SQ(1.0*(vbinarea[p]*weight[p]*weight[p]));

        if (deltaf>=0){

            vbinarea[i]=vbinarea[i]-(current_node1->area) +(current_node2-
                    >area);
            vbinarea[p]=vbinarea[p]-(current_node2->area) +(current_node1-
                    >area);

            deleted_node1= DeleteNode(rootp1,j);
            deleted_node2= DeleteNode(rootp2,q);

            deleted_node1->binnum=p;
            deleted_node2->binnum=i;

            HorInsertNode(rootp1,deleted_node2);
            HorInsertNode(rootp2,deleted_node1);

            stackheight[i]=alterdirection(rootp1);
            stackheight[p]=alterdirection(rootp2);

            /*update curcap after swaps*/
            count=0;
            for (i=0;i<vnumbin ; i++){
                vareacap[i]= binarea-vbinarea[i];
                if (vbinarea[i] > 0)
                    count++;
            }

            if (count==lowerbound)
                done=1;

            for (; duplicate1!=NULL; duplicate1=temp_node){
                temp_node=duplicate1->link;
                free(duplicate1);
            }
```

```
                        for (; duplicate2!=NULL; duplicate2=temp_node){
                            temp_node=duplicate2->link;
                            free(duplicate2);
                        }

                        return TRUE;

                    }

                }

            }
            deleted_node1= DeleteTagNode(&duplicate1);
            deleted_node2= DeleteTagNode(&duplicate2);
            deleted_node1->tag=0;
            deleted_node2->tag=0;
            HorInsertNode(&duplicate1,deleted_node2);
            HorInsertNode(&duplicate2,deleted_node1);

        }

        q++;
        current_node2=current_node2->link;
    }
    j++;
    current_node1=current_node1->link;
}

for (; duplicate1!=NULL; duplicate1=temp_node){
    temp_node=duplicate1->link;
    free(duplicate1);
}

for (; duplicate2!=NULL; duplicate2=temp_node){
    temp_node=duplicate2->link;
    free(duplicate2);
}

return FALSE;
}

/*swapping items between bins, and repacking using alternate direction algorithm*/
int fswap10(Node ** rootp1, Node **rootp2, int i, int p, double weight[])
{
```

```
double deltaf, deltaf1, deltaf2, deltaf3;
int j=0;
int q=0;
int height1=0;
int height2=0;
int count=0;

Node * deleted_node1;
Node * current_node1;
Node * current_node2;
Node * duplicate1=NULL;
Node * duplicate2=NULL;
Node * temp_node;

deltaf=0.0;
deltaf1=0.0;
deltaf2=0.0;
deltaf3=0.0;

current_node1=*rootp1;
current_node2=*rootp2;
while (current_node1!=NULL){
    temp_node=CopyNode(current_node1);
    HorInsertNode(&duplicate1, temp_node);
    current_node1=current_node1->link;
}
while (current_node2!=NULL){
    temp_node=CopyNode(current_node2);
    HorInsertNode(&duplicate2, temp_node);
    current_node2=current_node2->link;
}
current_node1=*rootp1;
current_node2=*rootp2;

while (current_node1!=NULL){

        if (( vbinarea[p] + current_node1->area) <= binarea)  {

            deleted_node1= DeleteNode(&duplicate1,j);
            deleted_node1->tag=1;
            HorInsertNode(&duplicate2,deleted_node1);

            height1=alterdirection(&duplicate1);


            if (height1<=vbinsize){
```

```
height2=alterdirection(&duplicate2);

if (height2<=vbinsize){

    deltaf=SQ(1.0* (vbinarea[i]- current_node1->area)*weight[i]*
        weight[i])+ SQ(1.0*(vbinarea[p] + current_node1->area)*
        weight[p]*weight[p])-SQ(1.0*(vbinarea[i]*weight[i]*
        weight[i]))-SQ(1.0*(vbinarea[p]*weight[p]*weight[p]));

    if (deltaf>=0) {

        vbinarea[i]=vbinarea[i] -(current_node1->area);
        vbinarea[p]=vbinarea[p] +(current_node1->area);

        deleted_node1= DeleteNode(rootp1,j);

        deleted_node1->binnum=p;

        HorInsertNode(rootp2,deleted_node1);

        vernitembin[i]--;
        vernitembin[p]++;

        stackheight[i]=alterdirection(rootp1);
        stackheight[p]=alterdirection(rootp2);

        /*update curcap after swaps*/
        count=0;
        for (i=0;i<vnumbin ; i++){
            vareacap[i]= binarea-vbinarea[i];
            if (vbinarea[i] >0)
                count++;
        }
        if (count==lowerbound)
            done=1;

        for (; duplicate1!=NULL; duplicate1=temp_node){
            temp_node=duplicate1->link;
            free(duplicate1);
        }

        for (; duplicate2!=NULL; duplicate2=temp_node){
            temp_node=duplicate2->link;
            free(duplicate2);
        }
```

```
                        return TRUE;

                    }


                }

            }
            deleted_node1= DeleteTagNode(&duplicate2);
            deleted_node1->tag=0;
            HorInsertNode(&duplicate1,deleted_node1);

        }


        j++;
        current_node1=current_node1->link;
    }

for (; duplicate1!=NULL; duplicate1=temp_node){
    temp_node=duplicate1->link;
    free(duplicate1);
}

for (; duplicate2!=NULL; duplicate2=temp_node){
    temp_node=duplicate2->link;
    free(duplicate2);
}


return FALSE;
}

/*alternate direction algorithm*/
int alterdirection(Node ** rootp)
{

    int i,count;
    int item=0;
    int band=0;
    int delnum=0;
    int numfail=0;
    int succeed = 0;
    int culwidth=0;
    int startco_ord = 0;
```

```
int maxheight =0 ;
int current_max_ht =0;
int leftcorner =0;
int rightcorner =0;
int itemheight =0;
int baseheight[MAX_NUMBER_ITEM];
int baseleft  [MAX_NUMBER_ITEM];
int baseright [MAX_NUMBER_ITEM];

Node * current_node1=NULL;
Node * current_node2=NULL;
Node * deleted_node;

for (i=0;i<MAX_NUMBER_ITEM;i++){
    baseheight[i]=0;
    baseleft[i]=0;
    baseright[i]=0;
}

while (numfail<3){

    delnum=0;
    culwidth=0;
    startco_ord=0;
    succeed=0;
    count=0;
    current_node1=*rootp;

    while (current_node1!=NULL){

        if ((startco_ord + current_node1->width) <= hbinsize){

                leftcorner=culwidth;
                culwidth=culwidth+current_node1->width;
                startco_ord=culwidth;
                current_max_ht=0;
                rightcorner=culwidth;

                for (i=0;i<item;i++){
                    if ( (baseheight[i]> current_max_ht)&&
                        ( (( leftcorner<baseright[i])&&( leftcorner >  baseleft[i]))||
                        ((rightcorner<baseright[i])&&(rightcorner >  baseleft[i]))||
                        ((rightcorner>=baseright[i])&&(leftcorner <= baseleft[i])) ))

                        current_max_ht=baseheight[i];
                }
```

```
            itemheight=current_node1->height + current_max_ht;

            if (itemheight> maxheight)
                maxheight= itemheight;

            current_node1=current_node1->link;

            deleted_node=DeleteNode(rootp, delnum);

            deleted_node->vposition=band;
            deleted_node->hposition=count;

            HorInsertNode(&current_node2, deleted_node);

            baseleft[item]=leftcorner;
            baseright[item]=rightcorner;
            baseheight[item]=itemheight;
            item++;
            count++;
            succeed=1;


        }
        else{

            current_node1=current_node1->link;
            delnum++;
        }

    }

    if (succeed==0)
        numfail++;
    else{
        numfail=0;
    }

    band++;

    delnum=0;
    culwidth=hbinsize;
    startco_ord=hbinsize;
    succeed=0;
    count=0;
    current_node1=*rootp;
```

```
while (current_node1!=NULL){

    if (startco_ord - current_node1->width >= 0){

        rightcorner=culwidth;
        culwidth = culwidth-current_node1->width;
        leftcorner=culwidth;
        startco_ord=culwidth;
        current_max_ht=0;

        for (i=0;i<item;i++){
            if ( (baseheight[i]> current_max_ht)&&
              ( (( leftcorner<baseright[i])&&( leftcorner>baseleft[i]))||
                ((rightcorner<baseright[i])&&(rightcorner>baseleft[i]))||
                ((rightcorner>=baseright[i])&&( leftcorner<=baseleft[i]))))

                 current_max_ht=baseheight[i];
        }

        itemheight=current_node1->height + current_max_ht;

        if (itemheight> maxheight)
            maxheight= itemheight;

        current_node1=current_node1->link;
        deleted_node=DeleteNode(rootp, delnum);
        deleted_node->vposition=band;
        deleted_node->hposition=count;

        HorInsertNode(&current_node2, deleted_node);

        baseleft[item]=leftcorner;
        baseright[item]=rightcorner;
        baseheight[item]=itemheight;
        item++;
        count++;
        succeed=1;

    }
    else{

        current_node1=current_node1->link;
        delnum++;
    }
```

```
        }

        if (succeed==0)
            numfail++;
        else{
            numfail=0;
        }

        band++;

    }

    *rootp=current_node2;
    return maxheight;

}


void PrintFileNode(Node * item)
{
    fprintf(op, "%5d %5d %5d %5d %5d %5d %5d %5d %5d %5d\n", item->height,item-
>width,item->stripnum, item->hposition, item->binnum,item->vposition,item-
>floornum,item->ceilnum,item->area, item->tag);

}

void PrintFile(Node * root)
{
    if (root != NULL) {
        while (root != NULL) {
            PrintFileNode(root);
            root = root->link;
        }
    } else {
        fprintf(op, " No nodes have been entered yet!\n");
    }
}


void PrintNode(Node * item)
{
    printf("%5d %5d %5d %5d %5d %5d %5d %5d %5d %5d\n", item->height,item-
>width,item->stripnum,item->hposition,item->binnum,item->vposition,item-
>floornum,item->ceilnum,item->area, item->tag);

}
```

```
void PrintAllNode(Node * root)
{
    if (root != NULL) {
        while (root != NULL) {
            PrintNode(root);
            root = root->link;
        }
    } else {
        printf("No nodes have been entered yet!\n");
    }
}



Node * CopyNode(Node * root)
{
    struct NODE * temp = (struct NODE*) malloc(sizeof(struct NODE));

    temp->height=root->height;
    temp->width=root->width;
    temp->stripnum=root->stripnum;
    temp->binnum=root->binnum;
    temp->hposition=root->hposition;
    temp->vposition=root->vposition;
    temp->ceilnum=root->ceilnum;
    temp->floornum=root->floornum;
    temp->area=root->area;
    temp->tag=root->tag;

    return temp;
}

Node * CreatNode(int high, int wide)
{
    struct NODE * temp = (struct NODE*) malloc(sizeof(struct NODE));

    temp->height=high;
    temp->width=wide;
    temp->stripnum=0;
    temp->binnum=0;
    temp->hposition=0;
    temp->vposition=0;
    temp->ceilnum=0;
    temp->floornum=0;
```

```
    temp->area=0;
    temp->tag=0;

    return temp;
}

Node * AddNode(int high, int wide, int num1, int num2, Node * bin )
{
    struct NODE * temp = (struct NODE*) malloc(sizeof(struct NODE));

    temp->height = high;
    temp->width  = wide;
    temp->stripnum=num1;
    temp->binnum=num2;
    temp->vposition=0;
    temp->ceilnum=0;
    temp->floornum=0;
    temp->area=0;
    temp->link = bin;
    bin = temp;
    return bin;
}

int HorInsertNode( register Node **linkp, Node * inserted_node)
{
    register Node   *current;
    while( ( current = *linkp ) != NULL && current->height >=inserted_node->height){

        if ((current->height == inserted_node->height)&&(current->width <
inserted_node->width))
            break;

        linkp = &current->link;

    }

    inserted_node->link = current;
    *linkp = inserted_node;

    return TRUE;
}


int HorInsertOrderNode( register Node **linkp, Node * inserted_node)
{
    register Node   *current;
```

```
    while( ( current = *linkp ) != NULL && current->floornum < inserted_node-
>floornum){

        linkp = &current->link;

    }

    inserted_node->link = current;
    *linkp = inserted_node;

    return TRUE;
}

int VerInsertNode( register Node **linkp, Node * inserted_node)
{
    register Node   *current;
    while( ( current = *linkp ) != NULL &&
        hbinload[current->stripnum] >hbinload[inserted_node->stripnum])
        linkp = &current->link;

    inserted_node->link = current;
    *linkp = inserted_node;

    return TRUE;
}

Node * DeleteTagNode( Node **linkp)
{
    Node *current;
    Node *previous;
    Node *delnode;

    current = *linkp;
    previous = NULL;

    while((current != NULL) && (current->tag==0)){
        previous = current;
        current = current->link;
    }

    delnode = current;

    if(previous==NULL)
        *linkp=current->link;
    else
        previous->link=delnode->link;
```

```
    if(current!=NULL)
        current = current->link;
    else
        current->link=NULL;

    return  delnode;
}


////////////////////////////////////////////////////////////////////////////////
//          2bp_o_f.h is the header file for WA2BPOF.cpp
//
////////////////////////////////////////////////////////////////////////////////
typedef  struct NODE   {
    struct NODE   *link;
    int height;
    int width;
    int stripnum;
    int binnum;
    int hposition;
    int vposition;
    int floornum;
    int ceilnum;
    int orientation;
    int area;
    int tag;
} Node;

void sort(unsigned long n, double arr[]);
#define SQ(a) (a*a)
#define  FALSE  0
#define TRUE 1
#define MAX_NUMBER_BIN 100
#define MAX_NUMBER_ITEM 150
#define MAX_NUMBER_HOR  20
#define MAX_NUMBER_VER 10
#define RIGHT 0
#define LEFT 1


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

Appendix F

Two-Dimensional Bin Packing Problem Code
(Free Cuts, Non-Oriented Items)

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//*     WA2BPRF.cpp is the program for solving the two-dimensional bin packking
// problem with free cuts and non-oriented items
//*     It calls functions from WA2BPOG.cpp and WA2BPOF.cpp
//*     It is to be complied with the following files (ANSI version) from Press et al.(2002)
//          1)    nrutil.cpp
//          2)    nrutil.h
//          3)    nr.h
//*     The header file 2bp_r_f.h is appended at the end of the program.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////


#include <io.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "nrutil.h"
#include "nr.h"
#include <sys\timeb.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include "2bp_r_f.h"
#include <string.h>

Node * bin_root[MAX_NUMBER_BIN];
Node * vbin_root[MAX_NUMBER_BIN];
Node * hbin_root[MAX_NUMBER_BIN];
Node * vertical_bin[MAX_NUMBER_BIN];
Node * space[MAX_NUMBER_BIN];

Node * obj_size;
Node * strip_size;

FILE * fp;
FILE * op;
```

```
int binload[MAX_NUMBER_BIN];
int vbinload[MAX_NUMBER_BIN];
int hbinload[MAX_NUMBER_BIN];
int vbinarea[MAX_NUMBER_BIN];
int vareacap[MAX_NUMBER_BIN];
int hbinarea[MAX_NUMBER_BIN];
int hareacap[MAX_NUMBER_BIN];
int hmaxht[MAX_NUMBER_BIN];
int hstripsize[MAX_NUMBER_BIN];
int curcap[MAX_NUMBER_BIN];
int vcurcap[MAX_NUMBER_BIN];
int hcurcap[MAX_NUMBER_BIN];
int nitembin[MAX_NUMBER_BIN];
int vnitembin[MAX_NUMBER_BIN];
int vernitembin[MAX_NUMBER_BIN];
int hnitembin[MAX_NUMBER_BIN];
int numcell[MAX_NUMBER_BIN];
int stackheight[MAX_NUMBER_BIN];
double evareacap[MAX_NUMBER_BIN];

int ceilheight [MAX_NUMBER_BIN][MAX_NUMBER_HOR];
int ceilwidth  [MAX_NUMBER_BIN][MAX_NUMBER_HOR];
int ceilcap    [MAX_NUMBER_BIN][MAX_NUMBER_HOR];
int floorheight[MAX_NUMBER_BIN][MAX_NUMBER_HOR];
int floorwidth [MAX_NUMBER_BIN][MAX_NUMBER_HOR];
int floorcap   [MAX_NUMBER_BIN][MAX_NUMBER_HOR];

int binsize=0;
int bound=0;
int vbinsize=0;
int hbinsize=0;
int binarea=0;

int vnumbin=30;
int maxhnumbin=100;
int hnumbin=0;

double comptime=0.0;
double totaltime=0.0;
double timelimit=100.0;
double ebinarea=0.0;
int done =0;
int loop =0;

double TimeElapsed(struct _timeb *begin, struct _timeb *end);
int irbit1(unsigned long *iseed);
```

