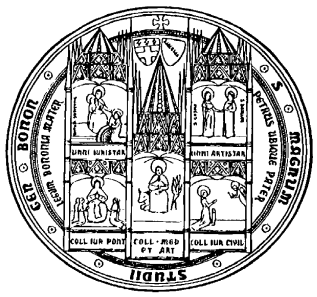


UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Dottorato di Ricerca in
Ingegneria dei Sistemi

XII Ciclo

Sede Amministrativa: Università di Bologna
Sedi Consorziate: Università di Firenze, Padova e Siena

Algorithms for Two-Dimensional Bin Packing and Assignment Problems

Andrea Lodi

Il Coordinatore
Prof. Giovanni Marro

I Tutori
Prof. Silvano Martello

Prof. Paolo Toth

AA. AA. 1996–1999

Contents

Acknowledgments

There are many people to acknowledge and I decided to do that in a reasonable amount of space.

First of all, my thanks go to my advisers, Silvano Martello and Paolo Toth, without whom this work would not have been possible. They have been able to set up, during the years, a research group in which an exciting scientific activity is performed in a really special and friendly surrounding. I am indebted to all the members of this enlarged group. To Daniele Vigo for his suggestions and his friendship, to Matteo Fischetti and Mauro Dell'Amico who spent so much time with me, and to Alberto Caprara, who not only offered me hospitality in his "office", but patiently answered all my questions and doubts. Michele Monaci asks me similar questions in these days. I thank him to force me to clarify my ideas.

I want to thank all the people who visited Bologna in these years, and in particular Juan José Salazar and David Pisinger, for the interesting scientific discussions, and a few very nice evenings.

Thanks to Professor Thomas M. Liebling for having invited me to Lausanne at the beginning of the PhD, and for having always encouraged and followed my research activity.

Many thanks to all the co-authors of the work presented in this thesis, and to all the undergraduate students I collaborated with, in particular to Matteo Porcù.

I don't want to forget the people at DEIS and their work supporting the research activity. Thanks for doing this so kindly.

Finally, I warmly want to thank my families.

Bologna, January 15, 2000

Andrea Lodi

List of Figures

List of Tables

Part I

Algorithms for Two-Dimensional Bin Packing Problems

Chapter 1

Outline of Part I

Informally speaking, the *Two-Dimensional Bin Packing Problem*, 2BP for short, is the problem of packing a finite set of "small" rectangles, called items, into the minimum number of identical "large" rectangles, called bins, with the only (obvious) requirement that the items are packed without overlapping.

The problem is the two-dimensional extension of the classic (One-Dimensional) Bin Packing Problem and is one of the most studied problem in the so called *Cutting & Packing* category (see Dyckhoff, Scheithauer, and Terno [64] for an annotated bibliography on the subject).

The great interest of 2BP is mainly due to the number of real-world applications in which it arises, and depending on these applications, 2BP can be found in the literature with the addition of different practical requirements which originate many interesting variants.

In Chapter 2 the state of the art on 2BP is presented, together with a discussion on its main variants and extensions. The problem has been classically addressed through heuristic techniques, whereas in the last five years both exact and metaheuristic (more precisely Tabu Search) approaches have been proposed. In Chapter 2 the classical results and the recent approaches are reviewed, whereas in Chapters 3, 4 and 5 the Tabu Search scheme is presented in detail. In particular, Chapter 3 contains the original Tabu Search algorithm whose effectiveness is proved with computational experiments over the benchmark instances in the literature. In Chapter 4 a slight extension of the original approach is proved to be effective for the 2BP variant in which the items are allowed to rotate by 90°. Finally in Chapter 5 the evolution of the original Tabu Search is presented as a unified framework. This framework can be applied without modifications to all variants of two-dimensional bin packing, and its effectiveness is tested on the class of problems originated by two of the most common requirements: (i) the (already mentioned) orientation of the items and (ii) the guillotine cutting (items must be obtained through a sequence of edge-to-edge cuts parallel to the edges of the bin). These requirements generate the following class of 2BP problems:

2BP|O|G: the items are oriented (O), and guillotine cutting (G) is required;

2BP|R|G: the items may be rotated by 90° (R) and guillotine cutting is required;

2BP|O|F: the items are oriented and cutting is free (F);

2BP|R|F: the items may be rotated by 90° and cutting is free.

The above chapters are presented in this thesis as self-contained entities because, during the PhD years, some steps have been "fixed" into almost separate papers. The advantage here is to find the overall research unified, with a logical view of the complete project and of its evolution.

The last chapter of Part I is devoted to recent advances and future research in the Cutting & Packing area. In particular, Chapter 6 addresses three different topics: in Section 6.1 the Tabu Search method discussed in Chapters 3, 4 and 5 is applied to the *Three-Dimensional Bin Packing Problem*, 3BP for short, showing once more its generality and effectiveness. In Section 6.2 a new lower bound for 2BP (and its extension to 3BP) is presented, whereas in Section 6.3 we consider some Cutting & Packing problems with the additional constraint that the items must be packed into shelves, i.e., rows forming levels. Mathematical models for these *Shelf Packing* problems are presented, and computationally tested.

Chapter 2

The Two-Dimensional Bin Packing Problem

2.1 Introduction

In[1] the (finite) *two-dimensional bin packing problem* (2BP) we are given a set of n rectangular *items* $j \in J = \{1, \ldots, n\}$, each having *width* w_j and *height* h_j, and an unlimited number of finite identical rectangular *bins*, having width W and height H. The problem is to allocate, without overlapping, all the items to the minimum number of bins, with their edges parallel to those of the bins. It is assumed that the items have fixed orientation, i.e., they cannot be rotated.

The problem has many industrial applications, especially in cutting (wood and glass industries) and packing (transportation and warehousing). Certain applications may require additional constraints and/or assumptions, some of which are discussed in the final section of this chapter.

The special case where $w_j = W$ $(j = 1, \ldots, n)$ is the famous *one-dimensional bin packing problem* (1BP): partition n elements, each having an associated size h_j, into the minimum number of subsets so that the sum of the sizes in each subset does not exceed a given capacity H. Since 1BP is known to be strongly NP-hard, the same holds for 2BP.

In this chapter we survey recent advances obtained for the two-dimensional bin packing problem. Up to the mid-Nineties, almost all results in the literature concerned heuristic algorithms. These are reviewed in the next section, together with more recent results. The next sections are devoted to other results obtained in the last few years: lower bounds (Section 2.3), exact algorithms (Section 2.4) and metaheuristic approaches (Section 2.5). We conclude by discussing relevant variants of the problem (Section 2.6). For many of the above techniques we summarize the results of computational testings, illustrating their average effectiveness. For some upper and lower bounds, worst-case results are also discussed.

Without loss of generality, we will assume throughout the chapter that all input data are positive integers, and that $w_j \leq W$ and $h_j \leq H$ $(j = 1, \ldots, n)$.

[1] The results of this chapter appear in: A. Lodi, S. Martello, D. Vigo, "Recent Advances on Two-Dimensional Bin Packing Problems", *Technical Report* OR/99/2, DEIS - Università di Bologna,[110].

2.2 Upper Bounds

Most of the algorithms from the literature are of greedy type, and can be classified in two families:

- *one-phase algorithms* directly pack the items into the finite bins;
- *two-phase algorithms* start by packing the items into a single *strip*, i.e., a bin having width W and infinite height. In the second phase, the strip solution is used to construct a packing into finite bins.

In addition, most of the approaches are *shelf algorithms*, i.e., the bin/strip packing is obtained by placing the items, from left to right, in rows forming levels (*shelves*). The first shelf is the bottom of the bin/strip, and subsequent shelves are produced by the horizontal line coinciding with the top of the tallest item packed on the shelf below. Three classical strategies for the shelf packing have been derived from famous algorithms for the one-dimensional case. In each case, the items are initially sorted by nondecreasing height and packed in the corresponding sequence. Let j denote the current item, and s the last created shelf:

- *Next-Fit Decreasing Height* (NFDH) strategy: item j is packed left justified on shelf s, if it fits. Otherwise, a new shelf ($s := s+1$) is created, and j is packed left justified into it;
- *First-Fit Decreasing Height* (FFDH) strategy: item j is packed left justified on the first shelf where it fits, if any. If no shelf can accommodate j, a new shelf is initialized as in NFDH;
- *Best-Fit Decreasing Height* (BFDH) strategy: item j is packed left justified on that shelf, among those where it fits, for which the unused horizontal space is a minimum. If no shelf can accommodate j, a new shelf is initialized as in NFDH.

The above strategies are illustrated through an example in Figure 2.1.

In what follows we assume, unless otherwise specified, that the items are initially sorted by nonincreasing height.

2.2.1 Strip packing

Coffman, Garey, Johnson and Tarjan [43] analyzed NFDH and FFDH for the solution of the *two-dimensional strip packing problem*, in which one is required to pack all the items into a strip of minimum height, and determined their asymptotic worst-case behavior. Given a minimization problem P and an approximation algorithm A, let $A(I)$ and $OPT(I)$ denote the value produced by A and the optimal solution value, respectively, for an instance I of P. Coffman, Garey, Johnson and Tarjan [43] proved that, if the heights are normalized so that $\max_j\{h_j\} = 1$,

$$NFDH(I) \leq 2 \cdot OPT(I) + 1 \tag{2.1}$$

and

$$FFDH(I) \leq \frac{17}{10} \cdot OPT(I) + 1 \tag{2.2}$$

Figure 2.1: Shelf packing strategies.

Both bounds are tight and, if the h_j's are not normalized, only the additive term is affected. Observe the similarity of (2.1) and (2.2) with famous results on the one-dimensional counterparts of NFDH and FFDH (algorithms *Next-Fit* and *First-Fit*, respectively, see Johnson, Demers, Ullman, Garey and Graham [98]).

Both algorithms can be implemented so as to require $O(n \log n)$ time, by using the appropriate data structures adopted for the one-dimensional case (see Johnson [97]).

Several other papers on the strip packing problem can be found in the literature: see, e.g., Baker, Coffman and Rivest [7], Sleator [145], Brown [27], Golan [83], Baker, Brown and Katseff [6], Baker and Schwarz [8], Høyland [95], Steinberg [146]. The algorithm of Baker, Coffman and Rivest [7] is considered in Section 2.2.4, while the other results, which have not been directly used for the finite bin case, are beyond the scope of this survey, and will not be discussed here.

2.2.2 Bin packing: two-phase algorithms

A two-phase algorithm for the finite bin packing problem, called *Hybrid First-Fit* (HFF), was proposed by Chung, Garey and Johnson [42]. In the first phase, a strip packing is obtained through the FFDH strategy. Let $H_1, H_2, \ldots$ be the heights of the resulting shelves, and observe that $H_1 \geq H_2 \geq \ldots$. A finite bin packing solution is then obtained by heuristically solving a one-dimensional bin packing problem (with item sizes H_i and bin capacity H) through the *First-Fit Decreasing* algorithm: initialize bin 1 to pack shelf 1, and, for increasing $i = 2, \ldots$, pack the current shelf i into the lowest indexed bin where it fits, if any; if no bin can accommodate i, initialize a new bin. An example is shown in Figure 2.2. Chung, Garey and Johnson [42] proved that, if the heights are normalized to one,

$$HFF(I) \leq \frac{17}{8} \cdot OPT(I) + 5 \tag{2.3}$$

The bound is not proved to be tight: the worst example gives $HFF(I) = \frac{91}{45} \cdot (OPT(I) - 1)$. Both phases can be implemented so as to require $O(n \log n)$ time.

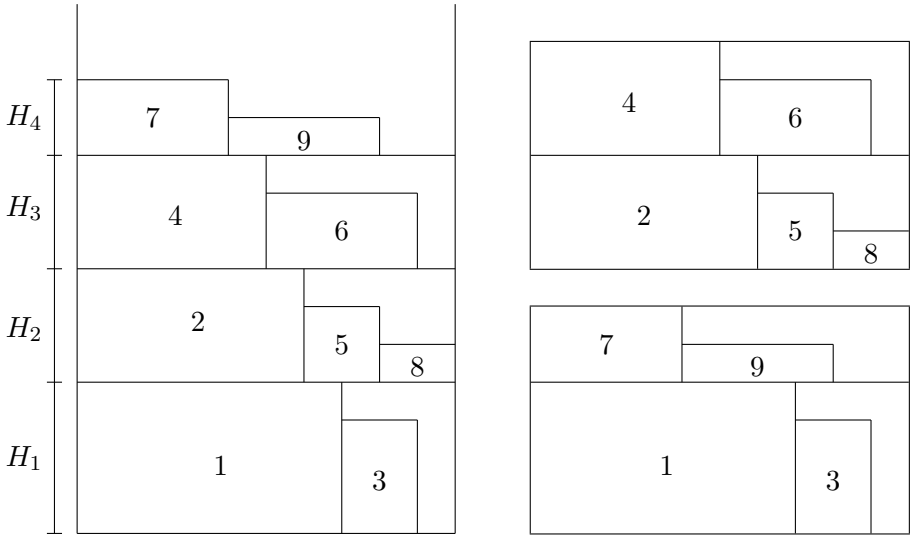


Figure 2.2: First and second phase of algorithm HFF.