```
void PrintNode(Node * item);
void PrintAllNode(Node * root);
void PrintFileNode(Node * item);
void PrintFile(Node * root);
void removebin (Node **linkp);
void UpdateHposition(Node **rootp);
void UpdateVposition(Node **rootp);

Node * AddNode(int high, int wide, int num1, int num2, Node * bin);
Node * CreatNode(int high, int wide);
Node * CopyNode(Node * root);
Node * DeleteNode( Node **linkp, int item_num );
Node * DeleteTagNode( Node **linkp);
int HorInsertNode( register Node **linkp, Node * inserted_node);
int VerInsertNode( register Node **linkp, Node * inserted_node);
int HorInsertOrderNode( register Node **linkp, Node * inserted_node);

int CompVbinLoad(Node * root);
int CompHbinLoad(Node * root);
int CompBinItem(Node * root);

int vswap10(Node ** rootp1, Node **rootp2, int x, int y, double vweight[]);
int vswap11(Node ** rootp1, Node **rootp2, int x, int y, double vweight[]);
int vswap12(Node ** rootp1, Node **rootp2, int x, int y, double vweight[]);
int vswap22(Node ** rootp1, Node **rootp2, int x, int y, double vweight[]);

int hswap10(Node ** rootp1, Node **rootp2, Node * temp_node1, Node * temp_node2,
        int x, int y, int xbin, int ybin, double hweight[]);
int hswap11(Node ** rootp1, Node **rootp2, Node * temp_node1, Node * temp_node2,
        int x, int y, int xbin, int ybin, double hweight[]);
int hswap12(Node ** rootp1, Node **rootp2, Node * temp_node1, Node * temp_node2,
        int x, int y, int xbin, int ybin, double hweight[]);
int hswap22(Node ** rootp1, Node **rootp2, Node * temp_node1, Node * temp_node2,
        int x, int y, int xbin, int ybin, double hweight[]);
int levelpack(Node ** rootp1, Node **rootp2, int s, int t, double hweight[]);

int alterdirection(Node ** rootp);

int fswap10(Node ** rootp1, Node **rootp2, int i, int p, double weight[]);
int fswap11(Node ** rootp1, Node **rootp2, int i, int p, double weight[]);
int fswap12(Node ** rootp1, Node **rootp2, int i, int p, double weight[]);
int fswap22(Node ** rootp1, Node **rootp2, int i, int p, double weight[]);

void ComputeSpace(Node ** rootp);

double binpara;
```

```
int lowerbound=15;
int
main(void)
{
    struct _timeb time1, time2;
    unsigned long iseed;
    Node * deleted_node;
    Node * deleted_node1;
    Node * deleted_node2;
    Node * inserted_node;
    Node * current_node;
    Node * temp_node;
    Node * temp_node1;
    Node * temp_node2;
    int i,j,k,idec,p,q,m,n,s,preht;
    int numbitem=0;
    int improved=0;
    int feasible=0;
    int numtype=0;
    int temp1=0;
    int temp2=0;
    int numstrip=0;
    int empty=0;
    int data=0;
    int count=0;
    int count1=0;
    int count2=0;
    int binheight=0;
    int culwidth=0;
    int maxind=0;
    int maxsize=0;
    int minind=0;
    int mincap=0;
    int rotated=0;
    double maxcap=0;
    double hweight [MAX_NUMBER_BIN];
    double vweight [MAX_NUMBER_BIN];
    double bweight [MAX_NUMBER_BIN];
    int height[MAX_NUMBER_ITEM];
    int width[MAX_NUMBER_ITEM];
    double T = 1.0;
    double Tred =0.95;
    float temp=0.0;
    char str[10];
    done=0;
```

```
    iseed=111;

    if((op=fopen("out.txt", "w")) == NULL) {
        printf("Cannot open file.\n");
        exit(1);
    }

    /* Input file format – Lodi, Martello and Toth (1999)*/
    if((fp=fopen( "10_100_10.txt", "r")) == NULL)
    {       printf("Cannot open file.\n");
        exit(1);
    }

    fscanf(fp, "%d %s %s", &data, &str, &str);
    fscanf(fp, "%d %s %s %s", &numbitem, &str, &str, &str);
    fscanf(fp, "%d %d %s %s %s %s %s %s", &data, &data, &str,
              &str, &str, &str, &str, &str);
    fscanf(fp, "%d %d %s", &vbinsize, &hbinsize, &str);

    for(i =0; i <numbitem;i++){
        if (i==0)
            fscanf(fp, "%d  %d %s", &height[i], &width[i], &str);
        else
            fscanf(fp, "%d  %d ", &height[i], &width[i]);

    }

    bound=0;
    hbinsize=hbinsize+bound;
    vbinsize=vbinsize+bound;
    binarea=vbinsize*hbinsize;

_ftime(&time1);


    for (loop=0;loop<20;loop++){
    printf("\n");
    printf("loop = %d\n", loop);
    printf("\n");
    fprintf(op,"loop = %d\n", loop);

    /* initialisation*/
    obj_size=NULL;
    strip_size=NULL;
    deleted_node  =NULL;
    deleted_node1 =NULL;
```

```
deleted_node2 =NULL;
inserted_node =NULL;
current_node  =NULL;
temp_node=NULL;
temp_node1=NULL;
temp_node2=NULL;

vnumbin=lowerbound+5;

for (i=0;i<vnumbin;i++){
    vbin_root[i] = NULL;
    vertical_bin[i]=NULL;
    space[i]=NULL;
}
for (i=0;i<maxhnumbin;i++)
    hbin_root[i] = NULL;

for( i=0;i<vnumbin;i++)
    vcurcap[i]=vbinsize;
for( i=0;i<maxhnumbin;i++)
    hcurcap[i]=hbinsize;

for( i=0;i<vnumbin;i++){
    vbinload[i]=0;
    vnitembin[i]=0;
    vernitembin[i]=0;
}
for( i=0;i<maxhnumbin;i++){
    hbinload[i]=0;
    hnitembin[i]=0;
}
for (i=0;i< MAX_NUMBER_BIN;i++){
    for(j=0;j<MAX_NUMBER_HOR; j++){
        ceilheight [i][j]=0;
        ceilwidth  [i][j]=0;
        ceilcap  [i][j]=0;
        floorheight[i][j]=0;
        floorwidth [i][j]=0;
        floorcap   [i][j]=0;

    }
}

for (i=0;i<MAX_NUMBER_BIN;i++)
    numcell[i]=0;
```

```
T=1.0;
Tred=0.95;
numstrip=0;
count=0;

    /*read data file*/
    for (i=0;i<numbitem;i++){
        current_node=CreatNode(height[i], width[i]);
        current_node->area=height[i]*width[i];
        HorInsertNode(&obj_size, current_node);
    }

    idec=irbit1(&iseed);

    /* FIRST-FIT DECREASING for horizontal bins*/
    for (i=0;i<numbitem;i++){
        idec=irbit1(&iseed);

        if ((i== (numbitem-1))||(i==(numbitem-2))) idec=0;


        deleted_node= DeleteNode(&obj_size,idec);
        for (j=0;j<maxhnumbin;j++)
            if (hcurcap[j] >= deleted_node->width){
                hcurcap[j]=hcurcap[j]- (deleted_node->width);
                deleted_node->stripnum=j;
                HorInsertNode(&hbin_root[j],deleted_node);

                break;
            }
    }

    /*linked list for vertical bins*/
    for (i=(maxhnumbin-1);i>=0;i--){

        if (hbin_root[i] !=NULL){
            numstrip++;
            strip_size = AddNode(hbin_root[i]->height, hbin_root[i]->width,
            hbin_root[i]->stripnum, hbin_root[i]->binnum, strip_size);
        }
    }

    hnumbin=numstrip;

    /*compute the total load and number of items in each bin*/
    for (i=0; i<hnumbin;i++) {
```

```
    hbinload[i]=CompHbinLoad(hbin_root[i]);
    hnitembin[i]=CompBinItem(hbin_root[i]);
}



/* first-fit decreasing for vertical bins*/
for (i=0;i<hnumbin;i++){

    deleted_node= DeleteNode(&strip_size,0);
    for (j=0;j<vnumbin;j++)
        if (vcurcap[j] >= deleted_node->height){
            vcurcap[j]=vcurcap[j]- (deleted_node->height);

            temp_node=hbin_root[deleted_node->stripnum];
            for (k=0;(k<hnitembin[(deleted_node->stripnum)]);k++){
                temp_node->binnum=j;
                if (temp_node->link != NULL)
                temp_node=temp_node->link;
            }
            temp_node=NULL;

            deleted_node ->binnum=j;
            VerInsertNode(&vbin_root[j],deleted_node);
            break;
        }
}

/*compute the total load and number of items in each bin*/
for (i=0; i<vnumbin;i++) {
    vbinload[i]=CompVbinLoad(vbin_root[i]);
    vnitembin[i]=CompBinItem(vbin_root[i]);
}


/*compute utilised areas in each horizontal bin*/

for (i=0;i<hnumbin;i++)
        hbinarea[i]=0;

for (i=0; i<hnumbin;i++){
    temp_node=hbin_root[i];
    for (k=0;(k<hnitembin[i]);k++){
        hbinarea[i]=hbinarea[i]+temp_node->area;
        temp_node=temp_node->link;
    }
```

```
}

for (m=0;m<20;m++){


    /* update weights, set K = 0.05*/
    for (i=0;i<hnumbin ; i++){
        hweight[i]= pow((1.0+((double)hcurcap[i]/(double)hbinsize)*0.05), T);
    }

    /*swap item amongst levels */
    for (i=0;i<vnumbin;i++) {
        improved=0;

        temp_node1=vbin_root[i];
        for(j=0; j<vnitembin[i];j++){

            for (p=0;p<vnumbin;p++){

                temp_node2=vbin_root[p];
                for(q=0;q<vnitembin[p];q++){

                    if  ((temp_node1->stripnum)!=(temp_node2->stripnum)){


                        improved = hswap10(&hbin_root[temp_node1->stripnum],
                            &hbin_root[temp_node2->stripnum], temp_node1,
                            temp_node2,(temp_node1->stripnum),
                            (temp_node2->stripnum),i, p, hweight);

                        if (hbin_root[temp_node1->stripnum]==NULL){
                            removebin(&vbin_root[i]);
                            vnitembin[i]--;
                            break;
                        }
                        if (done) break;

                        improved = hswap11(&hbin_root[temp_node1->stripnum],
                            &hbin_root[temp_node2->stripnum], temp_node1,
                            temp_node2,(temp_node1->stripnum),
                            (temp_node2->stripnum), i, p, hweight);
                        if (done) break;

                        improved = hswap12(&hbin_root[temp_node1->stripnum],
                            &hbin_root[temp_node2->stripnum],temp_node1,
                            temp_node2,(temp_node1->stripnum),
```

```
                        (temp_node2->stripnum), i, p, hweight);
                    if (done) break;

                    improved = hswap22(&hbin_root[temp_node1->stripnum],
                        &hbin_root[temp_node2->stripnum], temp_node1,
                        temp_node2,(temp_node1->stripnum),
                        (temp_node2->stripnum), i, p, hweight);
                    if (done) break;

                }

                temp_node2=temp_node2->link;

            }
            if (hbin_root[temp_node1->stripnum]==NULL) break;
            if (done) break;
        }
        if (done) break;
        if(temp_node1 !=NULL)
        temp_node1=temp_node1->link;
    }
    if (done) break;
}
if(done) break;
T=T*Tred;

}

/*packing vertical bins*/
    T=1.0;
    for (m=0;m<20;m++){

        /* update weights, set K = 0.05*/

        for (i=0;i<vnumbin ; i++)
                vweight[i]= pow((1.0+((double)vcurcap[i]/(double)vbinsize)*0.05), T);

        /*swapping items between vertical bins*/
        for (i=0;i<vnumbin;i++) {
                improved=0;

            for (p=0;p<vnumbin;p++){

                if  (p!=i){

                    improved = vswap10(&vbin_root[i], &vbin_root[p],i,p, vweight);
```

```
                    if (done) break;

                    improved = vswap11(&vbin_root[i], &vbin_root[p],i,p,vweight);
                    if (done) break;

                    improved = vswap12(&vbin_root[i], &vbin_root[p],i,p,vweight);
                    if (done) break;

                    improved = vswap22(&vbin_root[i], &vbin_root[p],i,p,vweight);
                    if (done) break;


                }

            }

            if (done) break;
        }
        if (done) break;
        T=T*Tred;
    }


count1=0;
for (i=0;i<vnumbin;i++){
    if (vbin_root[i]!=NULL)
        count1++;
}

/*initialise hposition, vposition, floornum, ceilnum for horizontal levels  after hswap*/

        for (i=0;i<vnumbin;i++){
            temp_node1=vbin_root[i];

            for (j=0;(j<vnitembin[i]);j++){
                s=temp_node1->stripnum;
                temp_node2=hbin_root[s];

                for (k=0;(k<hnitembin[s]);k++){

                    temp_node2->binnum=i;
                    temp_node2->hposition = k;
                    temp_node2->vposition = j;
                    temp_node2->floornum = k;
                    temp_node2->ceilnum  =-1;
                    temp_node2=temp_node2->link;
```

```
            }

            temp_node1->binnum=i;
            temp_node1->hposition=0;
            temp_node1->vposition = j;
            temp_node1->floornum = 0;
            temp_node1->ceilnum  =-1;
            temp_node1->area=temp_node1->height * temp_node1->width;
            temp_node1=temp_node1->link;
        }
    }

/* determine space occupied by items, and residual areas in each level */

    for (i=0;i<hnumbin;i++){

        if (hbin_root[i]!=NULL){

            numcell[i]=hnitembin[i];
            temp_node1=hbin_root[i];
            preht = 0;
            culwidth=0;

            for (j=0;(j<hnitembin[i]);j++){

                if (j==0){
                    culwidth=temp_node1->width;
                    if (temp_node1->vposition == (vnitembin[temp_node1->binnum]-1)){
                        ceilheight [i][0]=temp_node1->height + vcurcap[temp_node1-
                                        >binnum];
                        preht=temp_node1->height+ vcurcap[temp_node1->binnum];

                        ceilheight [i][numcell[i]]=vcurcap[temp_node1->binnum];
                        ceilwidth  [i][numcell[i]]=temp_node1->width;
                        ceilcap    [i][numcell[i]]=ceilwidth [i][numcell[i]];
                        floorheight[i][numcell[i]]=0;
                        floorwidth [i][numcell[i]]=0;
                        floorcap   [i][numcell[i]]=0;

                        numcell[i]++;
                    }
                    else{
                        ceilheight [i][0]=temp_node1->height;
                        preht=temp_node1->height;
                    }
                    ceilwidth  [i][0]=hbinsize-culwidth;
```

```
                ceilcap    [i][0]=ceilwidth[i][0];
                floorheight[i][0]=temp_node1->height;
                floorwidth [i][0]=temp_node1->width;
                floorcap   [i][0]=0;

            }

            else if (j==1){
                ceilheight [i][0]=preht - temp_node1->height;

                culwidth        =culwidth+temp_node1->width;
                ceilheight [i][1]=temp_node1->height;
                ceilwidth  [i][1]=hbinsize-culwidth;
                ceilcap    [i][1]=ceilwidth[i][1];
                floorheight[i][1]=temp_node1->height;
                floorwidth [i][1]=temp_node1->width;;
                floorcap   [i][1]=0;

                preht=temp_node1->height;

            }

            else {
                ceilheight [i][j-1]=preht - temp_node1->height;
                ceilwidth  [i][j-1]=temp_node1->width;
                ceilcap    [i][j-1]=ceilwidth[i][j-1];

                culwidth          =culwidth+temp_node1->width;
                ceilheight [i][j]  =preht;
                ceilwidth  [i][j]  =hbinsize-culwidth;
                ceilcap    [i][j]  =ceilwidth[i][j];
                floorheight[i][j]  =temp_node1->height;
                floorwidth [i][j]  =temp_node1->width;;
                floorcap   [i][j]  =0;


            }
            temp_node1=temp_node1->link;

        }

    }

}

/*compute utilised areas in each horizontal bin*/
```

```
    for (i=0;i<hnumbin;i++){
            hbinarea[i]=0;
            hareacap[i]=0;
    }
    for (i=0; i<hnumbin;i++){
        if (hbin_root[i]!=NULL){
            temp_node=hbin_root[i];
            for (k=0;(k<hnitembin[i]);k++){
                if (k==0){
                    hmaxht[i]=temp_node->height;
                    hstripsize[i]=hmaxht[i]*hbinsize;
            }

                hbinarea[i]=hbinarea[i]+temp_node->area;
                temp_node=temp_node->link;
            }
            hareacap[i]=hstripsize[i]-hbinarea[i];
        }

    }

    for (i=0;i<vnumbin;i++){
        vbinarea[i]=0;
        vareacap[i]=binarea;
    }
    for (i=0; i<vnumbin;i++){
        if (vbin_root[i]!=NULL){

        temp_node1=vbin_root[i];
        for (j=0;(j<vnitembin[i]);j++){

            vbinarea[i]=vbinarea[i]+hbinarea[temp_node1->stripnum];
            temp_node1=temp_node1->link;
        }
        vareacap[i]=binarea-vbinarea[i];

        }
    }

/*filling residual space in  horizontal levels */
T=1.0;

for (n=0;n<20;n++){

    for (i=0;i<vnumbin ; i++)
```

```
    bweight[i]= pow( (1.0+((double)vareacap[i]/(double)binarea)*0.05), T);

for (i=0;i<hnumbin;i++){

    if (hbin_root[i]!=NULL)
        vweight[i]=bweight[hbin_root[i]->binnum];
    else
        vweight[i]=0.0;

}
for (i=0;i<hnumbin;i++)
    hweight[i]= pow( (1.0+((double)hareacap[i]/(double)hbinarea[i])*0.05), T);


/*filling the residual space within each level */
for (i=0;i<hnumbin;i++) {
    improved=0;

    if (hbin_root[i]!=NULL){

        for(j=0; j<hnumbin;j++){

            if (hbin_root[j]!=NULL){

                if  (i!=j){

                    if (hbin_root[i]->binnum == hbin_root[j]->binnum)
                        improved = levelpack(&hbin_root[i], &hbin_root[j], i, j, hweight);
                    else
                        improved = levelpack(&hbin_root[i], &hbin_root[j], i, j, vweight);

                    if (hbin_root[i]==NULL)
                    break;
                    if (done) break;

                }

            }

        }
        if (done) break;
    }

}
if (done) break;
T=T*Tred;
```

```
}

count=0;
for (i=0;i<hnumbin;i++){
    count=count+hnitembin[i];
}

count=0;
for (i=0;i<vnumbin;i++){
    if (vbin_root[i]!=NULL)
        count++;
}

/*Exit the loop if the 2BP|O|G solution conincide with the best lower bound*/
if (count==lowerbound){
    done=1;

    printf("\n");
    printf("Done!!\n");
    printf("\n");
}

if(done)
    break;

/* consolidate all items in a  vertical bin, remove empty vertical bins and re-number
bins*/
count1=0;
count2=vnumbin;

for (i=0;i<vnumbin;i++){
    temp_node1=vbin_root[i];
    if(vbin_root[i]!=NULL){
        for (j=0;(j<vnitembin[i]);j++){
            s=temp_node1->stripnum;
            temp_node2=hbin_root[s];
            if(hbin_root[s]!=NULL){
                for (k=0;(k<hnitembin[s]);k++){
                    deleted_node= DeleteNode(&hbin_root[s], 0);
                    deleted_node->binnum=count1;
                    deleted_node->stripnum=0;
                    deleted_node->hposition=0;
                    deleted_node->vposition=0;
                    deleted_node->floornum=0;
                    deleted_node->ceilnum=0;
```

```
                    HorInsertNode(&vertical_bin[count1],deleted_node);
                    vernitembin[count1]++;
                    if (temp_node2!=NULL)
                        temp_node2=temp_node2->link;
                }
            }
            temp_node1=temp_node1->link;
        }
        count1++;
    }
    else {
    count2--;

    }
}
vnumbin=count2;

/* recompute utilised bin areas after renumber*/

for (i=0;i<(vnumbin+2);i++){
    vbinarea[i]=0;
    vareacap[i]=binarea;
}
for (i=0; i<vnumbin;i++){

    temp_node1=vertical_bin[i];

    while (temp_node1!=NULL){

        vbinarea[i]=vbinarea[i]+ temp_node1->area;
        temp_node1=temp_node1->link;

    }
    vareacap[i]=binarea-vbinarea[i];

}

/*repack all bins using altenate direction algorithm*/

for(i=0;i<vnumbin;i++){
    stackheight[i]=alterdirection(&vertical_bin[i]);
}

feasible=0;
while(feasible==0){
```

```
    feasible=1;
    for (i=0;i<vnumbin;i++){

        /* repack infeasible bins*/
        if(stackheight[i]>vbinsize){
            deleted_node = DeleteNode(&vertical_bin[i], (vernitembin[i]-1));
            vernitembin[i]--;
            vbinarea[i]=vbinarea[i]-deleted_node->area;
            vareacap[i]=binarea-vbinarea[i];
            stackheight[i]=alterdirection(&vertical_bin[i]);

            if (vareacap[vnumbin]>=deleted_node->area){
                deleted_node->binnum=vnumbin;
                HorInsertNode(&vertical_bin[vnumbin],deleted_node);
                vernitembin[vnumbin]++;
                vbinarea[vnumbin]=vbinarea[vnumbin]+deleted_node->area;
                vareacap[vnumbin]=binarea-vbinarea[vnumbin];
                stackheight[vnumbin]=alterdirection(&vertical_bin[vnumbin]);
                feasible=0;
                break;
            }
            else{
                deleted_node->binnum=vnumbin+1;
                HorInsertNode(&vertical_bin[vnumbin+1],deleted_node);
                vernitembin[vnumbin+1]++;
                vbinarea[vnumbin+1]=vbinarea[vnumbin+1]+deleted_node->area;
                vareacap[vnumbin+1]=binarea-vbinarea[vnumbin+1];
                stackheight[vnumbin+1]=alterdirection(&vertical_bin[vnumbin+1]);
                feasible=0;
                vnumbin++;
                break;

            }


        }
    }
}

vnumbin++;


for(i=0;i<vnumbin;i++){
    stackheight[i]=alterdirection(&vertical_bin[i]);
}
```

```
T=1.0;
Tred=0.95;

/*swapping items between bins, and repacking using alternate direction algorithm*/

for (n=0;n<30;n++){

    for (i=0;i<vnumbin ; i++)
        bweight[i]= pow( (1.0-((1.0*stackheight[i])/(1.0*vbinsize))*0.05), T);

    for (i=0;i<vnumbin;i++) {

        for(j=0; j<vnumbin;j++){

            if  ( i!=j){

                        improved = fswap10(&vertical_bin[i], &vertical_bin[j], i, j, bweight);
                        if (done) break;

                        improved = fswap11(&vertical_bin[i], &vertical_bin[j], i, j, bweight);
                        if (done) break;

                        improved = fswap12(&vertical_bin[i], &vertical_bin[j], i, j, bweight);
                        if (done) break;

                        improved = fswap22(&vertical_bin[i], &vertical_bin[j], i, j, bweight);
                        if (done) break;

            }

        }
        if (done) break;

    }
    if (done) break;
    T=T*Tred;

}



if (done) break;


for (i=0;i<vnumbin;i++){
    stackheight[i]=alterdirection(&vertical_bin[i]);
```

```
    if(vertical_bin[i]!=NULL)
        /* compute the coordinates of dead space*/
        ComputeSpace(&vertical_bin[i]);
}

maxcap=0;
maxind=0;

for (i=0;i<vnumbin;i++){

    if ((vareacap[i]>maxcap)&&(vareacap[i]!=binarea)){
        maxcap=vareacap[i];
        maxind=i;
    }
}

count1=0;

done=0;

/* move items to occupy dead space*/

for (i=0;i<vernitembin[maxind];i++){

    improved=0;

    for(j=0;j<vnumbin;j++){

        if ((j!=maxind)||(vareacap[j]!=binarea)){

            temp_node1=space[j];

            while(temp_node1!=NULL){

                /*allow rotations of items*/
                if(( ((temp_node1->height>=vertical_bin[maxind]->height) &&
                        (temp_node1->width >=vertical_bin[maxind]->width))||
                    ((temp_node1->width >=vertical_bin[maxind]->height) &&
                        (temp_node1->height>=vertical_bin[maxind]->width)) ) &&
                    (temp_node1->tag!=1) ){

                    if ((temp_node1->height>=vertical_bin[maxind]->height) &&
                        (temp_node1->width >=vertical_bin[maxind]->width))
                        rotated=0;
                    else
```

```
                rotated=1;

            deleted_node=DeleteNode(&vertical_bin[maxind],0);
            temp_node1->tag=1;
            deleted_node->tag=1;
            count1++;

            if (rotated==0)
                deleted_node->orientation=0;
            else
                deleted_node->orientation=1;

            HorInsertNode(&vertical_bin[temp_node1->binnum], deleted_node);
            vernitembin[temp_node1->binnum]++;
            improved=1;

            count=0;
            done=0;
            for (k=0;k<vnumbin ; k++)
                if (vertical_bin[k]!=NULL)
                    count++;

            if (count==lowerbound){
                done = 1;
                stackheight[maxind]=0;
                vernitembin[maxind]=0;
            }

            break;
        }

        temp_node1=temp_node1->link;
    }
    if(improved==1)
        break;

  }
  if(done==1)
    break;

 }
if(done==1)
    break;

}
```

```
if (done==1)
    break;
else
    vernitembin[maxind]=vernitembin[maxind]-count1;


_ftime(&time2);
totaltime=TimeElapsed(&time1,&time2);
printf("time = %f\n",totaltime);
if(totaltime>timelimit)
    break;

}   /* end of loop*/

    fprintf(op,"\n");
    fprintf(op, "Subroutine Elapsed time:%f \n", totaltime);
    fprintf(op, "Program Elapsed time:  %f\n", (float)clock()/CLOCKS_PER_SEC);
    printf("Subroutine Elapsed time:%f \n", totaltime);

    fclose(fp);
    fclose(op);
    return EXIT_SUCCESS;

}



/* functions*/

void PrintFileNode(Node * item)
{
    fprintf(op, "%5d %5d %5d %5d %5d %5d %5d %5d %5d %5d %5d\n", item-
>height,item->width,item->stripnum, item->hposition, item->binnum,item-
>vposition,item->floornum,item->ceilnum,item->area, item->tag,item->orientation);

}

void PrintFile(Node * root)
{
    if (root != NULL) {
        while (root != NULL) {
            PrintFileNode(root);
            root = root->link;
        }
```

```
    } else {
        fprintf(op, " No nodes have been entered yet!\n");
    }
}


void PrintNode(Node * item)
{
    printf("%5d %5d %5d %5d %5d %5d %5d %5d %5d %5d %5d\n", item-
>height,item->width,item->stripnum, item->hposition,item->binnum,item-
>vposition,item->floornum,item->ceilnum,item->area, item->tag, item->orientation);

}

Node * CopyNode(Node * root)
{
    struct NODE * temp = (struct NODE*) malloc(sizeof(struct NODE));

    temp->height=root->height;
    temp->width=root->width;
    temp->stripnum=root->stripnum;
    temp->binnum=root->binnum;
    temp->hposition=root->hposition;
    temp->vposition=root->vposition;
    temp->ceilnum=root->ceilnum;
    temp->floornum=root->floornum;
    temp->area=root->area;
    temp->tag=root->tag;
    temp->orientation=root->orientation;

    return temp;
}



Node * CreatNode(int high, int wide)
{
    struct NODE * temp = (struct NODE*) malloc(sizeof(struct NODE));

    temp->height=high;
    temp->width=wide;
    temp->stripnum=0;
    temp->binnum=0;
    temp->hposition=0;
    temp->vposition=0;
    temp->ceilnum=0;
```

```
    temp->floornum=0;
    temp->area=0;
    temp->tag=0;
    temp->orientation=0;

    return temp;
}



Node * AddNode(int high, int wide, int num1, int num2, Node * bin )
{
    struct NODE * temp = (struct NODE*) malloc(sizeof(struct NODE));

    temp->height = high;
    temp->width  = wide;
    temp->stripnum=num1;
    temp->binnum=num2;
    temp->vposition=0;
    temp->ceilnum=0;
    temp->floornum=0;
    temp->area=0;
    temp->link = bin;
    bin = temp;
    return bin;
}



int HorInsertNode( register Node **linkp, Node * inserted_node)
{
    register Node   *current;
    while( ( current = *linkp ) != NULL && current->height >=inserted_node->height){

        if ((current->height == inserted_node->height)&&(current->width <
inserted_node->width))
            break;

        linkp = &current->link;

    }

    inserted_node->link = current;
    *linkp = inserted_node;

    return TRUE;
```

```
}


int HorInsertOrderNode( register Node **linkp, Node * inserted_node)
{
    register Node   *current;
    while( ( current = *linkp ) != NULL && current->floornum < inserted_node-
>floornum){

        linkp = &current->link;

    }

    inserted_node->link = current;
    *linkp = inserted_node;

    return TRUE;
}


int VerInsertNode( register Node **linkp, Node * inserted_node)
{
    register Node   *current;
    while( ( current = *linkp ) != NULL &&
        hbinload[current->stripnum] >hbinload[inserted_node->stripnum])
        linkp = &current->link;

    inserted_node->link = current;
    *linkp = inserted_node;

    return TRUE;
}



Node * DeleteTagNode( Node **linkp)
{
    Node *current;
    Node *previous;
    Node *delnode;

    current = *linkp;
    previous = NULL;

    while((current != NULL) && (current->tag==0)){
```

```
        previous = current;
        current = current->link;
    }

    delnode = current;

    if(previous==NULL)
        *linkp=current->link;
    else
        previous->link=delnode->link;

    if(current!=NULL)
        current = current->link;
    else
        current->link=NULL;

    return  delnode;
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//  2bp_r_f.h  is the header file for WA2BPRF.cpp
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

typedef  struct NODE   {
    struct NODE   *link;
    int height;
    int width;
    int stripnum;
    int binnum;
    int hposition;
    int vposition;
    int floornum;
    int ceilnum;
    int orientation;
    int area;
    int tag;
} Node;