Berkey and Wang [23] proposed and experimentally evaluated a two-phase algorithm, called *Finite Best-Strip* (FBS), which is a variation of HFF. The first phase is performed by using the BFDH strategy. In the second phase, the one-dimensional bin packing problem is solved through the *Best-Fit Decreasing* algorithm: pack the current shelf in that bin, among those where it fits (if any), for which the unused vertical space is a minimum, or by initializing a new bin. (For the sake of uniformity, *Hybrid Best-Fit* would be a more appropriate name for this algorithm.)

Let us consider now another variation of HFF, in which the NFDH strategy is adopted in the first phase, and the one-dimensional bin packing problem is solved through the *Next-Fit Decreasing* algorithm: pack the current shelf in the current bin if it fits, or initialize a new (current) bin otherwise. Due to the next-fit policy, this algorithm is equivalent to a one-phase algorithm in which the current item is packed on the current shelf of the current bin, if possible; otherwise, a new (current) shelf is initialized either in the current bin (if enough vertical space is available), or in a new (current) bin. Frenk and Galambos [76] analyzed the resulting algorithm, *Hybrid Next-Fit* (HNF), by characterizing its asymptotic worst-case performance as a function of $\max_j\{w_j\}$ and $\max_j\{h_j\}$. By assuming that the heights and widths are normalized to one, the worst performance occurs for $\max_j\{w_j\} > \frac{1}{2}$ and $\max_j\{h_j\} > \frac{1}{2}$, and gives:

$$HNF(I) \leq 3.382 \cdot OPT(I) + 9 \tag{2.4}$$

The bound is tight.

The three algorithms above can be implemented so as to require $O(n \log n)$ time. The next two algorithms have higher worst-case time complexities, although they are, on average, very fast in practice and more effective.

Lodi, Martello and Vigo [107, 109] presented an approach (*Floor-Ceiling*, FC) which extends the way items are packed on the shelves. Denote the horizontal line defined by the top (resp. bottom) edge of the tallest item packed on a shelf as the *ceiling* (resp. *floor*) of the

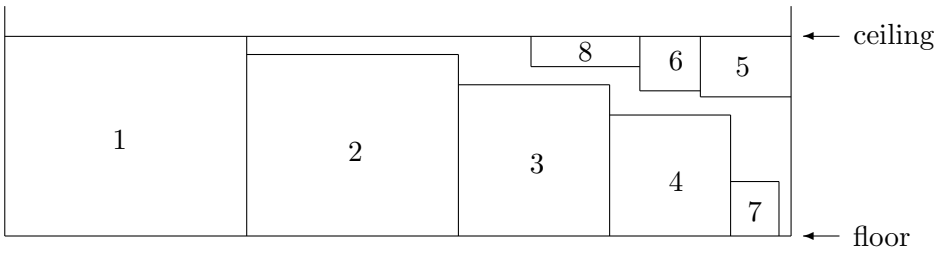


Figure 2.3: Algorithm FC.

shelf. The previous algorithms pack the items, from left to right, with their bottom edge on the shelf floor. Algorithm FC may, in addition, pack them, from right to left, with their top edge on the shelf ceiling. The first item packed on a ceiling can only be one which cannot be packed on the floor below. A possible floor-ceiling packing is shown in Figure 2.3. In the first phase, the current item is packed, in order of preference: (i) on a ceiling (provided that the requirement above is satisfied), according to a best-fit strategy; (ii) on a floor, according to a best-fit strategy; (iii) on the floor of a new shelf. In the second phase, the shelves are packed into finite bins, either through the Best-Fit Decreasing algorithm or by using an exact algorithm for the one-dimensional bin packing problem, halted after a prefixed number of iterations. The first phase can be implemented so as to require $O(n^3)$ time, while the complexity of the second one obviously depends on the selected algorithm.

Another shelf packing strategy based on the exact solution of induced subproblems is adopted in the *Knapsack Packing* (KP) algorithm proposed by Lodi, Martello and Vigo [109]. In the (binary) *knapsack problem* one has to select a subset of n elements, each having an associated profit and weight, so that the total weight does not exceed a given capacity and the total profit is a maximum. The first phase of algorithm KP packs one shelf at a time as follows. The first (tallest) unpacked item, say j^*, initializes the shelf, which is then completed by solving an associated knapsack problem instance over all the unpacked items, where: (i) the knapsack capacity is $W - w_{j^*}$; (ii) the weight of an item j is w_j; (iii) the profit of an item j is its area $w_j\ h_j$. Finite bins are finally obtained as in algorithm FC. Algorithms KP (as well as algorithm FC above) may require the solution of NP-hard subproblems, producing a non-polynomial time complexity. In practice, however, the execution of the codes for the NP-hard problems is always halted after a prefixed (small) number of iterations, and in almost all cases, the optimal solution is obtained before the limit is reached (see the computational experiments in [109] and in Section 2.2.5).

2.2.3 Bin packing: one-phase algorithms

Two one-phase algorithms were presented and experimentally evaluated by Berkey and Wang [23].

Algorithm *Finite Next-Fit* (FNF) directly packs the items into finite bins exactly in the way algorithm HNF of the previous section does. (Papers [23] and [76] appeared in the same year.)

Algorithm *Finite First-Fit* (FFF) adopts instead the FFDH strategy. The current item is packed on the lowest shelf of the first bin where it fits; if no shelf can accommodate it, a new shelf is created either in the first suitable bin, or by initializing a new bin (if no bin has

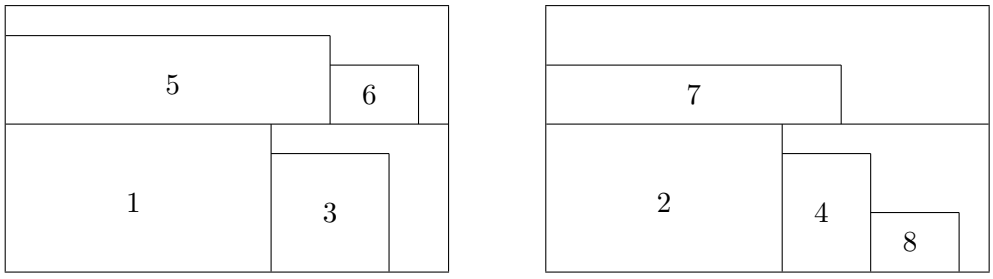


Figure 2.4: Algorithm FFF.

enough vertical space available). An example of application of FFF is given in Figure 2.4.

Both algorithms can be implemented so as to require $O(n \log n)$ time.

2.2.4 Bin packing: non-shelf algorithms

We finally consider algorithms which do not pack the items by shelves. All the algorithms discussed in the following are one-phase.

The main non-shelf strategy is known as *Bottom-Left* (BL), and consists in packing the current item in the lowest possible position, left justified. Baker, Coffman and Rivest [7] analyzed the worst-case performance of the resulting algorithm for the strip packing problem, and proved that: (i) if no item ordering is used, BL may be arbitrarily bad; (ii) if the items are ordered by nonincreasing width then $BL(I) \leq 3 \cdot OPT(I)$, and the bound is tight.

Berkey and Wang [23] proposed the BL approach for the finite bin case. Their *Finite Bottom-Left* (FBL) algorithm initially sorts the items by nonincreasing width. The current item is then packed in the lowest position of any initialized bin, left justified; if no bin can allocate it, a new one is initialized. The computer implementation of algorithm BL was studied by Chazelle [40], who gave a method for producing a packing in $O(n^2)$ time. The same approach was adopted by Berkey and Wang [23].

Lodi, Martello and Vigo [109] proposed a different non-shelf approach, called *Alternate Directions* (AD). The method is illustrated in Figure 2.5. The algorithm initializes L bins (L being a lower bound on the optimal solution value, see Section 2.3) by packing on their bottoms a subset of the items, following a best-fit decreasing policy (items 1, 2, 3, 7 and 9 in Figure 2.5, where it is assumed that $L = 2$). The remaining items are packed, one bin at a time, into *bands*, alternatively from left to right and from right to left. As soon as no item can be packed in either direction in the current bin, the next initialized bin or a new empty bin (the third one in Figure 2.5, when item 11 is considered) becomes the current one. The algorithm has $O(n^3)$ time complexity.

2.2.5 Computational experiments

In this section we summarize the outcome of a series of computational experiments aimed at analyzing the average behavior of the main heuristic algorithms. The benchmark consists of 500 random instances, with $n \in \{20, 40, 60, 80, 100\}$. Ten different classes of instances were used.

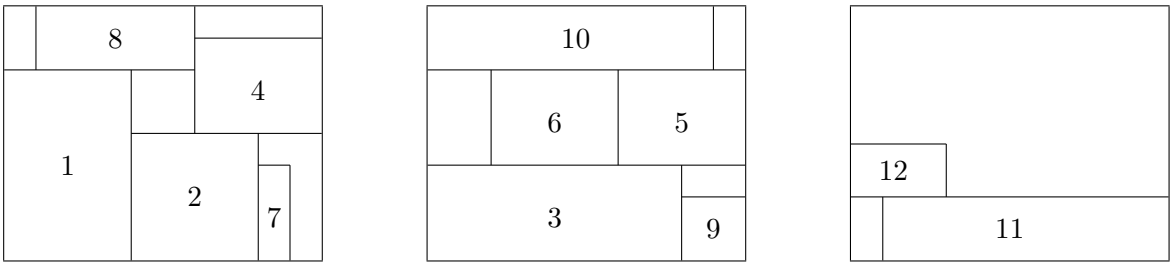


Figure 2.5: Algorithm AD.

The first four classes were proposed by Martello and Vigo [119], and are based on the generation of items of four different types:

type 1 : w_j uniformly random in $[\frac{2}{3}W, W]$, h_j uniformly random in $[1, \frac{1}{2}H]$;

type 2 : w_j uniformly random in $[1, \frac{1}{2}W]$, h_j uniformly random in $[\frac{2}{3}H, H]$;

type 3 : w_j uniformly random in $[\frac{1}{2}W, W]$, h_j uniformly random in $[\frac{1}{2}H, H]$;

type 4 : w_j uniformly random in $[1, \frac{1}{2}W]$, h_j uniformly random in $[1, \frac{1}{2}H]$.

Class k ($k \in \{1, 2, 3, 4\}$) is then obtained by generating an item of type k with probability 70%, and of the remaining types with probability 10% each. The bin size is always $W = H = 100$.

The next six classes have been proposed by Berkey and Wang [23]:

Class 5 : $W = H = 10$, w_j and h_j uniformly random in $[1, 10]$;

Class 6 : $W = H = 30$, w_j and h_j uniformly random in $[1, 10]$;

Class 7 : $W = H = 40$, w_j and h_j uniformly random in $[1, 35]$;

Class 8 : $W = H = 100$, w_j and h_j uniformly random in $[1, 35]$;

Class 9 : $W = H = 100$, w_j and h_j uniformly random in $[1, 100]$;

Class 10 : $W = H = 300$, w_j and h_j uniformly random in $[1, 100]$.

For each class and value of n, ten instances have been generated. Figure 2.6 summarizes the results, by giving, for each algorithm, the average percentage deviation of the heuristic solution value from the best known lower bound value, with respect to the 200 instances of Classes 1–4 (Figure 2.6 (a)) and to the 300 instances of Classes 5–10 (Figure 2.6 (b)).

The 500 instances, as well as the generator code, are available on the internet at the WWW address `http://www.or.deis.unibo.it/ORinstances/2BP/`.

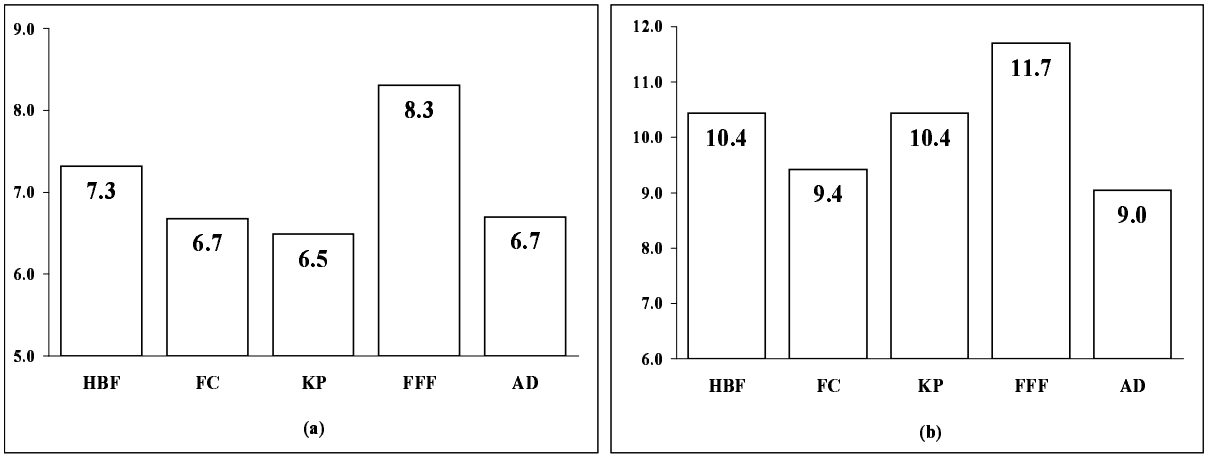


Figure 2.6: Average percentage deviations from lower bound: (a) Martello and Vigo instances, Classes 1–4; (b) Berkey and Wang instances, Classes 5–10.

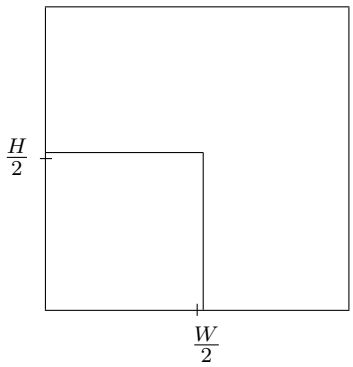


Figure 2.7: Worst-case of the continuous lower bound.

2.3 Lower Bounds

The simplest bound for 2BP is the *Continuous Lower Bound*

$$L_0 = \left\lceil \frac{\sum_{j=1}^{n} w_j h_j}{WH} \right\rceil$$

computable in linear time. Martello and Vigo [119] determined the absolute worst-case behavior of L_0:

$$L_0(I) \geq \frac{1}{4} \cdot OPT(I)$$

where $L_0(I)$ and $OPT(I)$ denote the value produced by L_0 and the optimal solution value, respectively, for an instance I of P. The bound is tight, as shown by the example in Figure 2.7. The result holds even if rotation of the items (by any angle) is allowed.

A better bound was proposed by Martello and Vigo [119]. Given any integer value q, $1 \leq q \leq \frac{1}{2}W$, let

$$K_1 = \{j \in J : w_j > W - q\} \tag{2.5}$$

$$K_2 = \{j \in J : W - q \geq w_j > \frac{1}{2}W\} \tag{2.6}$$

$$K_3 = \{j \in J : \frac{1}{2}W \geq w_j \geq q\} \tag{2.7}$$

and observe that no two items of $K_1 \cup K_2$ may be packed side by side into a bin. Hence, a lower bound L_1^W for the sub-instance given by the items in $K_1 \cup K_2$ can be obtained by using any lower bound for the 1BP instance defined by element sizes w_j ($j \in K_1 \cup K_2$) and capacity W (see Martello and Toth [118], Dell'Amico and Martello [57]). A lower bound for the complete instance is then obtained by taking into account the items in K_3, since none of them may be packed besides an item of K_1:

$$L_2^W(q) = L_1^W + \max\left\{0, \left\lceil \frac{\sum_{j\in K_2\cup K_3} w_j h_j - (HL_1^W - \sum_{j\in K_1} h_j)W}{WH} \right\rceil\right\} \tag{2.8}$$

A symmetric bound $L_2^H(q)$ is clearly obtained by interchanging widths and heights. By observing that both bounds are valid for any q, we have an overall lower bound:

$$L_2 = \max\left(\max_{1\leq q\leq \frac{1}{2}W}\{L_2^W(q)\}\ ,\quad \max_{1\leq q\leq \frac{1}{2}H}\{L_2^H(q)\}\right) \tag{2.9}$$

It is shown in [119] that L_2 dominates L_0, and can be computed in $O(n^2)$ time.

Martello and Vigo [119] also proposed a computationally more expensive lower bound, which in some cases improves on L_2. Given any pair of integers (p, q), with $1 \leq p \leq \frac{1}{2}H$ and $1 \leq q \leq \frac{1}{2}W$, define:

$$I_1 = \{j \in J : h_j > H - p \text{ and } w_j > W - q\} \tag{2.10}$$

$$I_2 = \{j \in J \setminus I_1 : h_j > \tfrac{1}{2}H \text{ and } w_j > \tfrac{1}{2}W\} \tag{2.11}$$

$$I_3 = \{j \in J : \tfrac{1}{2}H \geq h_j \geq p \text{ and } \tfrac{1}{2}W \geq w_j \geq q\} \tag{2.12}$$

(see Figure 2.8 (a)), and observe that: (i) $I_1 \cup I_2$ is independent of (p, q); (ii) no two items of $I_1 \cup I_2$ may be packed into the same bin; (iii) no item of I_3 fits into a bin containing an item of I_1. A valid lower bound can thus be computed by adding to $|I_1 \cup I_2|$ the minimum number of bins needed for those items of I_3 that cannot be packed into the bins used for the items of I_2. Such a bound can be determined by considering a relaxed instance where each item $i \in I_3$ has the minimum size, i.e., $h_i = p$ and $w_i = q$. Given a bin containing an item j, the maximum number of $p \times q$ items that can be packed into the bin is (see Figure 2.8 (b)):

$$m(j, p, q) = \left\lfloor \frac{H}{p} \right\rfloor \left\lfloor \frac{W - w_j}{q} \right\rfloor + \left\lfloor \frac{W}{q} \right\rfloor \left\lfloor \frac{H - h_j}{p} \right\rfloor - \left\lfloor \frac{H - h_j}{p} \right\rfloor \left\lfloor \frac{W - w_j}{q} \right\rfloor \tag{2.13}$$

Hence, for any pair (p, q) a valid lower bound is

$$L_3(p, q) = |I_1 \cup I_2| + \max\left\{0, \left\lceil \frac{|I_3| - \sum_{j\in I_2} m(j, p, q)}{\left\lfloor \frac{H}{p} \right\rfloor \left\lfloor \frac{W}{q} \right\rfloor} \right\rceil\right\} \tag{2.14}$$

so an overall bound is

$$L_3 = \max_{1\leq p\leq \frac{1}{2}H,\ \ 1\leq q\leq \frac{1}{2}W}\{L_3(p, q)\} \tag{2.15}$$

Lower bound L_3 can be computed in $O(n^3)$ time. No dominance relation exists between L_2 and L_3.

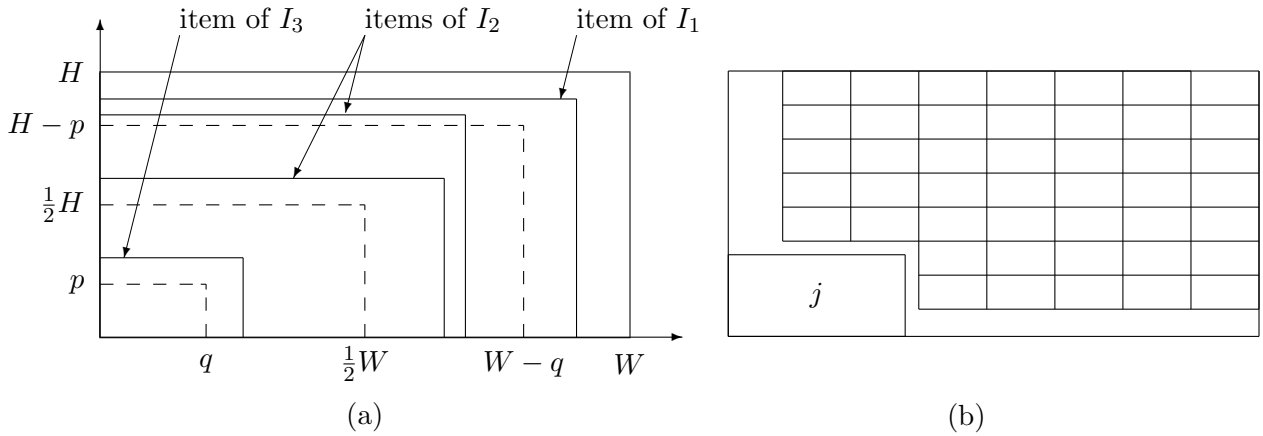


Figure 2.8: Lower bound L_3: (a) items in I_1, I_2 and I_3; (b) relaxed instance with reduced items.

2.4 Exact Algorithms

An enumerative approach for the exact solution of 2BP was presented by Martello and Vigo [119]. The items are initially sorted in nonincreasing order of their area. A reduction procedure tries to determine the optimal packing of some bins, thus reducing the size of the instance. A first incumbent solution, of value z^*, is then heuristically obtained.

The algorithm is based on a two-level branching scheme:

- *outer branch-decision tree*: at each decision node, an item is assigned to a bin without specifying its actual position;
- *inner branch-decision tree*: a feasible packing (if any) for the items currently assigned to a bin is determined, possibly through enumeration of all the possible patterns.

The outer branch-decision tree is searched in a depth-first way, making use of the lower bounds described in the previous section. Whenever it is possible to establish that no more unassigned items can be assigned to a given initialized bin, such a bin is *closed*: an initialized and not closed bin is called *active*. At level k ($k = 1, \ldots, n$), item k is assigned, in turn, to all the active bins and, possibly, to a new one (if the total number of active and closed bins is less than $z^* - 1$).

The feasibility of the assignment of an item to a bin is first heuristically checked. A lower bound $L(I)$ is computed for the instance I defined by the items currently assigned to the bin: if $L(I) > 1$, a backtracking follows. Otherwise, heuristic algorithms are applied to I: if a feasible single-bin packing is found, the outer enumeration is resumed. If not, the inner branching scheme enumerates all the possible ways to pack I into a bin through the *left-most downward* strategy (see Hadjiconstantinou and Christofides [88]): at each level, the next item is placed, in turn, into all positions where it has its left edge adjacent either to the right edge of another item or to the left edge of the bin, and its bottom edge adjacent either to the top edge of another item or to the bottom edge of the bin. As soon as a feasible packing is found for all the items of I, the outer enumeration is resumed. If no such packing exists, an outer backtracking is performed.

Whenever the current assignment is feasible, the possibility of closing the bin is checked through lower bound computations.

Table 2.1 gives the results of computational experiments performed, with a Fortran 77 implementation, on the 500 instances described in Section 2.2.5. The entries give, for each class and value of n, the number of instances (out of ten) solved to proved optimality within a time limit of 300 CPU seconds on a PC Pentium 200 MHz.

Table 2.1: Number of instances, out of ten, solved to proved optimality.

	Class										
n	1	2	3	4	5	6	7	8	9	10	Total
20	10	10	10	10	10	10	9	10	10	10	99
40	8	7	10	9	10	10	10	10	10	5	89
60	8	7	10	2	7	4	7	7	8	10	70
80	7	3	10	–	3	10	–	10	–	10	53
100	7	6	8	–	1	10	–	10	1	2	45
Total	40	33	48	21	31	44	26	47	29	37	356

2.5 Metaheuristics

In recent years, metaheuristic techniques have become a popular tool for the approximate solution of hard combinatorial optimization problems. (See Aarts and Lenstra [2] and Glover and Laguna [82] for general introductions to the field.) Lodi, Martello and Vigo [107, 108, 109] developed effective tabu search algorithms for 2BP and for some of the variants discussed in the next section. We briefly describe here the unified tabu search framework given in [109], whose main characteristic is the adoption of a search scheme and a neighborhood which are independent of the specific packing problem to be solved. The framework can thus be used for virtually any variant of 2BP, by simply changing the specific deterministic algorithm used for evaluating the moves within the neighborhood search.

Given a current solution, the moves modify it by changing the packing of a subset S of items, trying to empty a specified *target bin*. Let S_i be the set of items currently packed into bin i: the target bin t is the one minimizing, over all bins i, the function

$$\varphi(S_i) = \alpha \frac{\sum_{j \in S_i} w_j h_j}{WH} - \frac{|S_i|}{n} \tag{2.16}$$

(α is a pre-specified positive weight), which gives a measure of the easiness of emptying the bin. It favors, in fact, target bins packing a small area and a relatively large number of items.

Once the target bin has been selected, subset S is defined so as to include one item, j, from the target bin and the current contents of k other bins. The new packing for S is obtained by executing an appropriate heuristic algorithm A on S. The value of parameter k, which defines the size and the structure of the current neighborhood, is automatically updated during the search.

If the move packs the items of S into k (or less) bins, i.e., item j has been removed from the target bin, a new item is selected, a new set S is defined accordingly, and a new move is performed. Otherwise S is changed by selecting a different set of k bins, or a different item

j from the target bin (if all possible configurations of k bins have been attempted for the current j).

If the algorithm gets stuck, i.e., the target bin is not emptied, the neighborhood is enlarged by increasing the value of k, up to a prefixed upper limit. There are a tabu list and a tabu tenure for each value of k.

An initial incumbent solution is obtained by executing algorithm A on the complete instance, while the initial tabu search solution consists of packing one item per bin. In special situations, a move is followed by a diversification action. The execution is halted as soon as a proven optimal solution is found, or a time limit is reached.

Figure 2.9 shows the impact of tabu search for three of the heuristics described in Section 2.2: notation TS(A) indicates that algorithm A is used within the tabu search. The figure gives the average percentage deviations of the heuristic solution value (without and with tabu search) from the best known lower bound value, with respect to the 200 instances of Classes 1–4 (Figure 2.9 (a)) and to the 300 instances of Classes 5–10 (Figure 2.9 (b)), as described in Section 2.2.5. The tabu search was performed with a time limit of 60 seconds on a Silicon Graphics INDY R10000sc (195 MHz).

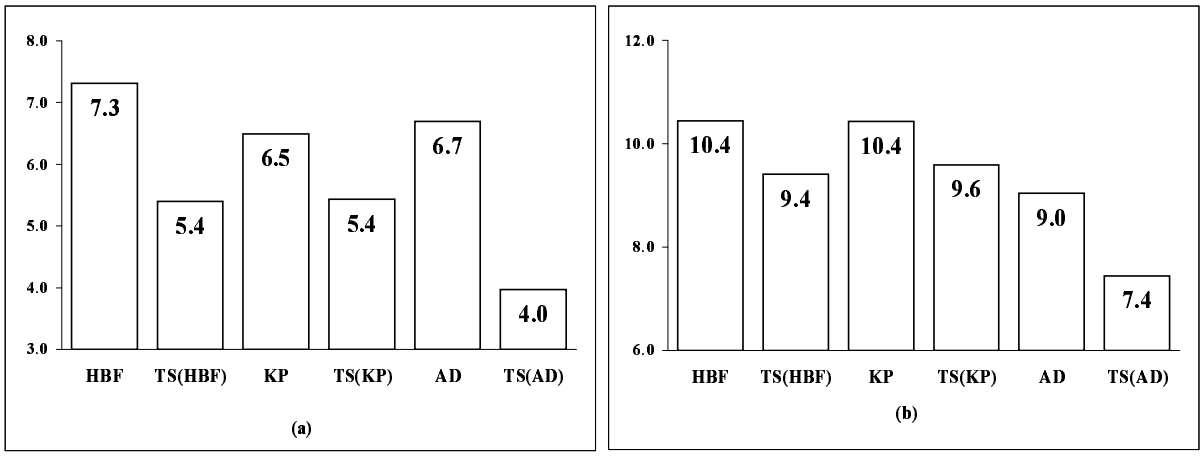


Figure 2.9: Average percentage deviations from lower bound: (a) Martello and Vigo instances, Classes 1–4; (b) Berkey and Wang instances, Classes 5–10.