void sort(unsigned long n, double arr[]);
#define SQ(a) (a*a)
#define  FALSE  0
#define TRUE 1
#define MAX_NUMBER_BIN 100
#define MAX_NUMBER_ITEM 150
#define MAX_NUMBER_HOR  20
#define MAX_NUMBER_VER 10
#define RIGHT 0
#define LEFT 1
```

Appendix G

Maximum Cardinality Bin Packing Problem Code

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//*     WAMCBP.cpp is the program for sovling the maximum cardinality bin packing
//problem. It generates the instances of Peeters et al. (2006).
//*     It computes the L2, and L3 bounds based on the procedures of Martello and Toth
//(1990).
//*     It calls functions from WA1BP.cpp
//*     It is to be complied with the following files (ANSI version) from Press et al.(2002)
//          1)    nrutil.cpp
//          2)    nrutil.h
//          3)    nr.h
//*     The header file cbp.h is appended at the end of this program.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#include <io.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "nrutil.h"
#include "nr.h"
#include <sys\timeb.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include "cbp.h"
#include <string.h>

Node * bin_root[MAX_NUMBER_BIN];
Node * obj_size;
Node * valid_items;

FILE * fp;
FILE * op;
int binload[MAX_NUMBER_BIN];
int curcap [MAX_NUMBER_BIN];
int nitembin[MAX_NUMBER_BIN];
int done =0;
int apriorbound=0;
int bound = 0;
int binsize = 0;
long idum =-1;
```

```
double comptime=0.0;
double totaltime=0.0;
double computime=0.0;

double TimeElapsed(struct _timeb *begin, struct _timeb *end);
int irbit1(unsigned long *iseed);
void PrintNode(Node * item);
void PrintAllNode(Node * root);
void PrintFileNode(Node * item);
void PrintFile(Node * root);
Node * AddNode(int data, Node * bin, int order);
Node * DeleteNode( Node **linkp, int item_num );
int InsertNode( register Node **linkp, Node * inserted_node);
int InsertNode_D( register Node **linkp, Node * inserted_node);
int CompBinLoad(Node * root);
int CompBinItem(Node * root);
void ListNodeDestroy(Node *nptr);
void ListDestroy(Node *list);
int ItemValue(Node **linkp, int item_num);
int GenInstance(int maxsize, int minsize,int numknapsack,int knapcap,int rep,int
itemsize[][MAX_NUMBER_ITEM]);
int AprioriUpperbound(Node * root, int numitemgen, int numknapsack, int knapcap,int
cnumbitem);
int MTL2(int knapcap,int cnumbitem,int itemlist[]);
int MTRP(int knapcap,int cnumbitem,int itemlist[], int bj[]);
int MTL3(int knapcap,int cnumbitem, int itemlist[]);
int swap11(Node ** rootp1, Node ** rootp2, int i,int p, float weight[], int numbin);
int swap12(Node ** rootp1, Node ** rootp2, int i,int p, float weight[], int numbin);
int swap22(Node ** rootp1, Node ** rootp2, int i,int p, float weight[], int numbin);

int
main(void)
{

    struct _timeb time1, time2;
    unsigned long iseed=111;


    Node * deleted_node;
    Node * deleted_node1;
    Node * deleted_node2;
    Node * inserted_node;
    Node * current_node;
    Node * temp_node;

    int i,j,idec,p,m,n;
```

```
int freq = 0;
int maxind;
int numbitem=0;
int improved=0;
int numtype=0;
int numsuc=0;
int cnumbitem=0;
int maxsize=999;
int minsize=0;
int expbinnum=0;
int maxcap;
int numknapsack=0;
int knapcap=0;
int numitemgen[10];
int sum=0;
int sumnum=0;
int swap=0;
int capacity[10]={1000,1200,1500,2000,3000,4000,5000,6000,7000,8000};
int itemsize[10][MAX_NUMBER_ITEM];
int itemlist[MAX_NUMBER_ITEM];
int numbin =0;
int numrep =10;
int rep;
int count=0;
int L2=0;
int L3=0;

float weight [MAX_NUMBER_BIN];
int obinsize = 0;
double T = 1;
double Tred =0.99;
float temp=0.0;

if((op=fopen("out.txt", "w")) == NULL) {
    printf("Cannot open file.\n");
    exit(1);
}
numrep=10;

expbinnum=100;
minsize=1;
knapcap=capacity[0];

/*compute the number of knapsack based on expected number of items*/
numknapsack=ceil((float)((minsize+maxsize)*expbinnum)/(float)(2*knapcap));
```

```
for (j=0;j<10;j++){
    for(i=0;i<10;i++){
        itemsize[j][i]=0;
    }
}

for (rep=0; rep<numrep; rep++){

    /*generate the instances of Peeters et al.(2006)*/
    numitemgen[rep]=GenInstance(maxsize,minsize,numknapsack,knapcap,rep,
                                itemsize);
    sumnum=sumnum+numitemgen[rep];

    _ftime(&time1);



    /* initialisation */
    obj_size=NULL;
    deleted_node  =NULL;
    deleted_node1 =NULL;
    deleted_node2 =NULL;
    inserted_node =NULL;
    current_node  =NULL;
    valid_items=NULL;

    for (i=0;i<numitemgen[rep];i++){
        obj_size = AddNode(itemsize[rep][i], obj_size,i);
    }

    temp_node=obj_size;
    while (temp_node!=NULL){
        temp_node=temp_node->link;
        deleted_node= DeleteNode(&obj_size,0);
        InsertNode_D(&valid_items,deleted_node);
    }

    /*compute a priori upper bounds*/
    apriorbound=AprioriUpperbound(valid_items, numitemgen[rep], numknapsack,
                                          knapcap, cnumbitem);
    cnumbitem=apriorbound;

    /*Update itemlist*/
    temp_node=valid_items;
    for(i=0;i<cnumbitem;i++){
        itemlist[cnumbitem-1-i]=temp_node->value;
```

```
    temp_node=temp_node->link;
}


temp_node=valid_items;
for(i=0;i<cnumbitem;i++){
    itemsize[rep][i]=temp_node->value;
    temp_node=temp_node->link;
}

/*compute L2 and L3*/
L2=MTL2(knapcap,cnumbitem,itemlist);
printf("L2(final)=%d\n", L2);

L3=MTL3(knapcap,cnumbitem,itemlist);
printf("L3(final)=%d\n", L3);

/*reduce upper bounds of residual instances until L3 = number of knapsacks*/
while(L3>numknapsack){
    apriorbound--;
    DeleteNode(&valid_items,apriorbound);
    cnumbitem--;
    itemsize[rep][apriorbound]=0;
    L3=MTL2(knapcap,cnumbitem,itemlist);
}

cnumbitem=apriorbound;

obj_size=valid_items;
numbin=numknapsack;
numbitem=cnumbitem;
obinsize=knapcap;
bound = (int)0.1*obinsize+1;
binsize = obinsize+bound;

for (i=0;i<numbin;i++)
    bin_root[i] = NULL;
for( i=0;i<numbin;i++)
    curcap[i]=binsize;
for( i=0;i<numbin;i++){
    binload[i]=0;
    nitembin[i]=0;
}

for (n=0;n<20;n++){
```

```
T=1;
temp=0.0;
done=0;

/*midified first-fit decreasing*/
for (i=0;i<numbitem;i++){
    maxind=0;
    maxcap=curcap[0];
    for (j=1;j<numbin;j++)
        if (curcap[j] > maxcap){
            maxcap=curcap[j];
            maxind=j;
        }
    idec=irbit1(&iseed);
    if (i== (numbitem-1)) idec=0;
    deleted_node= DeleteNode(&obj_size,idec);
    InsertNode(&bin_root[maxind],deleted_node);
    curcap[maxind]=curcap[maxind]- (deleted_node->value);
}

/*compute the total load and number of items in each bin*/
for (i=0; i<numbin;i++) {
    binload[i]=CompBinLoad(bin_root[i]);
    nitembin[i]=CompBinItem(bin_root[i]);
}

for (m=0;m<50;m++){

    /* update weights, set K = 0.05*/
    for (i=0;i<numbin ; i++){
        weight[i]= pow((1.0-(curcap[i]/binsize)*0.05), T);
    }

    /*swapping items between bins*/

    for (i=0;i<numbin;i++) {
        improved=0;

        for (p=0;p<numbin;p++){
            if (p!=i){

                improved = swap11(&bin_root[i], &bin_root[p], i, p, weight,
                                  numbin);
                if (done) break;
```

```
                improved = swap12(&bin_root[i], &bin_root[p], i, p, weight,
                                  numbin);
                if (done) break;

                improved = swap22(&bin_root[i], &bin_root[p], i, p, weight,
                                  numbin);
                if (done) break;


            }
        }
        if (done) break;
    }


    if(done) break;

    T=T*Tred;
}

obj_size=NULL;

for (i=(cnumbitem-1);i>=0;i--)
    obj_size = AddNode(itemsize[rep][i], obj_size,i);

deleted_node  =NULL;
deleted_node1 =NULL;
deleted_node2 =NULL;
inserted_node =NULL;
current_node  =NULL;
valid_items=NULL;
for (i=0;i<numbin;i++){
    ListDestroy(bin_root[i]);
    bin_root[i] = NULL;
    curcap[i]=binsize;
    binload[i]=0;
    nitembin[i]=0;
}

if (done){
    numsuc=numsuc+1;
    _ftime(&time2);
    computime=computime+TimeElapsed(&time1,&time2);
    fprintf(op, "Test  %d   done! \n", rep);
    printf(" rep  %d   done! \n", rep);
    printf("\n");
```

```
                break;
            }

        }

        _ftime(&time2);

        comptime= TimeElapsed(&time1,&time2);
        totaltime=totaltime+comptime;

        fprintf(op, "rep  %d     %f \n", rep, comptime);

        /*print output to files*/

        fprintf(op,"\n");
        fprintf(op, " Bin Loads         Number of Items\n");
        for(i =0; i <numbin; i++){
            fprintf(op, "%3d  ", i);
            fprintf(op,"%d", binload[i]);
            fprintf(op,"          %5d\n", nitembin[i]);
        }

        fprintf(op,"\n");
        fprintf(op, "Subroutine Elapsed time:%f \n", comptime);

        fprintf(op,"\n");

    }
    printf("\n");
    printf("number of instances solved = %d\n", numsuc);
    printf("\n");

    fprintf(op,"\n");
    fprintf(op, "Average time  %f  \n", totaltime/(float)(rep+1));
    printf("Average time (excluding non-optimal solutions)= %f  \n",
    computime/(float)(numsuc));
    fprintf(op,"\n");
    printf("\n");
    fclose(op);

    return EXIT_SUCCESS;

}
```

```
//* functions


/*compute the L3 bounf from Martello & Toth (1990)*/

int MTL3(int knapcap,int cnumbitem, int itemlist[])
{

    int i,j, L2,L3;
    int zr=0;
    int z=0;
    int nbar=0;
    int k=0;
    int bj[MAX_NUMBER_ITEM];
    int wjbar[MAX_NUMBER_ITEM];

    for(j=0;j<cnumbitem;j++)
        bj[j]=0;


    nbar=cnumbitem;
    for(j=0;j<cnumbitem;j++){
        wjbar[j]=itemlist[j];
    }

    while(nbar>=1){

        zr=MTRP(knapcap,nbar,wjbar, bj);


        z=z+zr;
        k=0;

        for(j=0;j<nbar;j++){

            if (bj[j]==0){
                wjbar[k]=wjbar[j];
                k++;
            }
        }

        nbar=k;
```

```
        for(j=0;j<nbar;j++)
            bj[j]=0;

        if (nbar==0)
            L2=0;
        else
            L2=MTL2(knapcap,nbar,wjbar);

        if((z+L2)>L3)
            L3=z+L2;

        nbar--;

    }

    return L3;
}


/*Reduction Procedures from Martello and Toth (1990)*/

int MTRP(int knapcap,int cnumbitem, int itemlist[], int bj[])

{
    Node *sorted_list=NULL;
    Node *assigned=NULL;
    Node *unassigned=NULL;
    Node *F=NULL;
    Node *J=NULL;
    Node *Jstar=NULL;
    Node *Ja=NULL;
    Node *Jb=NULL;
    Node * deleted_node=NULL;
    Node *temp_node=NULL;
    Node *temp_node1=NULL;
    Node *temp_node2=NULL;


    int i,j,zr,K, wjstar,wja,wjb,wjb1,wjb2, sum,listsize, r, s, A,B, bestsol,
        counter,jstarposn;
    int nodevalue[3];


    for(i=0;i<cnumbitem;i++){
        sorted_list=AddNode(itemlist[cnumbitem-i-1],sorted_list, (cnumbitem-i-1));
```

```
}

K=0;
zr=0;
listsize=cnumbitem;


while (sorted_list!=NULL){

    F=NULL;
    for(i=0;i<3;i++)
        nodevalue[i]=knapcap+1;

    J=DeleteNode(&sorted_list,0);

    if (sorted_list==NULL)
        InsertNode(&unassigned,J);
    else{

        listsize--;

        K=0;
        temp_node=sorted_list;

        for(i=0;i<listsize;i++){

            counter = listsize-i;

            if(counter==3){
                nodevalue[2]=temp_node->value;

            }
            else if (counter==2){
                nodevalue[1]=temp_node->value;
                        }
            else if (counter==1){
                nodevalue[0]=temp_node->value;

            }

            temp_node=temp_node->link;
        }

        sum=J->value;

        for (i=0;i<3;i++){
```

```
    sum=sum+nodevalue[i];

    if(sum<=knapcap){
        K++;

    }
}

//*assignments of last three items
if (((listsize==2)&&(K==2)) ||((listsize==1)&&(K==1)))
    K=3;

if (K==0){
    InsertNode(&F,J);
}
else{

    temp_node=sorted_list;
    jstarposn=0;
    while(temp_node!=NULL){
        if((J->value+temp_node->value)<=knapcap){

            wjstar=temp_node->value;
            break;

        }
        jstarposn++;
        temp_node=temp_node->link;
    }

    if ((K==1)||((J->value +wjstar)==knapcap)){
        InsertNode(&F,J);
        Jstar=DeleteNode(&sorted_list,jstarposn);
        listsize--;
        InsertNode(&F,Jstar);
    }
    else if(K==2){

        temp_node1=sorted_list;
        r=0;
        bestsol=0;

        while(temp_node1!=NULL){

            temp_node2=temp_node1->link;
```

```
    s=r+1;
    while(temp_node2!=NULL){

        if(((temp_node1->value + temp_node2->value)>bestsol)&&
          ((J->value + temp_node1->value + temp_node2-
                >value)<=knapcap)){
            bestsol=temp_node1->value+temp_node2->value;
            A=r;
            B=s;
            wja=temp_node1->value;
            wjb=temp_node2->value;
        }

        s++;
        temp_node2=temp_node2->link;
    }

    r++;
    temp_node1=temp_node1->link;
}

counter=0;
temp_node=sorted_list;
wjb1=0;
wjb2=0;
for (i=0;i<listsize;i++){

    if(counter==(B-2))
        wjb2=temp_node->value;

    if(counter==(B-1))
        wjb1=temp_node->value;

    counter++;
    temp_node=temp_node->link;
}
if(wjstar>= (wja+wjb)){
    InsertNode(&F,J);
    Jstar=DeleteNode(&sorted_list,jstarposn);
    listsize--;
    InsertNode(&F,Jstar);

}
else if ( (wjstar == wja) && ( ((B-A)<=2) || ((J->value +
            wjb2+wjb1)>knapcap) ) ){
    InsertNode(&F,J);
```

```
                    Jb=DeleteNode(&sorted_list,B);
                    listsize--;
                    Ja=DeleteNode(&sorted_list,A);
                    listsize--;
                    InsertNode(&F,Jb);
                    InsertNode(&F,Ja);

                }

            }

        }

        if (F==NULL){
            InsertNode(&unassigned,J);

        }
        else{
            zr++;

            counter=0;
            temp_node=F;
            while(temp_node!=NULL){
                temp_node->binnum=zr;
                bj[temp_node->index]=zr;
                counter++;
                temp_node=temp_node->link;
            }

            for(i=0;i<counter;i++){
                deleted_node=DeleteNode(&F,0);
                InsertNode(&assigned, deleted_node);
            }
        }

    }
}

    ListDestroy(assigned);
    ListDestroy(unassigned);
    return zr;

}


/*compute L2 bound from Martello&Toth (1990)*/
```

```
int MTL2(int knapcap,int cnumbitem, int itemlist[])

{
    Node * sorted_list=NULL;
    Node * temp_node=NULL;
    int i,count;
    int jmin=0;
    int sum=0;
    int L1=0;
    int L2=0;
    int cj12=0;
    int sjstar=0;
    int jp=0;
    int cj2=0;
    int sj2=0;
    int sum1=0;
    int sum2=0;
    int jpp=0;
    int sj3=0;

        for(i=0;i<cnumbitem;i++){
            sorted_list=AddNode(itemlist[cnumbitem-1-i],sorted_list,i);
        }

        count=1;
        temp_node=sorted_list;
        for(i=0;i<cnumbitem;i++){
            if(temp_node->value<=(knapcap/2))
                break;
            count++;
            temp_node=temp_node->link;
        }
        jmin=count;

        L1= CompBinLoad(sorted_list)/knapcap;
        if((CompBinLoad(sorted_list)%knapcap) >0)
            L1++;

        for (i=0;i<cnumbitem;i++){
            sum=sum+itemlist[i];
        }

        L1=sum/knapcap;
        if((sum%knapcap) >0)
            L1++;
```

```
count=1;
for(i=0;i<cnumbitem;i++){
    if(itemlist[i]<=(knapcap/2))
        break;
    count++;
}
jmin=count;

if (jmin==1)
    L2=L1;
else{

    cj12=jmin-1;
    sjstar=0;

    for(i=(jmin-1);i<cnumbitem;i++)
        sjstar=sjstar+itemlist[i];

    for(i=0;i<jmin;i++)
        if (itemlist[i]<=(knapcap-itemlist[jmin-1])){
            jp=i+1;
            break;
        }
        else
            jp=jmin;

    cj2=jmin-jp;
    sj2=0;

    for(i=(jp-1);i<(jmin-1);i++)
      sj2=sj2+itemlist[i];

    jpp=jmin;
    sj3=itemlist[jpp-1];

    itemlist[cnumbitem]=0;
    while(itemlist[jpp]==itemlist[jpp-1]){
        jpp=jpp++;
        sj3=sj3+itemlist[jpp];
    }

    L2=cj12;

    sum2=(int)(sjstar+sj2)/knapcap;
    if(((sjstar+sj2)%knapcap) >1)
```

```
            sum2++;

        while( (jpp<=cnumbitem)&&((cj12+sum2-cj2)>(L2) )){

            sum1=(int)(sj3+sj2)/knapcap;
            if(((sj3+sj2)%knapcap) >1)
                sum1++;

            if (L2<(cj12+sum1-cj2))
                L2=cj12+sum1-cj2;

            jpp++;

            if(jpp<=cnumbitem){

                sj3=sj3+itemlist[jpp-1];

                while(itemlist[jpp]==itemlist[jpp-1]){
                    jpp++;
                    sj3=sj3+itemlist[jpp-1];
                }

                while ( (jp>1) && (itemlist[jp-2]<=(knapcap-itemlist[jpp-1])) ){
                    jp--;
                    cj2++;
                    sj2=sj2+itemlist[jp-1];

                }

            }

            sum2=(sjstar+sj2)/knapcap;
            if (((sjstar+sj2)%knapcap)>1)
                sum2++;

        }

    }

    ListDestroy(sorted_list);
    return L2;

}
```

```
/* compute a priori upper bounds*/
int AprioriUpperbound(Node * root, int numitemgen, int numknapsack, int knapcap,int
cnumbitem)
{
  int sum=0;
  int i,j;
  int upperbound0;
  int upperbound1;
  int upperbound2;
  int bestbound;
  int bound_knapsack;
  int bound_item;

  Node *temp_node;

        /*Upper bound U0*/
        sum=0;
        temp_node=root;
        for(i=0;i<numitemgen;i++){
            sum = sum + temp_node->value;
            if (sum > (knapcap*numknapsack))
                break;
            temp_node=temp_node->link;
        }
        upperbound0=i;
        cnumbitem = i;
        bestbound=upperbound0;


        for (i=upperbound0; i<numitemgen; i++)
            DeleteNode(&root,(upperbound0));


        /*Upperbound U1*/
        upperbound1=numitemgen;
        for (j=0;j<numknapsack;j++){
            sum = 0;
            temp_node=root;
            for(i=0;i<cnumbitem;i++){
                sum = sum + temp_node->value;
                if (sum > (knapcap*(j+1)))
                    break;
                temp_node=temp_node->link;
            }

            bound_knapsack=i+(i/(j+1))*(numknapsack-(j+1));
```

```
            if (upperbound1>bound_knapsack)
                upperbound1=bound_knapsack;
        }
        if (upperbound1<bestbound)
            bestbound=upperbound1;

        /*Upper bound U2*/
        upperbound2=numitemgen;
        for(i=0;i<cnumbitem;i++){
            bound_item= i + numknapsack*(knapcap/ItemValue(&root, i));
            if (upperbound2>bound_item)
                upperbound2 = bound_item;
        }
        if (upperbound2<bestbound)
            bestbound=upperbound2;

        for (i=bestbound; i<cnumbitem; i++)
            DeleteNode(&root, bestbound);

        cnumbitem=bestbound;

        return bestbound;

}


/*generate the instances of Peeters et al.(2006)*/
int GenInstance(int maxsize,int minsize,int numknapsack,int knapcap,int rep, int
itemsize[][MAX_NUMBER_ITEM])
{

    int numberitem=0;
    int sum=0;
    int i=0;

    while (sum<=(knapcap*numknapsack)){
        itemsize[rep][i]=minsize+(int)((maxsize-minsize+1)*ran2(&idum));
        sum=sum+itemsize[rep][i];
        i++;
        numberitem++;
    }
    return numberitem;

}
```

```
void ListNodeDestroy(Node *nptr)
{ free(nptr);}


void ListDestroy(Node *list)
{
    Node * current = list;
    Node * temp;

    for(; current!=NULL; current=temp)
    {
        temp=current->link;
        ListNodeDestroy(current);
    }

}

int InsertNode_D( register Node **linkp, Node * inserted_node)
{
    register Node   *current;
    while( ( current = *linkp ) != NULL &&
        current->value < inserted_node->value )
        linkp = &current->link;

    inserted_node->link = current;
    *linkp = inserted_node;

    return TRUE;
}

int ItemValue(Node **linkp, int item_num)
{
   Node * current;

   int count=0;

    current = *linkp;

    while( current != NULL && count<item_num){
        count++;
        current=current->link;
    }

    return (current->value);

}
```

```
float ran2(long *idum)
{
    int j;
    long k;
    static long idum2=123456789;
    static long iy=0;
    static long iv[NTAB];
    float temp;

    if (*idum <= 0) {
        if (-(*idum) < 1) *idum=1;
        else *idum = -(*idum);
        idum2=(*idum);
        for (j=NTAB+7;j>=0;j--) {
            k=(*idum)/IQ1;
            *idum=IA1*(*idum-k*IQ1)-k*IR1;
            if (*idum < 0) *idum += IM1;
            if (j < NTAB) iv[j] = *idum;
        }
        iy=iv[0];
    }
    k=(*idum)/IQ1;
    *idum=IA1*(*idum-k*IQ1)-k*IR1;
    if (*idum < 0) *idum += IM1;
    k=idum2/IQ2;
    idum2=IA2*(idum2-k*IQ2)-k*IR2;
    if (idum2 < 0) idum2 += IM2;
    j=iy/NDIV;
    iy=iv[j]-idum2;
    iv[j] = *idum;
    if (iy < 1) iy += IMM1;
    if ((temp=AM*iy) > RNMX) return RNMX;
    else return temp;
}

float ran3(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=labs(MSEED-labs(*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;
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//*   cbp.h is the header file for WAMCBP.cpp
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

typedef  struct NODE   {
    struct NODE   *link;
    int value;
    int index;
    int binnum;
} Node;


#define SQ(a) (a*a)
#define  FALSE  0
#define TRUE 1
#define MAX_NUMBER_BIN 500
#define MAX_NUMBER_ITEM 1200
#define NSTK 30000


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

#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)
```