2.6 Variants and Extensions

Two-dimensional bin packing problems occur in several real-world contexts, especially in cutting and packing industries. As a consequence, a number of variants arises, according to specific applications. In most cases the additional requirements concern *orientation* (the items may either have a fixed orientation or they can be rotated by 90°), and/or *guillotine cutting* (it may or not be imposed that the items are obtained through a sequence of edge-to-edge cuts parallel to the edges of the bin). For example, rotation is not allowed when the items are articles to be paged in newspapers or are pieces to be cut from decorated or corrugated stock units, whereas it is allowed in the cutting of plain materials and in most packing contexts. The guillotine constraint is usually imposed by technological characteristics of the automated

cutting machines, whereas it is generally not present in packing applications.

Lodi, Martello and Vigo [109] proposed the following typology for the four possible cases produced by the above two requirements:

2BP|O|G: the items are oriented (O) and guillotine cutting (G) is required;

2BP|R|G: the items may be rotated by 90° (R) and guillotine cutting is required;

2BP|O|F: the items are oriented and cutting is free (F);

2BP|R|F: the items may be rotated by 90° and cutting is free.

(The problem considered so far is thus 2BP|O|F.) The following references are examples of industrial applications involving the above variants. A problem of trim-loss minimization in a crepe-rubber mill, studied by Schneider [143], induces subproblems of 2BP|O|G type; fuzzy two-dimensional cutting stock problems arising in the steel industry, discussed by Vasko, Wolf and Stott [148], are related to 2BP|R|G; the problem of optimally placing articles and advertisements in newspapers and yellow pages, studied by Lagus, Karanta and Ylä-Jääski [102], falls into the 2BP|O|F case; finally, several applications of 2BP|R|F are considered by Bengtsson [21].

An algorithm for one of the variants may obviously guarantee solutions which are feasible for others. The complete set of compatibilities between algorithms and problems is shown in Figure 2.10, where A_{XY} is an algorithm for 2BP$|X|Y$ and an edge (A_{XY},2BP$|Q|T$) indicates that A_{XY} produces solutions feasible for 2BP$|Q|T$. Most of the heuristics in Section 2.2 can be modified so as to handle rotation and/or guillotine cutting, as discussed in Lodi, Martello and Vigo [109]. It is also worth mentioning that all the shelf algorithms described in Section 2.2 directly produce guillotine packings, the only exception being FC. A variant of FC which slightly modifies the way items are packed on the ceilings so as to preserve the guillotine constraint, was presented by Lodi, Martello and Vigo [107].

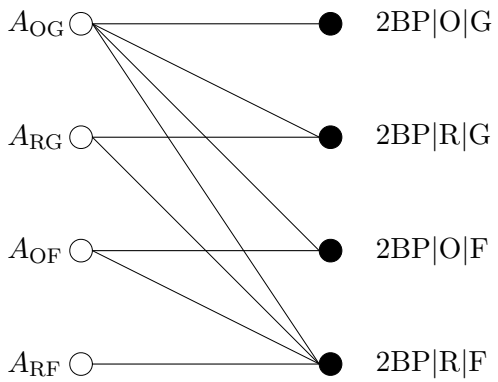


Figure 2.10: Compatibilities between algorithms and problems.

We finally mention that other variants of the two-dimensional bin packing problem can be found in the literature. For example, in guillotine cutting, an upper bound (usually two or three) may be imposed on the number of *stages* (rounds of cuts having the same direction) that are needed to obtain all the items: see, e.g., Hifi [94]. In certain practical applications

a secondary objective can also be of interest, namely the maximization of the unused area in one bin, so as to produce a possibly large trim to be used later: see, e.g., Bengtsson [21] and El-Bouri, Popplewell, Balakrishnan and Alfa [65] for 2BP|R|F.

2.6.1 The Three-Dimensional Bin Packing Problem

In the extension to the three-dimensional case, each item is a rectangular box having width w_j, height h_j and *depth* d_j, and the bins have width W, height H and depth D. The *Three-Dimensional Bin Packing Problem* (3BP) is to allocate, without overlapping and without rotation, all the items to the minimum number of bins, with their faces parallel to those of the bins.

Martello, Pisinger and Vigo [116] proved that, for 3BP, the continuous lower bound,

$$\left\lceil \frac{\sum_{j=1}^{n} w_j h_j d_j}{WHD} \right\rceil$$

has an asymptotic worst-case performance ratio equal to $\frac{1}{8}$. The result was generalized by Dell'Amico [47], who showed that, for the k-dimensional bin packing problem, the continuous lower bound has an asymptotic worst-case performance ratio equal to $\frac{1}{2^k}$.

Other lower bounds, partially based on extensions of those discussed in Section 2.3, as well as an exact branch-and-bound approach, were also proposed in [116]. Table 2.2 gives the results of computational experiments performed, with an ANSI-C language implementation of the branch-and-bound algorithm, on nine classes of random instances: Classes 1–5 and 6–8 were obtained by generalizing the 2BP classes 1–4 and 5–10, respectively (see Section 2.2.5), while Class 9 contains very difficult special instances. The entries give, for each class and value of n, the number of instances (out of ten) solved to proved optimality within a time limit of 1000 CPU seconds on a HP9000/C160 (160 MHz).

Table 2.2: 3BP: number of instances, out of ten, solved to proved optimality.

	Class									
n	1	2	3	4	5	6	7	8	9	Total
10	10	10	10	10	10	10	10	10	10	90
15	10	10	10	10	10	10	10	10	10	90
20	10	10	10	10	10	10	10	10	10	90
25	10	10	10	10	10	10	9	10	10	89
30	10	10	10	10	10	10	5	10	10	85
35	9	9	10	10	7	10	5	6	9	75
40	9	8	8	10	4	10	3	8	8	68
45	5	6	4	10	8	10	1	7	2	53
50	5	2	4	10	3	10	2	4	—	40
60	1	1	—	9	1	6	—	5	—	23
70	—	—	—	6	2	4	—	4	—	16
80	—	—	1	4	—	4	—	2	—	11
90	—	—	—	2	—	7	—	1	—	10
Total	79	76	77	111	75	111	55	87	69	740

In the *Three-Dimensional Strip Packing Problem* there is single bin having fixed width and depth, and infinite height, and one is required to pack all the items within minimum height. Worst-case results of approximation algorithms for this problem were presented by Li and Cheng [104, 105] and by Miyazawa and Wakabayashi [121].

Chapter 3

Two-Dimensional Bin Packing: the 2BP|O|G case

3.1 Introduction

Given[1] n rectangular *items*, each characterized by a height h_j and a width w_j ($j = 1, \ldots, n$), and an unlimited number of identical rectangular *bins*, each having height H and width W, the *Oriented Two-Dimensional Bin Packing Problem* (2BP) is to pack each item into a bin so that no two items overlap and the number of required bins is minimized. We assume that the items cannot be rotated. We also assume, without loss of generality, that all input data are positive integers and that $h_j \leq H$, $w_j \leq W$ ($j = 1, \ldots n$). According to the typology given by Dykhoff [62], the problem can be classified as 2/V/I/M.

The problem is known to be strongly NP-hard and finds many practical applications. For example, in the glass industry and in the wood industry it is requested to cut rectangular items in demand from standardized stock sheets by minimizing the trim loss, i.e., by using the minimum number of stock sheets. According to the specific application, the items in demand may have a prefixed orientation (e.g., in the cutting of corrugated iron) or may be rotated (usually by 90°). As previously stated, this chapter deals with the *oriented case* in which no item rotation is allowed. In cutting applications it is frequently required that the resulting patterns are *guillotine-cuttable*, i.e., that the items can be obtained through a sequence of edge-to-edge cuts parallel to the edges of the bin: as will be pointed out later, the algorithms we propose satisfy such constraint.

Approximation algorithms for 2BP have been given by Chung, Garey and Johnson [42], Berkey and Wang [23] and Frenk and Galambos [76], whereas Bengtsson [21] and El-Bouri et al. [65] considered the case with 90° rotation of the items. An exact branch-and-bound approach for 2BP has been developed by Martello and Vigo [119]. Surveys on packing problems can be found in Dyckhoff and Finke [63] and Dowsland and Dowsland [60], while an annotated bibliography has recently been presented by Dyckhoff, Scheithauer and Terno [64].

This chapter presents a tabu search approach to 2BP. The algorithm is initialized with the solution obtained by using a simple and fast heuristic which proved to be an excellent starting point for the subsequent search. In Section 3.2 we describe this heuristic and discuss its worst

[1] The results of this chapter appear in: A. Lodi, S. Martello, D. Vigo, "Approximation Algorithms for the Oriented Two-Dimensional Bin Packing Problem", *European Journal of Operational Research* 112, 158–166, 1999, [108].

case performance. In Section 3.3 we define the neighborhoods used and give the relevant details of the tabu search algorithm. The results of extensive computational experiments are given in Section 3.4.

3.2 Initial feasible solution

In this section we describe and analyze a simple heuristic used to initialize the tabu search algorithm. Although its absolute worst case performance is poor and the solutions it produces require, on average, a high number of bins, it has experimentally proved to provide a very good starting point for the subsequent tabu search. Indeed, better starting solutions (e.g., those provided by algorithms FFF and FBS, see Section 3.3) proved to be much harder to modify through our tabu search. On the other hand, starting with trivial solutions typically increases the number of iterations needed to obtain good local optima.

The algorithm is based on a technique developed by Martello and Vigo [119] for proving the worst-case performance of the continuous lower bound for 2BP. We start by packing all the items into a strip having height H and infinite width: the strip is then cut so as to produce a feasible 2BP solution. Let (x, y) be a coordinate system with the origin in the bottom left corner of the strip. For each item j we define an attribute $class(j) = \min\{r : h_j \geq \frac{H}{2^{r+1}}, r \text{ integer}\}$: observe that any horizontal unit of strip containing 2^r items of class r has at least half of its area occupied by such items. Hence we define the set of admissible vertical coordinates for any item of class r as $V(r) = \{\frac{t}{2^r}H : t = 0, 1, \ldots, 2^r - 1\}$, and we pack the items according to nonincreasing class, at admissible vertical coordinates in such a way that the total occupied area before a current horizontal coordinate $\overline{x}$ is no less than $\overline{x}H/2$. The resulting strip is then subdivided into *slices* of width W and, for each slice, a feasible packing requiring one or two bins is determined. The algorithm can be outlined as follows. Set X contains the horizontal coordinates, greater than $\overline{x}$, corresponding to right edges of already packed items.

Algorithm IH
0. **for** $j := 1$ **to** n **do**
  if $h_j < H$ **then** $class(j) := \lceil \log_2(H/h_j) \rceil - 1$
  else $class(j) := 0$;
  sort (and renumber) the items by nondecreasing $class(j)$ values;
  $\overline{x} := 0$;
  $X := \varnothing$;
1. **for** $j := 1$ **to** n **do**
  begin
    if all admissible vertical coordinates at $\overline{x}$ are occupied **then**
      begin
        $\overline{x} := \min\{x : x \in X\}$;
        $X := X \setminus \{\overline{x}\}$
      end;
    place item j with its bottom left corner in $(\overline{x}, \overline{y})$, where $\overline{y}$ is the lowest unoccupied admissible vertical coordinate of $V(class(j))$, at $\overline{x}$;
    $X := X \cup \{\overline{x} + w_j\}$

end;

2. $k := \lfloor \overline{x}/W \rfloor$ (**comment :** observe that no item can terminate after $(k+2)W$);
subdivide the strip into $k+2$ slices of width W, starting from the origin;
for $i := 1$ **to** k **do**
begin
pack the items entirely contained in slice i, if any, into a new bin;
pack the items crossing the right boundary of slice i, if any, into a new bin
end;
pack the items terminating after $\overline{x}$, if any, into a new bin;
pack the remaining items of slice $k+1$, if any, into a new bin
end.

Figure 3.1 gives an example of packing obtained at the end of Step 1. (After renumbering, we have $[class(j)] = [0, 1, 1, 1, 2, 2, 2, 2, 2, 3]$; observe that ties may be broken arbitrarily.) At Step 2 we have $k = 2$: two bins are then introduced for slice 1 (with contents $\{1\}$ and $\{2, 3\}$), two for slice 2 (with contents $\{4, 5\}$ and $\{6, 7\}$), one for $\{9, 10\}$ and one for $\{8\}$.

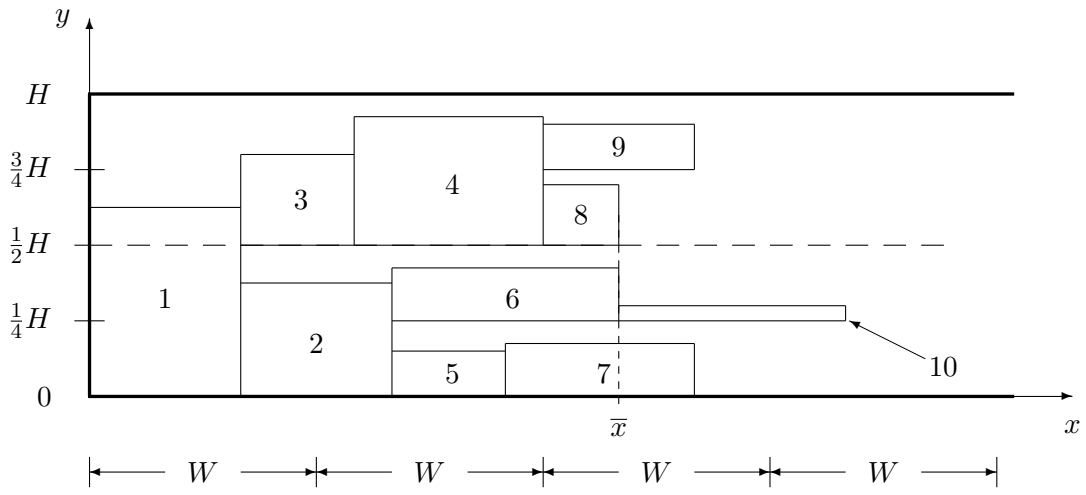


Figure 3.1: Example of the strip packing solution determined by algorithm IH.

It is easily seen that algorithm IH can be implemented so as to run in $O(n \log n)$ time. Indeed, by using a heap for storing set X, the operations involving set X at each iteration of the *for* loop in Step 1 may be performed in $O(\log n)$ time. On the other hand, whenever an item is placed in $(\overline{x}, \overline{y})$, the next unoccupied admissible vertical coordinate at $\overline{x}$ (if any) can be easily determined in constant time.

It is also clear that the patterns produced by IH are guillotine-cuttable. The items can indeed be obtained through the following recursive operations. Let $\widetilde{H}$ be the height of the current strip: obtain each item j having $h_j \geq \widetilde{H}/2$ through a vertical edge-to-edge cut, possibly followed by an horizontal edge-to-edge cut; subdivide the remaining part of the strip into two strips of height $\widetilde{H}/2$ through an horizontal edge-to-edge cut, and so on, recursively.

Given any instance I of a minimization problem P, let $z(I)$ be the value of the optimal solution to I, and $U(I)$ the value provided by a heuristic algorithm U. The *worst-case performance ratio* of U is defined as the smallest real number ρ such that

$$\frac{U(I)}{z(I)} \leq \rho \qquad \text{for all instances } I \text{ of } P.$$

Theorem 3.1. *The worst-case performance ratio of algorithm IH is 4.*

Proof. Let z^* denote the optimal number of bins and $\overline{z}$ the number of bins produced by IH. It has been proved in [119] that for any instance of 2BP we have $\overline{z} \leq 4LB$, where LB is a lower bound on z^*. Hence $\overline{z} \leq 4z^*$. To prove that this ratio is tight it is sufficient to consider an instance with $n = 6$, $h_1 = H/2$, $w_1 = W/2 + 1$, $h_2 = h_3 = h_4 = h_5 = h_6 = H/4$, $w_2 = w_3 = W/2$, $w_4 = w_6 = W/2 - 1$, $w_5 = W$. The strip packing solution determined by algorithm IH is shown in Figure 3.2(a): we then obtain four bins containing, respectively, $\{1\}$, $\{2, 3\}$, $\{5, 6\}$ and $\{4\}$. The optimal single bin solution is shown in Figure 3.2(b). □

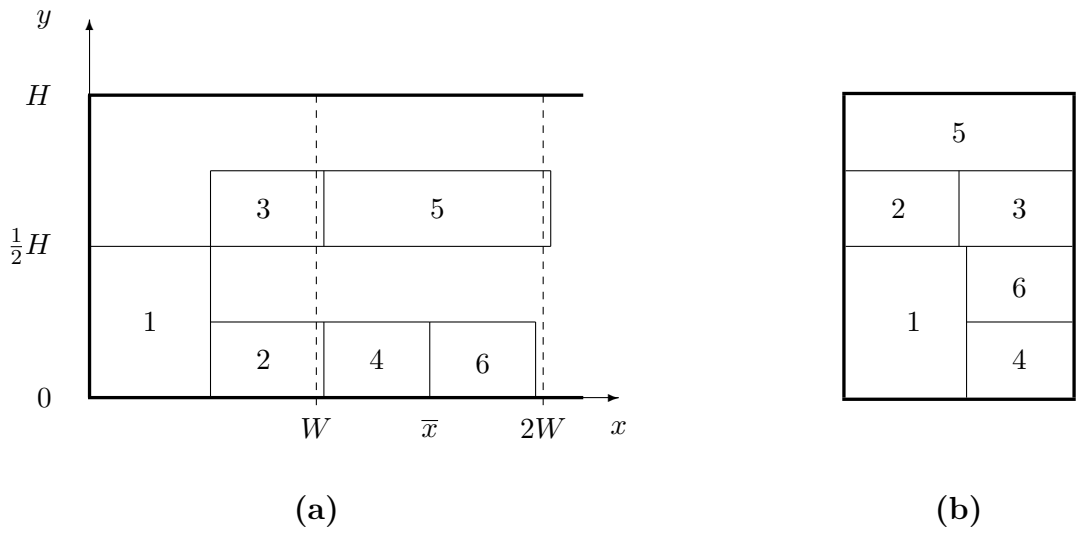


Figure 3.2: Worst-case example for algorithm IH.

The worst-case performance of IH is thus quite poor. Chung, Garey and Johnson [42] presented a hybrid algorithm, HFF, whose solution value satisfies $HFF(I)/z(I) < \frac{17}{8} + 5/z(I)$. This performance is better than that of IH for any instance I with $z(I) \geq 3$. However, as previously mentioned, the structure of the solutions provided by IH experimentally proved to be particularly suited for the tabu search initialization.

3.3 Tabu search

The algorithm starts by computing a lower bound LB as described in Martello and Vigo [119] and by applying two simple heuristics, called FFF (*Finite First-Fit*) and FBS (*Finite Best Strip*), described in Berkey and Wang [23].

The FFF heuristic initially sorts the items by nonincreasing height and then iteratively packs each of them into the first initialized bin in which it fits, or into a new bin if no such bin exists.

The FBS heuristic initially sorts the items by nonincreasing height and consists of two phases. First the items are packed into an infinite height strip using a best-fit algorithm to select the level for each item: the resulting strip packing is made up of "shelves", each corresponding to a different level, having width equal to the strip width and a different height. In the second phase the shelves are packed into finite bins using the well-known best-fit heuristic for the one-dimensional bin packing problem (see, Johnson, Demers, Ullman, Garey and Graham [98]). Algorithms FFF and FBS may be implemented to run in $O(n^2)$ and $O(n \log n)$ time, respectively. It is known that both FFF and FBS produce patterns that satisfy the guillotine constraint.

Let UB denote the value of the best heuristic solution found among those given by FFF and FBS. If $UB > LB$ the tabu search algorithm is initialized with the solution produced by algorithm IH. The search is based on a main loop where, at each iteration, one of two possible neighborhoods is explored. Let z denote the number of bins used in the current solution. The algorithm accepts moves that either decrease z or re-distribute items among the z bins, while moves increasing z are never accepted. As soon as no acceptable move is possible, a restart is performed.

Both neighborhoods consist of moves involving the items of a particular bin, which is called the *weakest bin* and is defined as follows. Let S_i denote the set of items that, in the current solution, are packed into bin i ($i = 1, \ldots, z$). The weakest bin is then defined as the one which minimizes the quantity

$$\varphi(i) = \alpha \, \frac{\sum_{j \in S_i} h_j w_j}{HW} - \frac{|S_i|}{n} \tag{3.1}$$

that gives an estimate of the difficulty of emptying bin i (α is a prefixed nonnegative parameter). At each iteration we consider an item j currently packed into the weakest bin b and we try to remove j from b: the first neighborhood tries to directly pack j into a different bin, while the second neighborhood tries to re-combine the items of two different bins so that one of them can accomodate j. The core of the algorithm can be outlined as follows:

```
determine the weakest bin b;
while a stopping criterion is not satisfied do
    begin
      repeat
        perform the next move according to the first neighborhood;
        if S_b = ∅ then determine the new weakest bin b
      until no move is possible;
      repeat
        perform the best move according to the second neighborhood;
        determine the new weakest bin b̄
      until b ≠ b̄;
      b := b̄
    end
```

Given a subset of items S, let $FBS(S)$ denote the number of $H \times W$ bins used by algorithm FBS for packing all items of S. Both neighborhoods use FBS for re-combining

subsets of items (note indeed that determining the *optimal* packing of a single bin is already an NP-hard problem).

The **first neighborhood** is obtained by determining the weakest bin b and by considering, in turn, each non-tabu item j currently packed into b: $FBS(\{j\} \cup S_i)$ is then iteratively computed for bins $i \neq b$. As soon as a bin i^* is found for which $FBS(\{j\} \cup S_{i^*}) = 1$, item j is moved to this bin, it is inserted in the first tabu-list, and the next item of b is considered. An aspiration criterion is used: if b contains a single item, then the search is performed even if this item is tabu, since the possible move would improve z. Whenever the weakest bin becomes empty, the new weakest bin is determined and the search continues with the first neighborhood. As soon as no move is possible, the search explores the second neighborhood. The first tabu list, which stores the indices of recently moved items, is preserved for the next exploration of the first neighborhood.

The second tabu list contains *scores* given by the total area of a subset of items representing a possible move. Also the **second neighborhood** considers, in turn, items j currently packed into the weakest bin b. For each of these j and for each pair of bins h and k, $(h, k \neq b)$, set $S = \{j\} \cup S_h \cup S_k$ is defined, and $FBS(S)$ is computed. Four cases may occur:

a. $FBS(S) = 1$, i.e., z is decreased by one or two units: the move is immediately performed and the exploration is resumed from the first neighborhood;

b. $FBS(S) = 2$, i.e., item j can be removed from bin b: the move is immediately performed. Two subcases are then possible:

   b.1. $|S_b| = 0$, i.e., the move has decreased z by one: the exploration is resumed from the first neighborhood;

   b.2. $|S_b| > 0$: the new weakest bin $\overline{b}$ is determined among b, h, and k. The search continues with the second neighborhood if $\overline{b} \equiv b$, or with the first neighborhood, otherwise;

c. $FBS(S) = 3$: let T be the set of items packed by FBS into the bin with minimum $\varphi(i)$ among the three resulting bins. Two subcases are possible:

   c.1. $FBS(T \cup (S_b \setminus \{j\})) > 1$: the move would increase z, hence it is not accepted;

   c.2. $FBS(T \cup (S_b \setminus \{j\})) = 1$: a *score*, defined by the φ value of the bin that packs the items in $T \cup (S_b \setminus \{j\})$, is assigned to the move. If this score is not tabu, the best acceptable move not improving z is possibly updated;

d. $FBS(S) > 3$: the move is not accepted since it would increase z.

At the end of the exploration the best non-improving acceptable move (if any) is performed, its score is stored in the second tabu list, and the search continues with the first neighborhood. For both tabu lists we used static tabu tenures of length τ_1 and τ_2, respectively.

The **stopping criteria** adopted are quite standard. The search terminates if (i) a feasible solution of value LB is determined, or (ii) a prefixed maximum number of iterations is attained, or (iii) a prefixed time limit is reached.

The algorithm also includes **restart actions**. If no acceptable move is found, or if the incumbent solution value z was not improved during the last μ moves, we modify the incumbent solution as follows. We renumber the bins so that $\varphi(i) \geq \varphi(i+1)$ for all i and execute

algorithm IH on the items of $\bigcup_{i=1}^{\lfloor z/2 \rfloor} S_i$, adding to the obtained solution the incumbent packings of bins $\lfloor z/2 \rfloor + 1, \ldots, z$. The resulting solution is then used to re-start the search.

We finally observe that the algorithm starts with a solution produced by IH (hence guillotine-cuttable) and modifies it through FBS, thus the resulting patterns satisfy the guillotine constraint.

3.4 Computational results

The tabu search algorithm was coded in FORTRAN 77 and run on a Silicon Graphics INDY R4000sc 100Mhz on test instances from the literature. The values we used for the parameters of the tabu search algorithm are $\alpha = 5.0$, $\tau_1 = 3$, $\tau_2 = 5$, $\mu = 50$.

Given any instance of 2BP, by interchanging H with W and h_j with w_j ($j = 1, \ldots, n$), we obtain an equivalent instance for which, however, a heuristic algorithm can produce a different solution. Hence, for each instance, we executed both the initial upper bound computation and the tabu search twice, getting the best solution: an overall limit of 100 CPU seconds was assigned for the solution of each instance (50 seconds per execution).