Appendix H

Multidimensional Knapsack Problem Code

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//*     WAMDKP.cpp is the program for solving the multidimensional knapsack problem
//*     It is to be complied with the following files (ANSI version) from Press et al.(2002)
//          1)    nrutil.cpp
//          2)    nrutil.h
//          3)    nr.h
//*     The header file mdkp.h is appended at the end of this program.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#include <io.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "nrutil.h"
#include "nr.h"
#include <sys\timeb.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include "mdkp.h"
#include <string.h>

Node * bin_root[MAX_NUMBER_BIN];
Node * sorted_obj;
FILE * fp;
FILE * op;
float binload[MAX_NUMBER_BIN];
float curcap [MAX_NUMBER_BIN];
float rhs[MAX_NUMBER_DIM];
float slack[MAX_NUMBER_DIM];
float matcoef[MAX_NUMBER_ITEM][MAX_NUMBER_ITEM];
float profit[MAX_NUMBER_ITEM];
float evalue[MAX_NUMBER_ITEM];
int bestsol[MAX_NUMBER_ITEM];

int nitembin[MAX_NUMBER_BIN];
float obj_value=0.0;
float best_obj_value=0.0;
float bound = 0.0;
```

```
float binsize = 0;
double comptime=0.0;
int done =0;
int optimum =0;
int numdim=0;

double TimeElapsed(struct _timeb *begin, struct _timeb *end);
int irbit1(unsigned long *iseed);
void PrintNode(Node * item);
void PrintAllNode(Node * root);
void PrintFileNode(Node * item);
void PrintFile(Node * root);
Node * CreatNode(int order, float profits, float relevance);
Node * AddNode(int order, float data, Node * bin);
Node * DeleteNode( Node **linkp, int item_num );
int InsertNode( register Node **linkp, Node * inserted_node);
float CompBinLoad(Node * root);
int CompBinItem(Node * root);
int mswap10(Node ** rootp1, Node ** rootp2, int i,int p, float weight[]);
int mswap11(Node ** rootp1, Node ** rootp2, int i,int p, float weight[]);
int mswap12(Node ** rootp1, Node ** rootp2, int i,int p, float weight[]);
int mswap22(Node ** rootp1, Node ** rootp2, int i,int p, float weight[]);

int
main(void)
{

    struct _timeb time1, time2;
    unsigned long iseed =111;
    long idum=-1;
    Node * deleted_node;
    Node * deleted_node1;
    Node * deleted_node2;
    Node * inserted_node;
    Node * current_node;

    int i,j,p,m,idec,rep;
    int feasible=0;
    int freq = 0;
    int numbitem=0;
    int improved=0;
    int numtype=0;
    int data;
    int batch=0;
    int swap=0;
```

```
float weight [MAX_NUMBER_BIN];

double T = 1;
double Tred =0.95;
float temp=0.0;
int numbin =2;
char *name;


if((op=fopen("out.txt", "w")) == NULL) {
    printf("Cannot open file.\n");
    exit(1);
}

/*Input file format – Chu and Beasley (1999)*/
name="5.100-00.txt";

if((fp=fopen( name, "r")) == NULL) {
    printf("Cannot open file.\n");
    exit(1);
}

fscanf(fp,"%d", &numbitem);
fscanf(fp, "%d", &numdim);
fscanf(fp,"%d", &data);

for(i=0;i<numbitem; i++){
    fscanf(fp, "%f", &profit[i]);
}

for(i=0;i<numdim;i++)
    for(j=0;j<numbitem;j++)
        fscanf(fp,"%f", &matcoef[i][j]);

for(i=0;i<numdim; i++){
    fscanf(fp, "%f", &rhs[i]);
}

for(i=0;i<numbitem;i++){
    temp=0.0;
    for (j=0;j<numdim;j++){
        temp=temp + matcoef[j][i]/rhs[j];
    }

    evalue[i]=profit[i]/temp;
}
```

```
batch=(int)(numbitem*0.2);

    _ftime(&time1);

for (rep=0;rep<2;rep++){

T=1;

    /* initialisation*/
    sorted_obj=NULL;
    deleted_node  =NULL;
    deleted_node1 =NULL;
    deleted_node2 =NULL;
    inserted_node =NULL;
    current_node  =NULL;

    for (i=0;i<numbin;i++)
        bin_root[i] = NULL;

    for( i=0;i<numdim;i++){
        slack[i]=rhs[i];

    }
    for( i=0;i<numbin;i++){
        binload[i]=0;
        nitembin[i]=0;
    }

    /*read data file*/
    for (i=0;i<numbitem;i++){
        inserted_node=CreatNode(i,profit[i],evalue[i]);
        InsertNode(&sorted_obj, inserted_node);

    }

    /*Initial solution*/
        obj_value=0;
        for (i=0;i<numbitem;i++){

            idec=(int)(batch*ran3(&idum));

            if (i>= (numbitem-(batch))) idec=0;

            deleted_node= DeleteNode(&sorted_obj,idec);
```

```
        feasible=1;

        for (j=0;j<numdim;j++)
            if (slack[j]< matcoef[j][deleted_node->index])
                feasible=0;

        if (feasible){

            InsertNode(&bin_root[0],deleted_node);
            nitembin[0]++;
            obj_value=obj_value+deleted_node->value;
            for (j=0;j<numdim;j++)
                slack[j]=slack[j]-matcoef[j][deleted_node->index];

        }
        else{

            InsertNode(&bin_root[1],deleted_node);
            nitembin[1]++;
        }

    }

Tred=0.95;

for (m=0;m<50;m++){

        /* update weight*/
            weight[0]= pow(1.05, T);
            weight[1]= 1.0;


        /*swapping items between the knapsack and bin 0)*/

        for (i=0;i<numbin;i++) {
            improved=0;

            for (p=0;p<numbin;p++){
                if (p!=i){

                    /*randomly select one of the four swap schemes*/

                    swap=(int)(4*ran3(&idum));

                    switch (swap){
```

```
                    case 0:
                        improved = mswap11(&bin_root[i], &bin_root[p], i, p, weight);
                        if (done) break;
                        break;

                    case 1:
                        improved = mswap12(&bin_root[i], &bin_root[p], i, p, weight);
                        if (done) break;
                        break;

                    case 2:
                        improved = mswap22(&bin_root[i], &bin_root[p], i, p, weight);
                        if (done) break;
                        break;

                    }
                }
            }
            if (done) break;
        }

        if(done) break;

        T=T*Tred;
    }

}
_ftime(&time2);


comptime= TimeElapsed(&time1,&time2);

printf("\n");
printf("Best objective function value = %f\n", best_obj_value);
printf("\n");
printf("Elapsed time    %f \n",  comptime);
printf("\n");
/*print output to files*/
fprintf(op, "after mswap  \n");

for(i =0; i <numbin; i++){
    fprintf(op, "%3d  ", i);
    PrintFile(bin_root[i]);
    fprintf(op, " \n");
}
fprintf(op,"\n");
```

```
    fprintf(op, "           slacks \n");
    for(i =0; i <numdim; i++){
        fprintf(op, "%3d  ", i);
        fprintf(op,"%10.0f\n", slack[i]);
    }
    fprintf(op,"\n");
    fprintf(op,"Best objective function value = %f\n", best_obj_value);
    fprintf(op,"Best Combination\n");
    for (i=0;i<nitembin[0];i++)
        fprintf(op,"%5d\n", bestsol[i]);


    fprintf(op,"\n");
    fprintf(op, "Subroutine Elapsed time:%f \n", comptime);
    fprintf(op, "Program Elapsed time:  %f\n", (float)clock()/CLOCKS_PER_SEC);

    fclose(fp);
    fclose(op);

    return EXIT_SUCCESS;

}



/* functions*/
/*swapping items between a knapsack and bin 0*/
int mswap22(Node ** rootp1, Node **rootp2, int i, int p, float weight[])
{

    double deltaf=0.0;
    int count=0;
    int feasible=0;
    int w=0;
    int m=0;
    int n=0;
    int j=0;
    int k=0;
    int q=0;
    int r=0;

    Node * deleted_node1;
    Node * deleted_node2;
    Node * deleted_node3;
    Node * deleted_node4;
    Node * current_node1;
```

```
Node * current_node2;
Node * current_node3;
Node * current_node4;
Node * current_node;

current_node1=*rootp1;

for(m=0;m<(nitembin[i]-1);m++){

    k=j+1;
    current_node2=current_node1->link;

    while (current_node2 != NULL){

        q=0;
        current_node3=*rootp2;

        for(n=0;n<(nitembin[p]-1);n++){

            r=q+1;
            current_node4=current_node3->link;

            while (current_node4!=NULL){


                /*check feasibility of swap*/
                feasible=1;

                if (i==0){
                    for (w=0;w<numdim;w++)
                    if (slack[w]< (matcoef[w][current_node3->index]+matcoef[w]
                        [current_node4->index]-matcoef[w][current_node1->index]-
                        matcoef[w][current_node2->index]))
                        feasible=0;
                }
                else{
                    for (w=0;w<numdim;w++)
                    if (slack[w]< (matcoef[w][current_node1->index]+matcoef[w]
                        [current_node2->index]-matcoef[w][current_node3->index]-
                        matcoef[w][current_node4->index]))
                        feasible=0;
                }

                if(feasible){

                    if(i==0)
```

```
    deltaf=(current_node3->value + current_node4->value)*weight[p]
          -(current_node1->value + current_node2->value)*weight[i];
else
    deltaf=(current_node1->value + current_node2->value)*weight[i]
          -(current_node3->value + current_node4->value)*weight[p];

if (deltaf>=0){

    if(i==0)
        obj_value=obj_value+(current_node3->value + current_node4-
            >value)-(current_node1->value + current_node2->value);
    else
        obj_value=obj_value+(current_node1->value + current_node2-
            >value)-(current_node3->value + current_node4->value);

    if (i==0)
        for (w=0;w<numdim;w++)
            slack[w]=slack[w]-matcoef[w][current_node3->index]-
                matcoef[w][current_node4->index] +matcoef[w]
                [current_node1->index]+matcoef[w][current_node2-
                >index];
    else
        for (w=0;w<numdim;w++)
            slack[w]=slack[w]-matcoef[w][current_node1->index]-
                matcoef[w][current_node2->index]+matcoef[w]
                [current_node3->index]+matcoef[w][current_node4-
                >index];

    deleted_node2= DeleteNode(rootp1,k);
    deleted_node4= DeleteNode(rootp2,r);
    deleted_node1= DeleteNode(rootp1,j);
    deleted_node3= DeleteNode(rootp2,q);

    InsertNode(rootp1,deleted_node3);
    InsertNode(rootp1,deleted_node4);
    InsertNode(rootp2,deleted_node1);
    InsertNode(rootp2,deleted_node2);

    if (obj_value>best_obj_value)
        best_obj_value=obj_value;
    current_node=bin_root[0];
    for(i=0;i<nitembin[0];i++){
        bestsol[i]=current_node->index;
        current_node=current_node->link;
    }
```

```
                        return TRUE;

                    }

                }
                r++;
                current_node4=current_node4->link;
            }
            q++;
            current_node3=current_node3->link;
        }
        k++;
        current_node2=current_node2->link;
    }
    j++;
    current_node1=current_node1->link;
}

return FALSE;
}

/*swapping items between a knapsack and bin 0*/
int mswap12(Node ** rootp1, Node **rootp2, int i, int p, float weight[])
{

    double deltaf=0.0;
    int count=0;
    int feasible=0;
    int w=0;
    int n=0;
    int j=0;
    int q=0;
    int r=0;