Table 3.1 gives the results obtained on instances from the literature with up to 120 items. Problems `beng1`–`beng8` are the instances used in Bengtsson [21] for the variant of 2BP where 90° rotation of the items is allowed. The remaining problems are instances proposed in the literature for other two-dimensional cutting problems and transformed into 2BP instances by using the relative bin and item sizes. In particular problems `cgcut1`–`cgcut3` are described in Christofides and Whitlock [41], while problems `gcut1`–`gcut13` and `ngcut1`–`ngcut12` are described in Beasley [17] and [18], respectively. Problems `beng1`–`beng8` and `ngcut1`–`ngcut12` were originally proposed to test algorithms not satisfying the guillotine constraint.

For each problem the table gives the problem name, the number of items, the values of LB, FFF, FBS and TS (tabu search solution) and the computing time, expressed in seconds, required by the tabu search. An asterisk indicates the instances where TS is optimal (the values of the optimal solutions were taken from [119]).

In Table 3.2 we present the results for six classes of instances randomly generated as in Berkey and Wang [23]. Each class is characterized by a different size of the bins and by the ranges in which the item sizes were uniformly randomly generated. The first three columns give the intervals in which the sizes of the items were uniformly randomly generated, the (fixed) bin sizes and the number of items. The values in each row are referred to ten random problem instances. We give the average ratios of FFF, FBS and TS with respect to LB, and the average tabu search computing time (expressed in seconds). The last column ($\leq B\&B$) gives the number of instances in which TS was at least equal to the solution value obtained by a branch-and-bound algorithm for the exact solution of the problem, presented in [119]; since the execution of such algorithm was interrupted after 300 CPU seconds, the solutions it provided were not necessarily optimal.

Table 3.3 examines new randomly generated instances described in [119], having a different mix of items. Four types of items were considered, each defined by different intervals in which the sizes were randomly generated (see Table 3.3). Each class is characterized by a different percentage of items generated for each type. The table gives, for each pair of size intervals, the percentage of items that were uniformly randomly generated in such interval (the bin size

Table 3.1: Results on problem instances from the literature. Time limit of 100 CPU seconds on a Silicon Graphics INDY R4000sc.

Name	n	LB	FFF	FBS	TS		$time$
beng1	20	4	4	4	4	*	0.01
beng2	40	6	7	7	7		100.02
beng3	60	9	10	9	9	*	0.01
beng4	80	11	12	12	12		100.06
beng5	100	14	16	15	14	*	0.01
beng6	40	2	2	2	2	*	0.01
beng7	80	3	3	3	3	*	0.01
beng8	120	5	5	5	5	*	0.01
cgcut1	16	2	2	2	2	*	0.01
cgcut2	23	2	3	3	2	*	0.01
cgcut3	62	23	26	26	23	*	0.01
gcut1	10	4	5	5	5	*	100.02
gcut2	20	6	7	7	6	*	50.11
gcut3	30	8	9	8	8	*	0.01
gcut4	50	13	15	15	14	*	100.03
gcut5	10	3	4	4	4		100.02
gcut6	20	6	8	8	7	*	100.02
gcut7	30	10	12	12	12		100.01
gcut8	50	12	15	14	14		100.03
gcut9	10	3	3	3	3	*	0.01
gcut10	20	7	8	8	8		100.03
gcut11	30	8	10	10	9	*	100.03
gcut12	50	16	17	17	16	*	23.56
gcut13	32	2	2	2	2	*	0.01
ngcut1	10	2	3	3	3	*	100.02
ngcut2	17	3	4	4	4	*	100.02
ngcut3	21	3	4	4	4		100.02
ngcut4	7	2	2	2	2	*	0.01
ngcut5	14	3	4	4	3	*	0.01
ngcut6	15	2	3	3	3	*	100.02
ngcut7	8	1	2	2	1	*	0.01
ngcut8	13	2	2	2	2	*	0.01
ngcut9	18	3	4	4	4		100.02
ngcut10	13	3	3	3	3	*	0.01
ngcut11	15	2	3	3	3		100.02
ngcut12	22	3	4	4	4		100.02

was always 100×100), the number of items, plus the same information as in Table 3.2. In this case too the values in each row are referred to ten random problem instances.

Table 3.2: Results on the random problem instances proposed by Berkey and Wang. Time limit of 100 CPU seconds on a Silicon Graphics INDY R4000sc.

items	*bins*	n	FFF/LB	FBS/LB	TS/LB	*time*	$\leq B\&B$
		20	1.165	1.136	1.061	40.01	10
		40	1.122	1.090	1.072	85.13	8
$[1,10] \times [1,10]$	10×10	60	1.096	1.074	1.047	90.03	9
		80	1.083	1.060	1.030	67.89	8
		100	1.074	1.061	1.035	96.92	7
		20	1.100	1.100	1.100	0.01	9
		40	1.100	1.100	1.100	0.01	10
$[1,10] \times [1,10]$	30×30	60	1.150	1.150	1.150	30.00	10
		80	1.067	1.067	1.033	15.00	10
		100	1.058	1.058	1.033	10.00	10
		20	1.197	1.177	1.177	80.01	7
		40	1.177	1.136	1.110	90.01	8
$[1,35] \times [1,35]$	40×40	60	1.138	1.106	1.078	80.13	7
		80	1.131	1.099	1.073	90.02	8
		100	1.123	1.091	1.072	100.04	7
		20	1.000	1.000	1.000	0.01	10
		40	1.100	1.100	1.100	0.01	10
$[1,35] \times [1,35]$	100×100	60	1.200	1.200	1.200	40.00	10
		80	1.100	1.100	1.100	30.01	10
		100	1.100	1.100	1.067	20.00	10
		20	1.140	1.140	1.106	60.01	9
		40	1.105	1.105	1.087	80.01	6
$[1,100] \times [1,100]$	100×100	60	1.111	1.099	1.062	91.29	8
		80	1.119	1.089	1.060	92.22	6
		100	1.121	1.091	1.070	97.64	6
		20	1.000	1.000	1.000	0.01	10
		40	1.400	1.400	1.400	0.01	10
$[1,100] \times [1,100]$	300×300	60	1.100	1.100	1.050	10.00	10
		80	1.000	1.000	1.000	0.01	10
		100	1.133	1.100	1.100	30.00	10

The results of Tables 1–3 show a very good behaviour of the proposed tabu search algorithm. The TS/LB ratios are considerably lower than the FFF/LB and FBS/LB ratios (remind that algorithms FFF and FBS are classical and effective methods from the literature). Also the comparison with the branch-and-bound approach is satisfactory. For 26 out of 36 instances from the literature the exact solution was obtained, while for 90% of the randomly generated instances of Tables 3.2 and 3.3 the tabu search approach attained the same performance, in terms of solution quality, as a branch-and-bound algorithm.

Table 3.3: Results on the random problem instances proposed by Martello and Vigo. Time limit of 100 CPU seconds on a Silicon Graphics INDY R4000sc.

$h_j \in [1, \frac{H}{2}]$	$[\frac{2}{3}H, H]$	$[\frac{H}{2}, H]$	$[1, \frac{H}{2}]$						
$w_j \in [\frac{2}{3}W, W]$	$[1, \frac{W}{2}]$	$[\frac{W}{2}, W]$	$[1, \frac{W}{2}]$	n	FFF/LB	FBS/LB	TS/LB	$time$	$\leq B\&B$
				20	1.098	1.098	1.062	30.09	9
				40	1.107	1.107	1.070	70.01	7
70%	10%	10%	10%	60	1.077	1.077	1.044	64.91	9
				80	1.073	1.058	1.041	91.95	10
				100	1.045	1.041	1.030	73.79	9
				20	1.173	1.157	1.078	40.01	9
				40	1.093	1.084	1.022	30.69	10
10%	70%	10%	10%	60	1.062	1.062	1.025	46.72	10
				80	1.067	1.063	1.028	71.42	8
				100	1.062	1.062	1.030	83.75	10
				20	1.014	1.007	1.000	0.01	10
				40	1.022	1.019	1.014	40.01	10
10%	10%	70%	10%	60	1.018	1.016	1.009	40.04	10
				80	1.020	1.016	1.014	80.07	10
				100	1.018	1.015	1.009	50.03	10
				20	1.137	1.137	1.137	50.01	7
				40	1.145	1.089	1.075	50.01	9
10%	10%	10%	70%	60	1.146	1.119	1.084	80.17	9
				80	1.147	1.131	1.088	96.15	9
				100	1.137	1.104	1.073	100.09	10

Chapter 4

Two-Dimensional Bin Packing: the 2BP|R|G case

4.1 Introduction

In[1] two-dimensional geometrical packing problems we are given a set of n rectangular *items*, each characterized by width w_j and height h_j ($j = 1, \ldots, n$), which must be cut from standardized stock items. Two main versions of the problem have been considered in the literature: strip packing and bin packing. In the *Two-Dimensional Strip Packing Problem* we have a unique stock item (*strip*) having width W and infinite height: the objective is to determine a cutting pattern that provides all the items, such that the height to which the strip is filled is minimized. In the *Two-Dimensional Bin Packing Problem* we have an unlimited number of identical rectangular stock items (*bins*) having width W and height H: the objective is to determine cutting patterns that provide all the items, such that the total number of required bins is minimized.

Both problems are strongly NP-hard, as can be easily seen by transformation from the well-known (one-dimensional) *Bin Packing Problem*, in which a set of n positive values w_j has to be partitioned into the minimum number of subsets so that the total value in each subset does not exceed a given bin capacity W.

Two-dimensional packing problems have a number of important applications in industrial cutting (of glass, wood, textiles, paper, ...), and in logistics and transportation (packing on floors, shelves, truck beds, ...). According to specific applications, for both versions of the problem several variants are possible. For example, the items may have fixed orientation, i.e., they cannot be rotated, or *rotation* (usually by 90 degrees) may be allowed. Moreover, in cutting problems it is frequently required that the patterns be such that the items can be obtained by sequential edge-to-edge cuts parallel to the edges of the bin or of the strip (*guillotine cutting*).

In this chapter we consider the two-dimensional bin packing problem, with 90 degrees rotation allowed and guillotine cutting constraint: the problem will be denoted by 2BP|R|G. A practical situation where this type of problem is relevant arises in the glass industry: the

[1]The results of this chapter appear in: A. Lodi, S. Martello, D. Vigo, "Neighborhood Search Algorithm for the Guillotine Non-Oriented Two-Dimensional Bin Packing Problem", in S. Voss, S. Martello, I.H. Osman, C. Roucairol, Eds., *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, Kluwer Academic Publishers, Boston, 1998, 125–139, [107].

cutting machines can only operate in guillotine mode, and the orientation of the items in a pattern is inessential (if the glass is not decorated). Real world numerical instances of this type of problem are included in the computational experiments of Section 4.4.

Two-dimensional packing problems received considerable attention in the literature, although, to our knowledge, no work has been devoted explicitly to 2BP|R|G. For the two-dimensional bin packing problem in the case where no rotation is allowed and the guillotine constraint is not imposed, Martello and Vigo [119] presented lower bounds and an exact branch-and-bound algorithm, while Chung, Garey and Johnson [42] and Frenk and Galambos [76] gave approximation algorithms with worst case analysis. Berkey and Wang [23] and Lodi, Martello and Vigo [108] proposed heuristic algorithms for the variant in which rotation is not allowed but the guillotine constraint is satisfied. Bengtsson [21] and El-Bouri, Popplewell, Balakrishnan and Alfa [65] gave heuristic algorithms for the case where rotation is allowed but the guillotine constraint is not imposed. Approximation algorithms for the two-dimensional strip packing problem were studied by Baker, Coffman and Rivest [7], Coffman, Garey, Johnson and Tarjan [43], Bartholdi, Vande Vate and Zhang [16], Coffman and Shor [44] and Golan [83]. An annotated bibliography on the cutting and packing area has been presented by Dyckhoff, Scheithauer and Terno ([64]).

In Section 4.2 we show how some classical greedy algorithms given in the literature for the case where no rotation is allowed can be adapted to our problem; in addition, an original heuristic algorithm is presented. In Section 4.3 we give a tabu search approach to 2BP|R|G, obtained by improving the basic scheme used in [108] and by incorporating original features. An extensive computational testing is presented in Section 4.4, showing the effectiveness of the proposed algorithms.

We assume in the following, without loss of generality, that all input data are positive integers satisfying $\min\{w_j, h_j\} \leq \min\{W, H\}$ and $\max\{w_j, h_j\} \leq \max\{W, H\}$ for $j = 1, \ldots, n$.

4.2 Greedy heuristics

Berkey and Wang [23] experimentally compared simple and effective algorithms, satisfying the guillotine constraint, for the case where no rotation is allowed: good average results were obtained by the algorithms called FFF and FBS. Both algorithms start by sorting the items according to nonincreasing height, and pack the items in rows forming levels (*shelves*). The first shelf is the bottom of a bin. Pieces are packed from left to right with their bottoms at the selected level. When a new level is needed, it is created along the horizontal line which coincides with the top of the first (i.e., tallest) item packed in the lower level.

In the *Finite First Fit* (FFF) algorithm the bins are directly stratified into shelves. The next item j is packed left-justified into the lower level of the first bin where it fits. If no shelf can accommodate j, then a new level is created in the first suitable bin. If no bin has enough vertical space for a new level accommodating j, then a new bin is created and initialized with j.

The *Finite Best Strip* (FBS) algorithm works in two phases. First, a strip of width W and infinite height is stratified into shelves according to the following strategy. The next item j is packed into the feasible shelf having the minimum residual horizontal space; if no shelf can accommodate j a new shelf is created. Let $h_1, \ldots, h_m$ be the heights of the resulting shelves. In the second phase, the shelves are combined into finite bins using the well-known *Best-Fit Decreasing* algorithm for a one-dimensional bin packing instance consisting of the m values

h_j with bin capacity H.