    Node * deleted_node1;
    Node * deleted_node2;
    Node * deleted_node3;
    Node * current_node1;
    Node * current_node2;
    Node * current_node3;
    Node * current_node;

    current_node1=*rootp1;

    while (current_node1!=NULL){
```

```
q=0;
current_node2=*rootp2;

for(n=0;n<(nitembin[p]-1);n++){

        r=q+1;
        current_node3=current_node2->link;

        while (current_node3!=NULL){
            /*check feasibility of swap*/
            feasible=1;

            if (i==0){
                for (w=0;w<numdim;w++)
                if (slack[w]< (matcoef[w][current_node2->index]+matcoef[w]
                    [current_node3->index]-matcoef[w][current_node1->index]))
                    feasible=0;
            }
            else{
                for (w=0;w<numdim;w++)
                if (slack[w]< (matcoef[w][current_node1->index]-matcoef[w]
                    [current_node2->index]-matcoef[w][current_node3->index]))
                    feasible=0;
            }

            if (feasible) {

                if(i==0)
                    deltaf=(current_node2->value + current_node3->value)*weight[p]-
                        (current_node1->value)*weight[i];
                else
                    deltaf=(current_node1->value)*weight[i]-(current_node2->value +
                        current_node3->value)*weight[p];

                if (deltaf>=0){

                    if(i==0)
                        obj_value=obj_value+(current_node2->value + current_node3-
                            >value)-(current_node1->value);
                    else
                        obj_value=obj_value+(current_node1->value)-(current_node2-
                                >value + current_node3->value);

                    if (i==0)
                        for (w=0;w<numdim;w++)
                            slack[w]=slack[w]-matcoef[w][current_node2->index]-
```

```
                        matcoef[w][current_node3->index]+matcoef[w]
                        [current_node1->index];
                else
                    for (w=0;w<numdim;w++)
                        slack[w]=slack[w]-matcoef[w][current_node1->index]+
                            matcoef[w][current_node2->index]
                            +matcoef[w][current_node3->index];

                deleted_node3= DeleteNode(rootp2,r);
                deleted_node1= DeleteNode(rootp1,j);
                deleted_node2= DeleteNode(rootp2,q);

                InsertNode(rootp1,deleted_node2);
                InsertNode(rootp1,deleted_node3);
                InsertNode(rootp2,deleted_node1);

                /*update number of items in bins*/
                nitembin[p]--;
                nitembin[i]++;

                if (obj_value>best_obj_value)
                    best_obj_value=obj_value;
                current_node=bin_root[0];
                for(i=0;i<nitembin[0];i++){
                    bestsol[i]=current_node->index;
                    current_node=current_node->link;
                }

                    return TRUE;
                }
            }
            r++;
            current_node3=current_node3->link;
        }
        q++;
        current_node2=current_node2->link;
    }
    j++;
    current_node1=current_node1->link;
}

return FALSE;
}
```

```
/*swapping items between a knapsack and bin 0*/
int mswap11(Node ** rootp1, Node **rootp2, int i, int p, float weight[])
{

    double deltaf=0.0;
    int count =0;
    int feasible=0;
    int w=0;
    int j=0;
    int q=0;


    Node * deleted_node1;
    Node * deleted_node2;
    Node * current_node;
    Node * current_node1;
    Node * current_node2;

    current_node1=*rootp1;
    current_node2=*rootp2;

    while (current_node1!=NULL){

        q=0;
        current_node2=*rootp2;

        while (current_node2!=NULL){
            /*check feasibility of swap*/
            feasible=1;

            if (i==0){
                for (w=0;w<numdim;w++)
                    if (slack[w]< (matcoef[w][current_node2->index]-matcoef[w]
                        [current_node1->index]))
                        feasible=0;
            }
            else{
                for (w=0;w<numdim;w++)
                    if (slack[w]< (matcoef[w][current_node1->index]-matcoef[w]
                        [current_node2->index]))
                        feasible=0;
            }


            if (feasible){
```

```
        if(i==0)
            deltaf=(current_node2->value)*weight[p]-(current_node1->value)*
                weight[i];
        else
            deltaf=(current_node1->value)*weight[i]-(current_node2->value)*
                weight[p];

        if (deltaf>=0){

            if(i==0)
                obj_value=obj_value+(current_node2->value)-(current_node1->value);
            else
                obj_value=obj_value+(current_node1->value)-(current_node2->value);

        if (i==0)
                for (w=0;w<numdim;w++)
                    slack[w]=slack[w]-matcoef[w][current_node2->index]+matcoef[w]
                        [current_node1->index];
            else
                for (w=0;w<numdim;w++)
                    slack[w]=slack[w]-matcoef[w][current_node1->index]+matcoef[w]
                        [current_node2->index];

            deleted_node1= DeleteNode(rootp1,j);
            deleted_node2= DeleteNode(rootp2,q);

            InsertNode(rootp1,deleted_node2);
            InsertNode(rootp2,deleted_node1);

            if (obj_value>best_obj_value)
                best_obj_value=obj_value;
            current_node=bin_root[0];
            for(i=0;i<nitembin[0];i++){
                bestsol[i]=current_node->index;
                current_node=current_node->link;
            }

            return TRUE;

        }

    }
    q++;
    current_node2=current_node2->link;
}
j++;
```

```
        current_node1=current_node1->link;
    }

return FALSE;
}

/*swapping items between a knapsack and bin 0*/
int mswap10(Node ** rootp1, Node **rootp2, int i, int p, float weight[])
{
    int feasible=0;
    int w=0;
    int j=0;
    int count=0;
    double deltaf=0.0;

    Node * deleted_node1;
    Node * current_node;
    Node * current_node1;
    Node * current_node2;

    current_node1=*rootp1;
    current_node2=*rootp2;

    while (current_node1!=NULL){
        /*check feasibility of swap*/
        feasible=1;

        for (w=0;w<numdim;w++)
            if (slack[w]< (matcoef[w][current_node1->index]))
                feasible=0;

        if (feasible){

            deltaf=(current_node1->value)*weight[i];

            if (deltaf>0) {

                obj_value=obj_value+(current_node1->value);

                printf("mswap10:    obj_value = %f\n", obj_value);

                for (w=0;w<numdim;w++)
                    slack[w]=slack[w]-matcoef[w][current_node1->index];

                deleted_node1= DeleteNode(rootp1,j);
                InsertNode(rootp2,deleted_node1);
```

```
                /*update number of items in bins*/
                nitembin[p]++;
                nitembin[i]--;

                if (obj_value>best_obj_value)
                    best_obj_value=obj_value;
                current_node=bin_root[0];
                for(i=0;i<nitembin[0];i++){
                    bestsol[i]=current_node->index;
                    current_node=current_node->link;
                }

                return TRUE;

            }

        }

        j++;
        current_node1=current_node1->link;

    }

return FALSE;
}


int CompBinItem(Node * root)
{
    int count=0;
    if (root != NULL) {
        while (root != NULL) {
            count++;
            root = root->link;
        }
    } else {
        /*printf("Empty Bin\n");*/
    }
    return count;
}


float CompBinLoad(Node * root)
{
    float load=0;
```

```
    if (root != NULL) {
        while (root != NULL) {
            load = load+(root->value);
            root = root->link;
        }
    } else {
        /*printf(" Empty bin \n");*/
    }
    return load;
}



void PrintFileNode(Node * item)
{
    fprintf(op,"order=%5d,    value= %5.0f,    efficency= %5.0f\n", item->index, item-
>value, item->efficiency);
}


void PrintFile(Node * root)
{
    if (root != NULL) {
        while (root != NULL) {
            PrintFileNode(root);
            root = root->link;
        }
    } else {
        fprintf(op, "No nodes have been entered yet\n");
    }
}


void PrintNode(Node * item)
{
    printf("order=%5d,    value= %5.0f,    efficency= %5.0f\n", item->index, item->value,
item->efficiency);

}


void PrintAllNode(Node * root)
{
    if (root != NULL) {
        while (root != NULL) {
            PrintNode(root);
```

```
            root = root->link;
        }
    } else {
        printf("Error: No nodes have been entered yet!\n");
    }
}



Node * AddNode(int order, float data, Node * bin )
{
    struct NODE * temp = (struct NODE*) malloc(sizeof(struct NODE));
    temp->index = order;
    temp->value = data;
    temp->link = bin;
    bin = temp;
    return bin;
}


int InsertNode( register Node **linkp, Node * inserted_node)
{
    register Node   *current;
    while( ( current = *linkp ) != NULL &&
        current->efficiency > inserted_node->efficiency)
        linkp = &current->link;

    inserted_node->link = current;
    *linkp = inserted_node;

    return TRUE;
}


Node * DeleteNode( Node **linkp, int item_num )
{
    Node *current;
    Node *previous;
    Node *delnode;
    int count =0;

    current = *linkp;
    previous = NULL;

    while( current != NULL && count < item_num ){
        count++;
```

```
        previous = current;
        current = current->link;
    }

    delnode = current;

    if(previous==NULL)
        *linkp=current->link;
    else
        previous->link=delnode->link;

    if(current!=NULL)
        current = current->link;
    else
        current->link=NULL;

    return  delnode;
}



Node * CreatNode(int order, float profits, float relevance)
{
    struct NODE * temp = (struct NODE*) malloc(sizeof(struct NODE));

    temp->index=order;
    temp->value=profits;
    temp->efficiency=relevance;

    return temp;
}


/*random bit generator adapted from Press et al. (2006)*/
int irbit1(unsigned long *iseed)
{
    unsigned long newbit;

    newbit =  (*iseed >> 17) & 1
        ^ (*iseed >> 4) & 1
        ^ (*iseed >> 1) & 1
        ^ (*iseed & 1);
    *iseed=(*iseed << 1) | newbit;
    return (int) newbit;
}
```

```
double TimeElapsed(struct _timeb *begin, struct _timeb *end)
{float e=end->millitm, b=begin->millitm;
return (end->time+(e/1000))-(begin->time +(b/1000));}


/*random number generator adapted from Press et al. (2006)*/
float ran3(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=labs(MSEED-labs(*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;
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//  mdkp.h is the header file for WAMDKP.cpp
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
typedef  struct NODE   {
    struct NODE   *link;
    int index;
    float value;
    float efficiency;
} Node;


#define SQ(a) (a*a)
#define  FALSE  0
#define TRUE 1
#define MAX_NUMBER_BIN 2
#define MAX_NUMBER_DIM 30
#define MAX_NUMBER_ITEM 500
#define MAX_NUMBER_INSTANCE 30


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

Appendix I

Multidimensional Multiple Knapsack Problem Code

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//*     WAMDMKP.cpp is is the program for solving the multidimensional multiple
//knapsack problem. It generates instances, and computes the LP optimal solutions based
//on the simplex procedure as described in Press et al. (2002)
//*     It calls functions from WAMDKP.cpp
//*     It is to be complied with the following files (ANSI version) from Press et al.(2002)
//          1)     nrutil.cpp
//          2)     nrutil.h
//          3)     nr.h
//*     The header file mdmkp.h is appended at the end of the program.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#include <io.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "nrutil.h"
#include "nr.h"
#include <sys\timeb.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include "mdmkp.h"
#include <string.h>

Node * bin_root[MAX_NUMBER_BIN];
Node * sorted_obj;
FILE * fp;
FILE * op;
int binload[MAX_NUMBER_BIN];
int curcap [MAX_NUMBER_BIN];
int rhs[MAX_NUMBER_BIN][MAX_NUMBER_DIM];
int slack[MAX_NUMBER_BIN][MAX_NUMBER_DIM];
int matcoef[MAX_NUMBER_BIN][MAX_NUMBER_DIM][MAX_NUMBER_ITEM];
int profit[MAX_NUMBER_ITEM];
float evalue[MAX_NUMBER_ITEM];
int bestsol[MAX_NUMBER_BIN][MAX_NUMBER_ITEM];

float itemsize[MAX_NUMBER_ITEM][MAX_NUMBER_BIN];
```

```
float avgrhs[MAX_NUMBER_BIN];
float curlhs[MAX_NUMBER_BIN];

int nitembin[MAX_NUMBER_BIN];
int obj_value=0.0;
int best_obj_value=0.0;
int bound = 0.0;
int binsize =0.0;

double comptime=0.0;
int done =0;
int optimum =0;
int numdim=0;
int numknapsack=0;
int numbin =0;
double TimeElapsed(struct _timeb *begin, struct _timeb *end);
int irbit1(unsigned long *iseed);
void PrintNode(Node * item);
void PrintAllNode(Node * root);
void PrintFileNode(Node * item);
void PrintFile(Node * root);
Node * CreatNode(int order, float profits, float relevance);
Node * AddNode(int order, float data, Node * bin);
Node * DeleteNode( Node **linkp, int item_num );
int InsertNode( register Node **linkp, Node * inserted_node);
float CompBinLoad(Node * root);
int CompBinItem(Node * root);
int mswap10(Node ** rootp1, Node ** rootp2, int i,int p, float weight[]);
int mswap11(Node ** rootp1, Node ** rootp2, int i,int p, float weight[]);
int mswap12(Node ** rootp1, Node ** rootp2, int i,int p, float weight[]);
int mswap22(Node ** rootp1, Node ** rootp2, int i,int p, float weight[]);
int kswap11(Node ** rootp1, Node **rootp2, int i, int p, float weight[]);
int kswap12(Node ** rootp1, Node **rootp2, int i, int p, float weight[]);
int kswap22(Node ** rootp1, Node **rootp2, int i, int p, float weight[]);

void simplx(float **a, int m, int n, int m1, int m2, int m3, int *icase,int izrov[], int
iposv[]);

int
main(void)
{

    struct _timeb time1, time2;
    unsigned long iseed =111;
    long idum=-1;
    Node * deleted_node;
```

```
    Node * deleted_node1;
    Node * deleted_node2;
    Node * inserted_node;
    Node * current_node;
    Node * temp_node;
    int *icase;
    int *izrov;
    int *iposv;
    float **lp;

    int i,j,p,k,idec,rep,m, n, m1,m2,m3, count;
    int feasible=0;
    int freq = 0;
    int numbitem=0;
    int improved=0;
    int numtype=0;
    int batch=0;
    int swap=0;

    float weight [MAX_NUMBER_BIN];

    double T = 1;
    double Tred =0.95;
    float temp=0;
    float temp1=0.0;
    float temp2=0.0;
    float tightness=0.0;
    float LPbound=0.0;

    char *name;

/* generate instances*/
numbitem=10;
numknapsack=2;
numdim=2;
tightness=0.4;

for (k=1;k<=numknapsack;k++)
    for (i=0;i<numdim;i++)
        for (j=0;j<numbitem;j++)
            matcoef[k][i][j]=(int)1000*ran2(&idum);


for(k=1;k<=numknapsack;k++){
    for(i=0;i<numdim;i++){
        rhs[k][i]=0;
```

```
        for(j=0;j<numbitem;j++){
            rhs[k][i]=rhs[k][i]+matcoef[k][i][j];
        }
        rhs[k][i]=(int)(rhs[k][i]*tightness/(float)numknapsack);

    }
}


for(j=0;j<numbitem;j++){

    profit[j]=0;
    for (k=1;k<=numknapsack;k++){
        for (i=0;i<numdim;i++){
            profit[j]=profit[j]+matcoef[k][i][j];
        }
    }
    profit[j]=profit[j]/(numknapsack*numdim) + (int)500*ran2(&idum);

}



    if((op=fopen("out.txt", "w")) == NULL) {
        printf("Cannot open file.\n");
        exit(1);
    }

    for(i=0;i<numbitem;i++){
        temp=0;

        for (k=1;k<=numknapsack;k++){
            for (j=0;j<numdim;j++){
                temp=temp + (float)matcoef[k][j][i]/(float)rhs[k][j];

            }
        }
        evalue[i]=profit[i]/(float)temp;

    }


    for(i=0;i<numbitem;i++){
        for(k=1;k<=numknapsack;k++){
            temp1=0.0;
            for (j=0;j<=numdim;j++){
```

```
                temp1=temp1+matcoef[k][j][i];
            }
            itemsize[i][k]=temp1/(float)numdim;
        }
    }

    for(k=1;k<=numknapsack;k++){
        temp2=0.0;
        for (j=0;j<numdim;j++){
            temp2=temp2+rhs[k][j];

        }
        avgrhs[k]=temp2/(float)numdim;
    }

    for (i=0;i<MAX_NUMBER_BIN;i++)
        for(j=0;j<MAX_NUMBER_ITEM;j++)
            bestsol[i][j]=1000;


/*LP Solution*/
    n=numbitem*numknapsack;
    m=numknapsack*numdim + numbitem;
    m1=m;
    m2=0;
    m3=0;
    icase=ivector(1,1);
    icase[1]=0;
    iposv=ivector(1, (m+2));
    for(i=1;i<=(m+2);i++)
        iposv[i]=0;
        izrov=ivector(1,(n+1));
    for(i=1;i<=(n+1);i++)
        izrov[i]=0;
    lp=matrix(1,(m+2),1,(n+1));
    for(i=1;i<=(m+2);i++)
        for(j=1;j<=(n+1);j++)
            lp[i][j]=0;

    /*input LP tableau*/


for(i=0;i<numbitem;i++){
    count=2+i*numknapsack;
    for(j=count; j<=(count+numknapsack);j++)
        lp[1][j]=profit[i];
```

```
}

for(i=1;i<=numknapsack;i++){
    count=2+(i-1)*numdim;
    for(j=count;j<=(count+numdim-1);j++){
        lp[j][1] = rhs[i][j-(i-1)*numdim-2];
        for(k=1;k<=numbitem;k++){
            lp[j][2+(k-1)*numknapsack+(i-1)]=-matcoef[i][j-(i-1)*numdim-2][k-1];
        }
    }
}


count=2+numknapsack*numdim;
for(i=(count);i<=(count+numbitem);i++){
    lp[i][1]=1;
    for (j=1;j<=numknapsack;j++){
        lp[i][(i-count)*numknapsack+j+1]=-1;
    }

}

        simplx(lp, m, n, m1, m2, m3, icase, izrov, iposv);
        LPbound=lp[1][1];
        printf("LP bound = %f\n", LPbound);

batch=(int)(numbitem*0.21);

    _ftime(&time1);

for (rep=0;rep<1;rep++){

T=1;

    /* initialisation*/
    sorted_obj=NULL;
    deleted_node  =NULL;
    deleted_node1 =NULL;
    deleted_node2 =NULL;
    inserted_node =NULL;
    current_node  =NULL;

    numbin=numknapsack+1;

    for (i=0;i<numbin;i++)
        bin_root[i] = NULL;
```

```
for(k=1;k<=numknapsack;k++){
    for( i=0;i<numdim;i++){
        slack[k][i]=rhs[k][i];

    }
}

for( i=0;i<numbin;i++){
    curlhs[i]=0;
    binload[i]=0;
    nitembin[i]=0;
}

/*input data*/
for (i=0;i<numbitem;i++){
    inserted_node=CreatNode(i,profit[i],evalue[i]);
    InsertNode(&sorted_obj, inserted_node);

}

/*initial solution*/
    obj_value=0;
    for (i=0;i<numbitem;i++){
        idec=(int)(batch*ran2(&idum));
        if (i>= (numbitem-batch)) idec=0;
        deleted_node= DeleteNode(&sorted_obj,idec);
        for(k=1;k<=numknapsack;k++){
            feasible=1;
            for (j=0;j<numdim;j++){
                if (slack[k][j]< matcoef[k][j][deleted_node->index]){
                    feasible=0;
                    break;
                }
            }

            if (feasible){

                InsertNode(&bin_root[k],deleted_node);
                nitembin[k]++;
                obj_value=obj_value+deleted_node->profit;
                for (j=0;j<numdim;j++)
                    slack[k][j]=slack[k][j]-matcoef[k][j][deleted_node->index];

                break;
            }
```

```
        }

        if (feasible == 0){
            InsertNode(&bin_root[0],deleted_node);
            nitembin[0]++;

        }

    }

    /* update best solution*/

    if (obj_value>best_obj_value){
        best_obj_value=obj_value;
        current_node=NULL;
        for (i=0;i<numbin;i++){
            current_node=bin_root[i];
            for(j=0;j<nitembin[i];j++){
                bestsol[i][j]=current_node->index;
                current_node=current_node->link;
            }
        }
    }
    /*Update the current left hand side value*/
    current_node=NULL;
    for(k=1;k<=numknapsack;k++){
        current_node=bin_root[k];
        curlhs[k]=0;
        for(j=0;j<nitembin[k];j++){
            curlhs[k]=curlhs[k]+itemsize[current_node->index][k];
            /*printf("curlhs[k] == %10.2f\n", curlhs[k]);*/
            current_node=current_node->link;
        }
    }

Tred=0.95;

    for (m=0;m<100;m++){

        /*swapping items between knapsacks and bin 0*/

        for (i=0;i<numbin;i++) {
            improved=0;

            for (p=0;p<numbin;p++){
```

```
if (p!=i){
    /*swapping items between a knapsack and bin 0*/

    if((p==0)||(i==0)){

        weight[0]= pow(1.05, T);

        if (p!=0)
            weight[p]= 1.0;
        if (i!=0)
            weight[i]= 1.0;


        swap=(int)(4*ran3(&idum));
        switch (swap){

            case 0:
                improved = mswap10(&bin_root[i], &bin_root[p], i, p,
                                                    weight);
                if (done) break;
                break;

            case 1:
                improved = mswap11(&bin_root[i], &bin_root[p], i, p,
                                                    weight);
                if (done) break;
                break;

            case 2:
                improved = mswap12(&bin_root[i], &bin_root[p], i, p,
                                                    weight);
                if (done) break;
                break;

            case 3:
                improved = mswap22(&bin_root[i], &bin_root[p], i, p,
                                                    weight);
                if (done) break;
                break;

        }
    }

    else{
        /*swapping items between knapsacks*/
        weight[i]= pow((1.0-((avgrhs[i]-curlhs[i])/avgrhs[i])*0.01), T);
```

```
                    weight[p]= pow((1.0-((avgrhs[p]-curlhs[p])/avgrhs[p])*0.01), T);

                    swap=(int)(3*ran3(&idum));
                    switch (swap){

                        case 0:

                            improved = kswap11(&bin_root[i], &bin_root[p], i, p,
                                               weight);
                            if (done) break;
                            break;

                        case 1:
                            improved = kswap12(&bin_root[i], &bin_root[p], i, p,
                                               weight);
                            if (done) break;
                            break;

                        case 2:
                            improved = kswap22(&bin_root[i], &bin_root[p], i, p,
                                               weight);
                            if (done) break;
                            break;
                    }

                }

            }
        }
        if (done) break;

    }

    if(done) break;

    T=T*Tred;
  }

}
_ftime(&time2);

comptime= TimeElapsed(&time1,&time2);
printf("\n");
printf("LP bound = %f\n", LPbound);
printf("\n");
printf("Best objective function value = %d\n", best_obj_value);
```

```
    printf("\n");
    printf("Percentage = %2.2f %", (LPbound-best_obj_value)/LPbound*100);
    printf("\n");
    printf("Elapsed time    %f \n",  comptime);
    printf("\n");

    /*print output to files*/

    fprintf(op,"\n");
    fprintf(op,"LP bound = %f\n", LPbound);
    fprintf(op,"\n");
    fprintf(op,"\n");
    fprintf(op,"Best objective function value = %d\n", best_obj_value);
    fprintf(op,"Best Combination\n");

    for (i=0;i<numbin; i++){
        fprintf(op,"\n");
        fprintf(op,"bin number = %d\n", i);
        for (j=0;j<(nitembin[i]+2);j++)
            fprintf(op,"%5d\n", bestsol[i][j]);
    }

    fprintf(op,"\n");
    fprintf(op, "Subroutine Elapsed time:%f \n", comptime);
    fprintf(op, "Program Elapsed time:  %f\n", (float)clock()/CLOCKS_PER_SEC);

    fclose(fp);
    fclose(op);

    return EXIT_SUCCESS;

}


/* functions*/

/*swapping items between knapsacks*/
int kswap22(Node ** rootp1, Node **rootp2, int i, int p, float weight[])
{

    double deltaf=0.0;
    int count=0;
    int feasible=0;
    int w=0;
    int m=0;
    int n=0;
```

```
int j=0;
int k=0;
int q=0;
int r=0;

Node * deleted_node1;
Node * deleted_node2;
Node * deleted_node3;
Node * deleted_node4;
Node * current_node1;
Node * current_node2;
Node * current_node3;
Node * current_node4;
Node * current_node;

current_node1=*rootp1;

for(m=0;m<(nitembin[i]-1);m++){

    k=j+1;
    current_node2=current_node1->link;

    while (current_node2 != NULL){

        q=0;
        current_node3=*rootp2;

        for(n=0;n<(nitembin[p]-1);n++){

            r=q+1;
            current_node4=current_node3->link;

            while (current_node4!=NULL){

                /*check feasibility of a swap*/
                feasible=1;

                for (w=0;w<numdim;w++)
                    if (slack[i][w]< (matcoef[i][w][current_node3->index]+matcoef[i][w]
                        [current_node4->index]-matcoef[i][w][current_node1->index]-
                        matcoef[i][w][current_node2->index])){
                        feasible=0;
                        break;
                    }

                for (w=0;w<numdim;w++)
```

```
    if (slack[p][w]< (matcoef[p][w][current_node1->index]+matcoef[p]
        [w][current_node2->index]-matcoef[p][w][current_node3->index]-
        matcoef[p][w][current_node4->index])){
        feasible=0;
        break;
    }


if(feasible==1){

deltaf= (itemsize[current_node3->index][i]*weight[i]+itemsize
    [current_node4->index][i]*weight[i]- itemsize[current_node1->index]
    [i]*weight[i]-itemsize[current_node2->index][i]*weight[i])+
    (itemsize[current_node1->index][p]*weight[p]+itemsize
    [current_node2->index][p]*weight[p]- itemsize[current_node3
    ->index][p]*weight[p]-itemsize[current_node4->index][p]*weight[p]);


    if (deltaf<0){

        for (w=0;w<numdim;w++){
            slack[i][w]=slack[i][w]+matcoef[i][w][current_node1-
                >index]+matcoef[i][w][current_node2->index]
                -matcoef[i][w][current_node3->index]-
                matcoef[i][w][current_node4->index];

            slack[p][w]=slack[p][w]+matcoef[p][w][current_node3-
                >index]+matcoef[p][w][current_node4->index]
                -matcoef[p][w][current_node1->index]-matcoef[p]
                [w][current_node2->index];
        }

        deleted_node2= DeleteNode(rootp1,k);
        deleted_node4= DeleteNode(rootp2,r);
        deleted_node1= DeleteNode(rootp1,j);
        deleted_node3= DeleteNode(rootp2,q);

        InsertNode(rootp1,deleted_node3);
        InsertNode(rootp1,deleted_node4);
        InsertNode(rootp2,deleted_node1);
        InsertNode(rootp2,deleted_node2);

        return TRUE;
    }

}
```

```
                r++;
                current_node4=current_node4->link;
            }
            q++;
            current_node3=current_node3->link;
        }
        k++;
        current_node2=current_node2->link;
    }
    j++;
    current_node1=current_node1->link;
}

return FALSE;
}

/*swapping items between knapsacks*/
int kswap12(Node ** rootp1, Node **rootp2, int i, int p, float weight[])
{

    double deltaf=0.0;
    int count=0;
    int feasible=0;
    int w=0;
    int n=0;
    int j=0;
    int q=0;
    int r=0;

    Node * deleted_node1;
    Node * deleted_node2;
    Node * deleted_node3;
    Node * current_node1;
    Node * current_node2;
    Node * current_node3;
    Node * current_node;

    current_node1=*rootp1;

    while (current_node1!=NULL){

        q=0;
        current_node2=*rootp2;

        for(n=0;n<(nitembin[p]-1);n++){
```

```
r=q+1;
current_node3=current_node2->link;

while (current_node3!=NULL){
    /*check feasibility of a swap*/
    feasible=1;

    for (w=0;w<numdim;w++)
        if (slack[i][w]< (matcoef[i][w][current_node2->index]+matcoef[i]
            [w][current_node3->index]-matcoef[i][w][current_node1->index])){
            feasible=0;
            break;
        }
    for (w=0;w<numdim;w++)
        if (slack[p][w]< (matcoef[p][w][current_node1->index]-matcoef[p]
            [w][current_node2->index]-matcoef[p][w][current_node3-
            >index])){
            feasible=0;
            break;
        }

    if (feasible==1) {

    deltaf= (itemsize[current_node2->index][i]*weight[i] + itemsize
        [current_node3->index][i]*weight[i]- itemsize[current_node1
        ->index][i]*weight[i])+(itemsize[current_node1->index][p]*weight[p]-
         itemsize[current_node2->index][p]*weight[p]-itemsize[current_node3
        ->index][p]*weight[p]);

        if (deltaf<0){

            curlhs[i]=curlhs[i]-itemsize[current_node1->index][i] +itemsize
                [current_node2->index][i]+ itemsize[current_node3->index][i];
           curlhs[p]=curlhs[p]-itemsize[current_node2->index][p] –itemsize
                [current_node3->index][p]+itemsize[current_node1->index][p];

            for (w=0;w<numdim;w++){
                slack[i][w]=slack[i][w]+matcoef[i][w][current_node1->index]-
                    matcoef[i][w][current_node2->index]-matcoef[i][w]
                    [current_node3->index];
                slack[p][w]=slack[p][w]+matcoef[p][w][current_node2->index]+
                    matcoef[p][w][current_node3->index]-matcoef[p][w]
                    [current_node1->index];
            }

            deleted_node3= DeleteNode(rootp2,r);
```

```
                        deleted_node1= DeleteNode(rootp1,j);
                        deleted_node2= DeleteNode(rootp2,q);

                        InsertNode(rootp1,deleted_node2);
                        InsertNode(rootp1,deleted_node3);
                        InsertNode(rootp2,deleted_node1);

                        /*update number of items in bins*/
                        nitembin[p]--;
                        nitembin[i]++;

                        return TRUE;
                    }
                }
                r++;
                current_node3=current_node3->link;
            }
        q++;
        current_node2=current_node2->link;
    }
    j++;
    current_node1=current_node1->link;
}

return FALSE;
}

/*swapping items between knapsacks*/
int kswap11(Node ** rootp1, Node **rootp2, int i, int p, float weight[])
{

    float deltaf;
    int j=0;
    int q=0;
    int w=0;
    int m,k;
    int feasible=0;

    Node * deleted_node1;
    Node * deleted_node2;
    Node * current_node1;
    Node * current_node2;

    current_node1=*rootp1;
    current_node2=*rootp2;
```

```
while (current_node1!=NULL){

    q=0;
    current_node2=*rootp2;

    while (current_node2!=NULL){
        /*check feasibility of a swap*/
        feasible=1;

        for (w=0;w<numdim;w++)
            if (slack[i][w]< (matcoef[i][w][current_node2->index]-matcoef[i][w]
                [current_node1->index])){
                feasible=0;
                break;
            }
        for (w=0;w<numdim;w++)
            if (slack[p][w]< (matcoef[p][w][current_node1->index]-matcoef[p][w]
                [current_node2->index])){
                feasible=0;
                break;
            }


        if (feasible==1){

            deltaf=(itemsize[current_node2->index][i]*weight[i]-itemsize[current_node1
                ->index][i]*weight[i])+ (itemsize[current_node1->index][p]*weight[p]-
                itemsize[current_node2->index][p]*weight[p]);

            if (deltaf<0) {

                curlhs[i]=curlhs[i]-itemsize[current_node1->index][i] +itemsize
                        [current_node2->index][i];
                curlhs[p]=curlhs[p]-itemsize[current_node2->index][p] +itemsize
                        [current_node1->index][p];

                for (w=0;w<numdim;w++){
                    slack[i][w]=slack[i][w]+matcoef[i][w][current_node1->index]-
                            matcoef[i][w][current_node2->index];
                    slack[p][w]=slack[p][w]+matcoef[p][w][current_node2->index]-
                            matcoef[p][w][current_node1->index];
                }

                deleted_node1= DeleteNode(rootp1,j);
                deleted_node2= DeleteNode(rootp2,q);
```

```
                    InsertNode(rootp1,deleted_node2);
                    InsertNode(rootp2,deleted_node1);

                    return TRUE;
                }

            }
            q++;
            current_node2=current_node2->link;
        }
        j++;
        current_node1=current_node1->link;
    }

return FALSE;
}


void PrintFileNode(Node * item)
{
    fprintf(op,"order=%5d,    profit= %5.0f,    efficency= %5.0f\n", item->index, item-
>profit, item->efficiency);
}

void PrintNode(Node * item)
{
    printf("order=%5d,    profit= %5.0f,    efficency= %5.0f\n", item->index, item->profit,
item->efficiency);

}


Node * AddNode(int order, float data, Node * bin )
{
    struct NODE * temp = (struct NODE*) malloc(sizeof(struct NODE));
    temp->index = order;
    temp->profit = data;
    temp->link = bin;
    bin = temp;
    return bin;
}


int InsertNode( register Node **linkp, Node * inserted_node)
{
    register Node  *current;
```

```
    while( ( current = *linkp ) != NULL &&
       current->efficiency > inserted_node->efficiency)
        linkp = &current->link;

    inserted_node->link = current;
    *linkp = inserted_node;

    return TRUE;
}


Node * CreatNode(int order, float profits, float relevance)
{
    struct NODE * temp = (struct NODE*) malloc(sizeof(struct NODE));

    temp->index=order;
    temp->profit=profits;
    temp->efficiency=relevance;

    return temp;
}

/*simplx function – adapted from Press et al. (2002)*/

void simplx(float **a, int m, int n, int m1, int m2, int m3, int *icase,
    int izrov[], int iposv[])
{
    void simp1(float **a, int mm, int ll[], int nll, int iabf, int *kp,
        float *bmax);
    void simp2(float **a, int m, int n, int *ip, int kp);
    void simp3(float **a, int i1, int k1, int ip, int kp);
    int i,ip,is,k,kh,kp,nl1;
    int *l1,*l3;
    float q1,bmax;


    if (m != (m1+m2+m3)) nrerror("Bad input constraint counts in simplx");
    l1=ivector(1,n+1);
    l3=ivector(1,m);
    nl1=n;
    for (k=1;k<=n;k++) l1[k]=izrov[k]=k;
    for (i=1;i<=m;i++) {
        if (a[i+1][1] < 0.0) nrerror("Bad input tableau in simplx");
        iposv[i]=n+i;
    }
    if (m2+m3) {
```