As previously mentioned, FFF and FBS produce patterns satisfying the guillotine constraint: a sequence of horizontal cuts separates the shelves and, for each shelf, a sequence of vertical and horizontal cuts produces the items. We have modified such algorithms so as to handle the 90 degrees item rotations: we call these modified versions $\mathrm{FFF_{RG}}$ and $\mathrm{FBS_{RG}}$. We assume, without loss of generality, that $W \geq H$; we will say that an item is packed *horizontally* (resp. *vertically*) if its longest (resp. shortest) edge is parallel to the bottom of the shelf.

Both modified algorithms start by sorting the items according to nonincreasing value of the shortest edge. Algorithm $\mathrm{FFF_{RG}}$ is like FFF, with the following extensions:

(i) an item that initializes a shelf is always horizontally packed, so as to keep as low as possible the vertical occupancy of the corresponding bin;

(ii) when an item is packed into an existing shelf, if both orientations are feasible then the vertical one is selected, so as to keep as low as possible the horizontal occupancy of the shelf.

Analogously, the extension of FBS to $\mathrm{FBS_{RG}}$ is obtained by modifying the evaluation of a possible placement: if both orientations of the item are feasible then the vertical one is used, so as to maximize the probability of additional packings on the shelf.

We now introduce a new heuristic for 2BP|R|G, which works as $\mathrm{FBS_{RG}}$, in two phases but differs from it in the way the items are packed into the shelves. A shelf is initialized by placing an item with its left edge touching the left edge of the strip, into the lowest admissible position: we call *floor* (resp. *ceiling*) of the shelf the horizontal line touched by the bottom (resp. top) edge of such item. Algorithms $\mathrm{FFF_{RG}}$ and $\mathrm{FBS_{RG}}$ pack the items exclusively on the floor. Our algorithm *Floor-Ceiling* ($\mathrm{FC_{RG}}$), instead, packs the items into the shelves either with their bottom edge touching the floor or with their top edge touching the ceiling, in such a way that guillotine cutting of the items allocated to the shelves is possible. On the floor the items are packed from left to right, while on the ceiling they are packed from right to left in an attempt to fill the space over the lowest items packed on the floor. A item packed on the floor has always its left edge touching the right edge of another item packed on the floor, while items packed on the ceiling may be separated by empty spaces in order to allow guillotine packing.

The *right-end* of the shelf coincides with the right edge of the strip, while its *left-end* is the right edge of the rightmost item packed on the floor and such that the vertical space between the top edge of the item and the ceiling is too small to allocate any of the items that are not yet packed. Figure 4.1 illustrates the definitions and properties above.

We say that an item is *floor-feasible* for a given shelf if there is room for packing it (in either orientation) on the floor, at the right of the rightmost item already packed on the floor. A shelf is called *ceiling-initialized* if there are items packed on its ceiling. Ceiling initialization is always performed by packing an item that is *not* floor-feasible: the item is packed with its right edge touching the right edge of the strip (see item number 5 in Figure 4.1), in an orientation chosen according to rules given below.

A item is *ceiling-feasible* for a given shelf if either it can initialize the ceiling (see above) or it can be packed on the ceiling, in either orientation, to the left of the items already packed on the ceiling and in such a way that guillotine cutting is possible. To this end, first observe

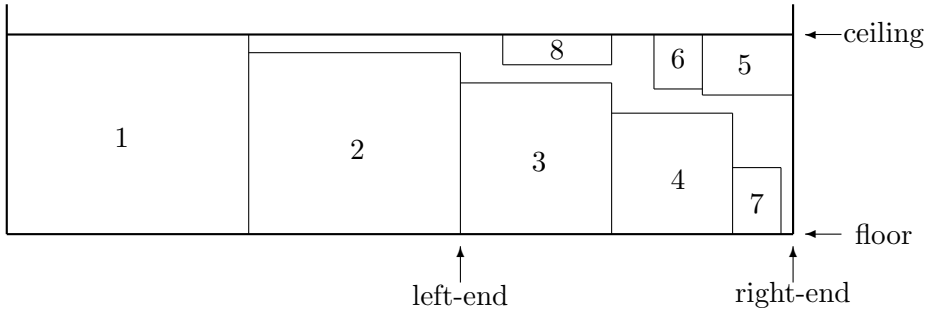


Figure 4.1: A shelf pattern produced by algorithm $\mathrm{FC_{RG}}$.

that a series of horizontal edge-to-edge cuts along the ceilings separates the shelves. Given a shelf:

(i) a series of vertical cuts obviously separates the items packed on the floor;

(ii) when an item j^* initializes the ceiling there is a horizontal cut C^* along its lower edge and extending from the right edge of the strip to the vertical cut $\overline{C}$ that separates (see (i)) the leftmost item packed on the floor which occupies an horizontal strip portion in common with j^*: hence no further item will be packed so as to cross C^*;

(iii) each item packed on the ceiling to the left of $\overline{C}$ is placed in the rightmost position where it fits without crossing any line extending the vertical edge of an item packed on the floor (see Figure 4.2).

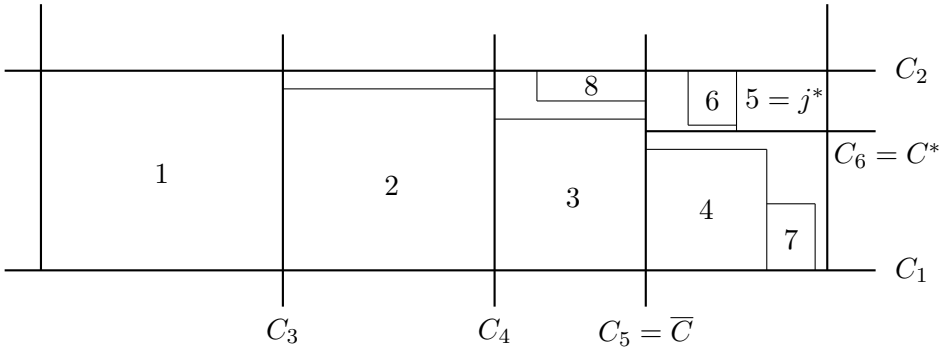


Figure 4.2: Guillotine-feasible cutting pattern for the solution of Figure 4.1.

The algorithm can thus be outlined as follows. (The term "best", used to select the shelf, will be defined later.)

Algorithm $\mathrm{FC_{RG}}$

1. sort and renumber the items according to nonincreasing value of the shortest edge;

2. **for** $j := 1$ **to** n **do**
  if j is ceiling-feasible for some shelf **then**
    pack j on the ceiling of the best shelf
  else
    if j is floor-feasible for some shelf **then**
      pack j on the floor of the best shelf
    else
      initialize a new shelf by horizontally packing j;
3. use a Best-Fit Decreasing 1-dimensional bin packing algorithm to pack the resulting shelves into finite bins.

(Note that, once the ceiling has been initialized, we always prefer ceiling packings, since a good filling of the upper portions of the shelves tends to delay the initialization of new shelves).

When both orientations are feasible, the item is always packed in the orientation which favors the possibility of future packings within the considered shelf. To this end, for ceiling placements we always select the vertical orientation, since this maximizes the residual horizontal space in the upper part of the shelf. For floor placements, a strategy depending on the current status of the shelf is adopted: if the shelf is ceiling-initialized, the vertical orientation is selected, since this maximizes the residual horizontal space in the lower part of the shelf; otherwise, the horizontal orientation is selected, thus increasing the possibility of future ceiling placements.

In order to decide which is the best shelf, we use a *score* defined by the residual horizontal space that would be left by such placing. More precisely, for a floor (resp. ceiling) placing the score is the distance between the right (resp. left) edge of the item and the right-end (resp. left-end) of the shelf: the best solution is then the one corresponding to the minimum score, in the spirit of the best-fit rule for the 1-dimensional bin packing problem.

4.3 Tabu-Search

In this section we present a tabu search approach algorithm for 2BP|R|G. The reader is referred to [82] for a general introduction to tabu search and to basic definitions.

Our algorithm combines the strategy adopted in [108] (based on a main loop where, at each iteration, two neighborhoods are explored) and an original restart policy (that can be viewed as a third neighborhood). The general structure of the algorithm is outlined below, while a detailed description of the various components is given in the following sections.

Algorithm TS
1. initialize the tabu lists to empty and determine a starting feasible solution;
2. select a bin b;
  comment: first neighborhood;
  for each non-tabu item j currently packed into b **do**
    try to move j to a different bin i^*;
    comment: the tabu list stores the recently moved items;
  if bin b is empty **then go to** 2
  else
    begin

3. **comment**: second neighborhood;
for each item j currently packed into b **do**
try to re-combine the contents of two bins h,k ($\neq b$) so that one of them can accommodate j, and associate a score with such move;
if *restart conditions* do not apply **then**
perform the non-tabu move with the highest score, store such score in the second tabu list and **go to** 2
else
if *stopping conditions* do not apply **then**
4. perform a restart action and **go to** 1
end.

In the next sections we describe in detail the various steps of algorithm TS.

4.3.1 Initialization and basic definitions

The algorithm starts by applying the heuristics of the previous section and computing the continuous lower bound

$$LB = \left\lceil \frac{\sum_{j=1}^{n} w_j h_j}{WH} \right\rceil \tag{4.1}$$

Let UB denote the value of the best heuristic solution found. If $UB > LB$ the tabu search algorithm is initialized with the solution produced by algorithm $\mathrm{FC_{RG}}$. Let z denote the number of bins used in the current solution. The algorithm accepts moves that either decrease z or re-distribute items among the z bins; moves increasing z are never accepted. Restart strategies are provided in order to have diversification and intensification during the search (see below). The search terminates if a feasible solution of value LB is determined, or if a prefixed maximum number of iterations (or a time limit) is reached.

Both neighborhoods consist of moves involving the items of a particular bin, which is called the *weakest bin* and is defined as follows. Let S_i denote the set of items that, in the current solution, are packed into bin i ($i = 1, \ldots, z$). The weakest bin is then defined as the one which minimizes the quantity

$$\varphi(i) = \alpha \, \frac{\sum_{j \in S_i} w_j h_j}{WH} - \frac{|S_i|}{n} \tag{4.2}$$

that gives an estimate of the difficulty of emptying bin i: a weak bin packs a small fraction of the available area (first term) using a relatively high number of items (second term); α is a prefixed nonnegative parameter measuring the relative weight of the two terms. Given a subset of items S, let $FC_{\mathrm{RG}}(S)$ denote the number of $W \times H$ bins used by algorithm $\mathrm{FC_{RG}}$ to pack all the items of S.

4.3.2 First neighborhood

Let b be either the weakest bin determined through (4.2), or, in case of restart (see Section 4.3.4), a bin received on input. In the first neighborhood we consider, in turn, each non-tabu item $j \in S_b$ and we compute $FC_{\mathrm{RG}}(\{j\} \cup S_i)$ for all bins $i \neq b$: whenever a bin i^* is found for which $FC_{\mathrm{RG}}(\{j\} \cup S_{i^*}) = 1$, item j is moved into bin i^*, it is inserted into tabu-list $T1$,

and the next item of b is considered. The following aspiration criterion is adopted: if $|S_b| = 1$ then the search is performed even if item j is tabu, since the possible move would improve z. When a series of moves empties the weakest bin, z is decreased by one, the new weakest bin is determined and the search continues.

As soon as no move is possible, the search explores the second neighborhood. Tabu list $T1$, which stores the recently moved items, is preserved for the next exploration of the first neighborhood.

4.3.3 Second neighborhood

In the second neighborhood, for each item $j \in S_b$ and for each pair of bins h and k $(h, k \neq b)$, we compute $FC = FC_{\mathrm{RG}}(\{j\} \cup S_h \cup S_k)$. Four cases may occur:

- if $FC = 1$ then z may be decreased by one or two units, so the move is immediately performed and the exploration is resumed from the first neighborhood;
- if $FC = 2$ then item j is removed from bin b and two subcases are considered:
  - if $|S_b| = 0$ then z is decreased by one and the exploration is resumed from the first neighborhood;
  - if $|S_b| > 0$ then the new weakest bin $\overline{b}$ is determined among b, h, and k and the search continues with the second neighborhood if $\overline{b} \equiv b$, or with the first neighborhood, otherwise;
- if $FC = 3$ then let Q be the set of items packed by $\mathrm{FC_{RG}}$ into the bin with minimum $\varphi(i)$ among the three resulting bins: we compute $\overline{FC} = FC_{\mathrm{RG}}(Q \cup (S_b \setminus \{j\}))$ and consider two subcases:
  - if $\overline{FC} > 1$ then the move is not accepted since it would increase z;
  - if $\overline{FC} = 1$ then we assign to the move a *score* defined by the φ value of the resulting packing of the items in $Q \cup (S_b \setminus \{j\})$: if this score is not tabu, the best acceptable move not improving z is possibly updated;
- if $\overline{FC} > 3$ then the move is rejected since it would increase z.

At the end of the exploration the best non-improving acceptable move (if any) is performed, its score is stored in the tabu list $T2$, and the search continues with the first neighborhood. For both tabu lists we used static tabu tenures of length τ_1 and τ_2, respectively.