```
    for (i=1;i<=m2;i++) l3[i]=1;
    for (k=1;k<=(n+1);k++) {
        q1=0.0;
        for (i=m1+1;i<=m;i++) q1 += a[i+1][k];
        a[m+2][k] = -q1;
    }
    for (;;) {
        simp1(a,m+1,l1,nl1,0,&kp,&bmax);
        if (bmax <= EPS && a[m+2][1] < -EPS) {
            *icase = -1;
            FREEALL return;
        } else if (bmax <= EPS && a[m+2][1] <= EPS) {
            for (ip=m1+m2+1;ip<=m;ip++) {
                if (iposv[ip] == (ip+n)) {
                    simp1(a,ip,l1,nl1,1,&kp,&bmax);
                    if (bmax > EPS)
                        goto one;
                }
            }
            for (i=m1+1;i<=m1+m2;i++)
                if (l3[i-m1] == 1)
                    for (k=1;k<=n+1;k++)
                        a[i+1][k] = -a[i+1][k];
            break;
        }
        simp2(a,m,n,&ip,kp);
        if (ip == 0) {
            *icase = -1;
            FREEALL return;
        }
one:    simp3(a,m+1,n,ip,kp);
        if (iposv[ip] >= (n+m1+m2+1)) {
            for (k=1;k<=nl1;k++)
                if (l1[k] == kp) break;
            --nl1;
            for (is=k;is<=nl1;is++) l1[is]=l1[is+1];
        } else {
            kh=iposv[ip]-m1-n;
            if (kh >= 1 && l3[kh]) {
                l3[kh]=0;
                ++a[m+2][kp+1];
                for (i=1;i<=m+2;i++)
                    a[i][kp+1] = -a[i][kp+1];
            }
        }
        is=izrov[kp];
```

```
            izrov[kp]=iposv[ip];
            iposv[ip]=is;
        }
    }
    for (;;) {
        simp1(a,0,l1,nl1,0,&kp,&bmax);
        if (bmax <= EPS) {
            *icase=0;
            FREEALL return;
        }
        simp2(a,m,n,&ip,kp);
        if (ip == 0) {
            *icase=1;
            FREEALL return;
        }
        simp3(a,m,n,ip,kp);
        is=izrov[kp];
        izrov[kp]=iposv[ip];
        iposv[ip]=is;
    }
}

/*simplex 1 – adapted from Press et al. (2002)*/

void simp1(float **a, int mm, int ll[], int nll, int iabf, int *kp,
    float *bmax)
{
    int k;
    float test;

    if (nll <= 0)
        *bmax=0.0;
    else {
        *kp=ll[1];
        *bmax=a[mm+1][*kp+1];
        for (k=2;k<=nll;k++) {
            if (iabf == 0)
                test=a[mm+1][ll[k]+1]-(*bmax);
            else
                test=fabs(a[mm+1][ll[k]+1])-fabs(*bmax);
            if (test > 0.0) {
                *bmax=a[mm+1][ll[k]+1];
                *kp=ll[k];
            }
        }
    }
```

```
}


/*simplex 2 – adapted from Press et al. (2002)*/
void simp2(float **a, int m, int n, int *ip, int kp)
{
    int k,i;
    float qp,q0,q,q1;

    *ip=0;
    for (i=1;i<=m;i++)
        if (a[i+1][kp+1] < -EPS) break;
    if (i>m) return;
    q1 = -a[i+1][1]/a[i+1][kp+1];
    *ip=i;
    for (i=*ip+1;i<=m;i++) {
        if (a[i+1][kp+1] < -EPS) {
            q = -a[i+1][1]/a[i+1][kp+1];
            if (q < q1) {
                *ip=i;
                q1=q;
            } else if (q == q1) {
                for (k=1;k<=n;k++) {
                    qp = -a[*ip+1][k+1]/a[*ip+1][kp+1];
                    q0 = -a[i+1][k+1]/a[i+1][kp+1];
                    if (q0 != qp) break;
                }
                if (q0 < qp) *ip=i;
            }
        }
    }
}
```

```
/*simplex 3 – adapted from Press et al. (2002)*/

void simp3(float **a, int i1, int k1, int ip, int kp)
{
    int kk,ii;
    float piv;

    piv=1.0/a[ip+1][kp+1];
    for (ii=1;ii<=i1+1;ii++)
        if (ii-1 != ip) {
            a[ii][kp+1] *= piv;
            for (kk=1;kk<=k1+1;kk++)
                if (kk-1 != kp)
                    a[ii][kk] -= a[ip+1][kk]*a[ii][kp+1];
        }
    for (kk=1;kk<=k1+1;kk++)
        if (kk-1 != kp) a[ip+1][kk] *= -piv;
    a[ip+1][kp+1]=piv;
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// mdmkp.h is the header file for WAMDMKP.cpp
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#define SQ(a) (a*a)
#define  FALSE  0
#define TRUE 1
#define MAX_NUMBER_BIN 10
#define MAX_NUMBER_DIM 20
#define MAX_NUMBER_ITEM 1000

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

typedef  struct NODE   {
    struct NODE   *link;
    int index;
    float profit;
    float efficiency;

} Node;


void simplx(float **a, int m, int n, int m1, int m2, int m3, int *icase,int izrov[], int
iposv[]);
void simp1(float **a, int mm, int ll[], int nll, int iabf, int *kp,float *bmax);
void simp2(float **a, int m, int n, int *ip, int kp);
void simp3(float **a, int i1, int k1, int ip, int kp);


#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)
```

References

E. Aarts and J.K. Lenstra, (eds), *Local Search in Combinatorial Optimization.* Princeton University Press, Princeton, New Jersey, (2003).

Y. Akcay and S.H. Xu, "Joint inventory replenishment and component allocation optimization in an assemble-to-order system," *Management Science* 50, 99-116 (2004).

A.C.E. Alvim, C.C. Ribeiro, F. Glover, and D.J. Aloise, "A hybrid improvement heuristic for the one-dimensional bin packing problem," *Journal of Heuristics* 10, 205-229 (2004).

B.E. Bengtsson, "Packing rectangle pieces – a heuristic approach," *The Computer Journal* 25, 353-357 (1982).

J.O. Berkey and P.Y. Wang, "Two dimensional finite bin packing algorithms," *Journal of the Operational Research Society* 38, 423-429 (1987).

M.A. Boschetti and A. Mingozzi, "Two-dimensional finite bin packing problems. Part II: New upper and lower bounds," *4OR*, 2, 135-147 (2003).

J.L. Bruno and P.J. Downey, "Probabilistic bounds for dual bin packing problem," *Acta Informatica* 22, 333-345 (1985).

J.M. Valério de Carvalho, "Exact solution of bin-packing problems using column generation and branch-and-bound," *Annals of Operations Research* 86, 629-659 (1999).

S. de Vries and R.V. Vokra, *Combinatorial auctions: A survey*, Discussion paper 1296, The Center for Mathematical Studies in Economics and Management Science, Northwestern University, (2001).

A. Caprara, H. Kellerer, U. Pferschy, and D. Pisinger, "Approximation algorithms for knapsack problems with cardinality constraints," *European Journal of Operational Research* 123, 333-345 (2000).

I. Charon and O. Hudry, "The noising method: A new method for combinatorial optimization," *Operations Research Letters,* 133-137 (1993).

P.C. Chu and J.E. Beasley, "A genetic algorithm for the multidimensional knapsack problem," *Journal of Heuristics* 4, 63-86 (1998).

F.K.R. Chung, M.R. Garey, and D.S. Johnson, "On packing two-dimensional bins," *SIAM Journal of Algebraic and Discrete Methods* 3, 66-76 (1982).

E.G. Coffman Jr., M.R.Garey, and D.S. Johnson, "Approximation algorithms for bin packing: A survey," in D.S. Hochbaum, (ed.) *Approximation Algorithm for NP-Hard Problems.* PWS Publishing, Boston, Massachusetts, 46-93 (1997).

E.G. Coffman Jr., J.Y.-T. Leung, and D.W. Ting, "Bin Packing: Maximizing the number of pieces packed," *Acta Informatica* 9, 263-271 (1978)

E.G. Coffman Jr. and J.Y.-T. Leung, "Combinatorial analysis of an efficient algorithm for processor and storage allocation," *SIAM Jounal on Computing* 8, 202-217 (1979).

S.P. Coy, B.L. Golden, and E.A. Wasil, "A computational study of smoothing heuristics for the traveling salesman problem," *European Journal of Operations Research* 124, 15-27 (2000).

M. Dell'Amico, S. Martello, and D. Vigo, "An exact algorithm for non-oriented two-dimensional bin packing problems," in preparation (1999).

S. Eilon and N. Christofides, "The loading problem," *Management Science* 17, 259-268 (1971).

A. El-Bouri, N. Popplewell, S. Balakrishnan, and A. Alfa, "A search based heuristic for the two-dimensional bin packing problem," *INFOR* 32, 265-274 (1994).

G. Elidan, M. Ninio, N. Freidman, and D. Schuurmans, "Data perturbation for escaping local maxima in learning," *AAAI-02/IAAI-02*, 132-139 (2002).

E. Falkenauer, "A hybrid grouping genetic algorithm for bin packing," *Journal of Heuristics* 2, 5-30 (1996).

O. Faroe, D. Pisinger, and M. Zachariasen, "Guided local search for the three-dimensional bin-packing problem," *INFORMS Journal on Computing* 15, 3:267-283, (2003).

C.M. Ferreira, A. Martin, and R. Weismantel, "Solving multiple knapsack problems by cutting planes," *SIAM Journal on Optimization* 6, 858-877 (1996).

K. Fleszar and K.S. Hindi, "New heuristics for one-dimensional bin packing," *Computers & Operations Research* 29(7), 821-839 (2002).

A. Fréville and G. Plateau, "An efficient preprocessing procedure for the multidimensional knapsack problem," *Discrete Applied Mathematics* 49, 189-212 (1994).

J.B. Frenk and G.G. Galambos, "Hybrid next-fit algorithm for the two-dimensional rectangle bin-packing problem," *Computing* 39, 201-217 (1987).

M.R. Garey and D.S Johnson, *Computers and intractability: A guide to the theory of NP-completeness*. W.H. Freeman and Company, San Francisco, California (1979).

J.N.D. Gupta and J.C. Ho, "A new heuristics algorithm for the one-dimensional bin-packing problem," *Production Planning and Control* 10(6), 598-603 (1999).

S. Hanafi and A. Fréville, "An efficient tabu search approach for the 0-1 multidimensional knapsack problem," *European Journal of Operational Research* 106, 659-675 (1998).

P. Hansen and N. Mladenović, "An introduction to variable neighborhood search," in S. Voss, S. Martello, I.H. Osman, and C. Roucairol (eds), *Metaheuristics : Advances and Trends in Local Search Procedures for Optimization*, Kluwer, Doudrecht, 433-458 (1999).

R. Hill and C. Reilly, "The effects of coefficients correlation structure in two-dimensional knapsack problems on solution procedure performance," *Management Sciences* 46, 302-317 (2000).

H. Kellerer, "A polynomial time approximation scheme for the multiple knapsack problem," *RANDOM-APPROX'99*, 51-62 (1999).

H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*. Springer-Verlag, Berlin, (2004).

A. Kleywegt and J.D. Papastavrou, "The dynamic and stochastic knapsack problem with random sized items', *Operations Research* 49, 26-41 (2001)

M. Labbé, G. Laporte, and S. Martello, "Upper bounds and algorithms for the maximum cardinality bin packing problem," *European Journal of Operational Research* 149, 490-498 (2003).

K. Lagus, I. Karanta, and J. Ylä-Jääski, "Paginating the generalized newspaper: A comparison of simulated annealing and a heuristic method," in *Proceedings of the 5th International Conference on Parallel Problem from Nature,* Berlin, 549-603 (1996).

A. Lodi, Private communication (2005).

A. Lodi, S. Martello, and D. Vigo, "Neighborhood search algorithm for the guillotine non-oriented two-dimensional bin packing problem," in S. Voss, S. Martello, I.H. Osman, and C. Roucairol, (eds.), *Metaheuristics: Advances and Trends in Local Search Paradigms for Optimization*, Kluwer Academic Publishers, Boston, Massachusetts, 125-139 (1998).

A. Lodi, S. Martello, and D. Vigo, "Approximation algorithms for the oriented two-dimensional bin packing problem," *European Journal of Operational Research* 112, 158-166 (1999a).

A. Lodi, S. Martello, and D. Vigo, "Heuristics and metaheuristics approaches for a class of two-dimensional bin packing problems," *INFORMS Journal on Computing* 11, 345-357 (1999b).

A. Lodi, S. Martello, and D. Vigo, "Heuristic algorithms for the three-dimensional bin packing problems," *European Journal of Operational Research* 141, 410-420 (2002a).

A. Lodi, S. Martello, and D. Vigo "Recent advances on two-dimensional bin packing problems," *Discrete Mathematics* 123, 379-396 (2002b).

R. Loulou and E. Michaelides, "New greedy-like heuristics for the multidimensional 0-1 knapsack problems – a parametric approach," *Operations Research* 27, 1101-1114 (1979).

L.L. Lu, S.Y. Chiu, and L.A. Cox, "Optimal project selection: Stochastic knapsack with finite time horizon," *Operations Research* 50, 645-650 (1999).

M.J. Magazine and O. Oguz, "A heuristic algorithm for the multidimensional zero-one knapsack problem," *European Journal of Operations Research* 16, 319-326 (1984).

S. Martello and P. Toth, *Knapsack Problems – Algorithms and Computer Implementations*. Wiley, Chichster (1990).

S. Martello and D. Vigo, "Exact solution of the two-dimensional finite bin packing problem," *Management Scien*ce 44, 388-399 (1998).

S. Martello and D. Vigo, "New computational results for the exact solution of the two-dimensional finite bin packing problem," Technical report OR/01/06, Dipartimento di Elettronica, Informatica e Sistemistica, Università di Bologna, Bologna, Italy (2001).

M. Monaci and P. Toth, "A set-covering-based heuristic approach for bin-packing problems," *INFORMS Journal on Computing*, 18, 71-85 (2006).

M. Ninio and J.J. Schneider, "Weight annealing," *Physica* A 349, 649-666 (2005).

M. Peeters and Z. Degraeve, "Branch-and-price algorithms for the dual bin packing and maximum cardinality bin packing problem," *European Journal of Operational Research* 170, 416-439 (2006).

C.C. Petersen, "Computational experience with variants of the Balas algorithm applied to the selection of R&D projects," *Management Science* 13, 736-750 (1967).

H. Pirkul, “A heuristic solution procedure for the multiconstraint zero-one knapsack problem,” *Naval Research Logistics* 34, 161-172 (1987).

H. Pirkul, and S. Narasimhan, “Efficient algorithms for the multiconstraint general knapsack problem,” *IIE Transactions* pp. 195-203 (1986).

D. Pisinger, “An exact algorithm for large multiple knapsack problems,” *European Journal of Operational Research* 114, 528-541 (1999).

W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes in C,* fourth edition, Cambridge University Press, New York, (2002).

J. Romaine, “Solving the multidimensional multiple knapsack problem with packing constraints using tabu search,” Master thesis, Air Force Institute of Technology, Wright Patterson AFB, Ohio (1999).

P. Schwerin and G. Wäscher, “The bin packing problem: a problem generator and some numerical experiments with FFD packing and MTP,” *International Transactions in Operational Research* 4, 377-389 (1997).

P. Schwerin and G. Wäscher, “A new lower bound for the bin packing problem and its integration into MTP,” *Pesquisa Operational* 19, 111-129 (1999).

A. Scholl, R. Klein, and C. Jürgens, “BISON: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem,” *Computers & Operations Research* 24(7), 627-45 (1997).

S. Senju and Y. Toyoda, “An approach to linear programming with 0-1 variables,” *Management Science* 15(4), B196-B207 (1968).

Y. Toyoda, “A simplified algorithm for obtaining approximate solutions to zero-one programming problems,” *Management Science* 21, 1417-1427 (1975).

A. Volgenant and J.A. Zoon, “An improved heuristic for the multidimensional 0-1 knapsack problems”, *Journal of Operational Research Society* 41, 963-970 (1990).

G. Wäscher and T. Gau, “Heuristics for the integer one-dimensional cutting stock problem: a computational study,” *OR Spektrum* 18, 131-144 (1996).

H.M. Weigartner and D.N. Ness, “Methods for the solution of the multidimensional 0/1 knapsack problem,” *Operational Research* 15, 83-103 (1967).