4.3.4 Restart (third neighborhood)

The algorithm performs three different types of restart actions (denoted by R1, R2 and R3) obtained through a combination of diversification and intensification strategies. Some of these actions imply to *fix* a bin, i.e., to continue the search over a reduced instance obtained by removing the items currently packed into the fixed bins and assuming that the corresponding patterns constitute an imposed partial solution:

- R1: let b be the current weakest bin, define $\overline{b}$ through $\varphi(\overline{b}) = \min\{\varphi(i) : \varphi(i) \geq \varphi(b), i \neq b\}$ (see 3.1) and restart from the first neighborhood giving $\overline{b}$ on input;

- R2: let P be the set of items currently packed into the $\lfloor z/2 \rfloor$ less weak non fixed bins (i.e., non fixed bins which, in the current solution, have the highest φ values), re-pack each item of P into a separate bin and restart from the first neighborhood. In this way the current solution is increased by $|P| - \lfloor z/2 \rfloor$;

- R3: let $\widetilde{b}$ be the less weak non fixed bin, fix it and perform a restart of type R2.

These restart actions are performed in two different situations (called S1 and S2), arising at the end of the exploration of the second neighborhood:

- S1 = no acceptable move was found, i.e., some of the moves are tabu and the others would produce an increase of the value of z;

- S2 = z was not improved during the last μ moves.

In other words, the term "restart conditions" in the **if** statement of Step 3 of algorithm TS means: S1 **or** S2.

The restart action of Step 4 is executed as follows. Let f be the number of currently fixed bins and r be the number of restarts of type R1 recently performed. Initially, f and r take value 0; t denotes the computing time spent so far; R and T are prefixed parameters:

```
...
4.  if S1 and r < R then
        perform a restart of type R1 and set r := r + 1
    else
        begin
            set r := 0;
            if t < T then
                perform a restart of type R2
            else
                if f < ⌊z/2⌋ then
                    set f := f + 1 and perform a restart of type R3
                else
                    reinsert into the instance all the items packed into the fixed bins,
                    set f := 0 and perform a restart of type R2
        end;
...
```

The restart of type R1 can be seen as a combination of intensification and diversification, that of type R2 as a pure diversification and that of type R3 as an evolution of R2 obtained by adding an intensification strategy. In particular, we follow the idea that after T time units the current solution is becoming more and more well-structured, so we try to speed up to the convergence to a local optimum.

The fixing strategy adopted in the restart of type R3 can be seen both as the definition of a *third neighborhood* (fixed bins are tabu) and as a *cooling strategy* according to a simulated annealing approach (in some sense we decrease the probability that bad moves are accepted).

4.4 Computational experiments

We coded the tabu search algorithm in FORTRAN 77 and run it on a Silicon Graphics INDY R10000sc 195Mhz on test instances from the literature. We assigned for the solution of each instance a time limit L of 100 CPU seconds.

We selected, through a series of preliminary tests, the following values for the parameters: $\alpha = 5.0$, $\tau_1 = 3$, $\tau_2 = 5$, $R = 5$, $T = L/5$ and $\mu = 50$.

Table 4.1 gives the results obtained on single instances involving up to 164 items. Problems `beng1`–`beng8` are the instances of Bengtsson [21]. The remaining problems are instances proposed in the literature for other two-dimensional cutting problems and transformed into 2BP|R|G instances by using the relative bin and item sizes (if the demand of an item j is d_j, we consider d_j different items with height h_j and width w_j): problems `cgcut1`–`cgcut3` are described in Christofides and Whitlock [41], while problems `gcut1`–`gcut13` and `ngcut1`–`ngcut12` are described in Beasley [17] and [18], respectively. In the last four lines of Table 4.1 we report the results with four new instances, `glass1`–`glass4`, of a real-world 2BP|R|G problem arising in a glass factory (the instances are available on request from the first author).

For each problem the table gives the problem name, the number of items, the value of LB, the value of the best solution found by FFF_{RG} and FBS_{RG} (indicated in tables with UB_F), the value of the solution found by FC_{RG}, the value TS found from tabu search and the computing time, expressed in seconds, required by the tabu search. An asterisk indicates the instances for which it could be proved that the solution found by tabu search is optimal (by comparison with LB and other lower bounds studied in Dell'Amico, Martello and Vigo [56]).

In Table 4.2 we present the results for six classes of instances randomly generated as in Berkey and Wang [23]. Each class is characterized by a different size of the bins and by the ranges in which the item dimensions were uniformly randomly generated. The table gives the intervals in which the sizes of the items were uniformly randomly generated, the (fixed) bin sizes, the number of items, the average ratios of the solutions obtained by greedy heuristics and by TS with respect to the value of LB, the average tabu search computing time, expressed in seconds, and the number of instances for which it can be proved that the solution found by tabu search is optimal. The entries in each row are average values computed over ten random problem instances.

In Table 4.3 we consider randomly generated instances described in [119] having a different mix of items. Four types of items were considered, each defined by different intervals in which the sizes were uniformly randomly generated (see Table 4.3). Each class is characterized by a different percentage of items generated for each type. The table gives, for each pair of size intervals, the percentage of items that were uniformly randomly generated in such interval (the bin size is always 100×100), plus the same information as in Table 4.2.

Tables 4.1, 4.2 and 4.3 show a satisfactory behavior both of the new heuristic and of the tabu search algorithm. With only a few exceptions (four times over a total of 540 instances), the solution found by FC_{RG} dominates the best solution found by FFF_{RG} and FBS_{RG}. The tabu search algorithm finds in many cases the optimal solution or a solution of value close to that of the lower bound. For the third class of Table 4.3 the high value of the ratios are only due to a very poor behavior of the continuous lower bound: indeed the solutions found are often optimal (see the last column) or very close to the optimum (as could be proved by comparison with the improved lower bounds presented in [56]).

The effectiveness of the tabu search approach is due to the efficient combination of the

Table 4.1: Results on problem instances from the literature and on four new real-world instances arising in glass industry. Time limit of 100 CPU seconds on a Silicon Graphics INDY R10000sc.

Name	n	LB	UB_F	FC_{RG}	TS		$time$
beng1	20	3	4	4	4		100.01
beng2	40	6	7	7	7		100.01
beng3	60	9	9	9	9	*	0.01
beng4	80	11	12	12	11	*	3.70
beng5	100	14	14	14	14	*	0.01
beng6	40	2	2	2	2	*	0.01
beng7	80	3	3	3	3	*	0.01
beng8	120	5	5	5	5	*	0.01
cgcut1	16	2	2	2	2	*	0.01
cgcut2	23	2	3	2	2	*	0.01
cgcut3	62	16	23	24	24		100.01
gcut1	10	4	4	4	4	*	100.01
gcut2	20	5	7	7	7		100.01
gcut3	30	7	9	9	8		100.01
gcut4	50	12	15	15	14		100.01
gcut5	10	3	3	3	3	*	0.01
gcut6	20	5	8	8	8		100.01
gcut7	30	9	11	11	11		100.01
gcut8	50	12	14	14	14		100.02
gcut9	10	3	3	3	3	*	0.01
gcut10	20	6	8	8	8		100.01
gcut11	30	7	10	10	9		100.01
gcut12	50	13	17	17	16		100.01
gcut13	32	2	2	2	2	*	0.01
ngcut1	10	2	3	3	3		100.01
ngcut2	17	3	4	4	4		100.01
ngcut3	21	3	4	4	4		100.01
ngcut4	7	2	2	2	2	*	0.01
ngcut5	14	3	4	4	4		100.01
ngcut6	15	2	3	3	3		100.01
ngcut7	8	1	1	1	1	*	0.01
ngcut8	13	2	2	2	2	*	0.01
ngcut9	18	3	4	4	4		100.01
ngcut10	13	2	3	3	3		0.01
ngcut11	15	2	3	3	3		100.01
ngcut12	22	3	4	4	4		100.01
glass1	82	8	9	9	8	*	3.75
glass2	112	11	13	13	12		100.02
glass3	146	15	19	18	17		100.07
glass4	164	9	11	11	11		100.28

Table 4.2: Results on the random problem instances proposed by Berkey and Wang. Time limit of 100 CPU seconds on a Silicon Graphics INDY R10000sc.

items	*bins*	n	$\frac{UB_F}{LB}$	$\frac{FC_{\mathrm{RG}}}{LB}$	$\frac{TS}{LB}$	*time*	*#opt*
$[1,10]\times[1,10]$	10×10	20	1.06	1.06	1.06	40.00	6
		40	1.12	1.11	1.09	70.41	3
		60	1.10	1.10	1.09	92.76	1
		80	1.10	1.10	1.08	91.08	1
		100	1.08	1.07	1.05	72.31	0
$[1,10]\times[1,10]$	30×30	20	1.00	1.00	1.00	0.01	10
		40	1.10	1.10	1.10	0.01	9
		60	1.10	1.05	1.05	10.00	9
		80	1.07	1.03	1.03	10.00	9
		100	1.03	1.03	1.03	10.00	9
$[1,35]\times[1,35]$	40×40	20	1.25	1.22	1.16	50.02	5
		40	1.21	1.21	1.20	100.01	0
		60	1.18	1.20	1.12	100.02	0
		80	1.16	1.16	1.11	100.03	0
		100	1.16	1.15	1.11	99.29	0
$[1,35]\times[1,35]$	100×100	20	1.00	1.00	1.00	0.01	10
		40	1.10	1.00	1.00	0.01	10
		60	1.10	1.10	1.10	20.01	8
		80	1.10	1.10	1.10	30.00	7
		100	1.07	1.07	1.07	20.01	8
$[1,100]\times[1,100]$	100×100	20	1.12	1.12	1.11	30.00	7
		40	1.18	1.17	1.17	70.01	3
		60	1.16	1.16	1.14	100.02	0
		80	1.15	1.16	1.15	100.03	0
		100	1.13	1.13	1.10	100.04	0
$[1,100]\times[1,100]$	300×300	20	1.00	1.00	1.00	0.01	10
		40	1.40	1.40	1.40	0.01	6
		60	1.05	1.05	1.05	10.00	9
		80	1.00	1.00	1.00	0.01	10
		100	1.07	1.07	1.07	20.01	8

Table 4.3: Results on the random problem instances proposed by Martello and Vigo. Time limit of 100 CPU seconds on a Silicon Graphics INDY R10000sc.

$h_j \in [1, \frac{H}{2}]$	$[\frac{2}{3}H, H]$	$[\frac{H}{2}, H]$	$[1, \frac{H}{2}]$						
$w_j \in [\frac{2}{3}W, W]$	$[1, \frac{W}{2}]$	$[\frac{W}{2}, W]$	$[1, \frac{W}{2}]$	n	$\frac{UB_F}{LB}$	$\frac{FC_{\mathrm{RG}}}{LB}$	$\frac{TS}{LB}$	*time*	*#opt*
				20	1.19	1.19	1.17	80.01	2
				40	1.17	1.17	1.17	100.02	0
70%	10%	10%	10%	60	1.18	1.18	1.16	100.04	0
				80	1.17	1.17	1.16	100.06	0
				100	1.17	1.17	1.16	100.07	0
				20	1.17	1.17	1.17	62.05	3
				40	1.19	1.19	1.19	100.02	0
10%	70%	10%	10%	60	1.18	1.18	1.17	100.03	0
				80	1.16	1.16	1.15	100.07	0
				100	1.17	1.17	1.17	100.07	0
				20	1.52	1.52	1.52	0.01	10
				40	1.52	1.52	1.52	30.01	7
10%	10%	70%	10%	60	1.58	1.58	1.58	30.01	7
				80	1.54	1.54	1.54	50.04	5
				100	1.54	1.54	1.54	50.02	5
				20	1.15	1.15	1.15	40.00	6
				40	1.09	1.09	1.06	40.03	6
10%	10%	10%	70%	60	1.09	1.09	1.07	63.89	4
				80	1.07	1.06	1.06	80.03	2
				100	1.07	1.07	1.06	89.36	1

strategies described in Section 4.3. In particular, preliminary experiments to test the relevance of each component of TS showed that the absence of the second or third neighborhood dramatically decreases the performance of the algorithm. The absence of the first neighborhood, instead, is not dramatic since its moves can be performed by the second neighborhood. However, in this case the main effect is a considerable reduction of the speed of the algorithm. We also tried to use the algorithm as a greedy combination of local searches (i.e., with empty tabu lists) but the results were not competitive with those of TS.

The values we adopted for the parameters (see above) turned out to be robust with respect to all our computational experiments. In order to evaluate the sensitivity of the results with respect to changes of these values, we also tried to individually modify the used values by $\pm 20\%$. These experiments showed, e.g., that it is convenient to always have $\tau_1 < \tau_2$ and, in particular, the chosen value for τ_1 ($\tau_1 = 3$) has to be considered as an upper bound in order to guarantee an efficient exploration of the first neighborhood. Anyway, as usual in metaheuristics, the goal of tuning is to provide at each step a sufficiently effective intensification before any diversification. Hence, the more sensitive parameters are those providing the re-start actions and our suggestion, if a problem with different characteristics is given, is to modify the parameters in order to identify such border between intensification and diversification.

Chapter 5

A Unified Tabu Search for 2BP|*|*

5.1 Introduction

Informally[1] speaking, a *two-dimensional bin packing problem* (2BP) consists of allocating, without overlapping, a given set of "small" rectangles (*items*) to a minimum number of "large" identical rectangles (*bins*). In general, it is additionally required that the items are packed with their edges parallel to those of the bins. This basic problem has many real-world applications: cutting of standardized stock units in wood or glass industries, packing on shelves or truck beds in transportation and warehousing, paging of articles in newspapers, to mention just a few.

According to the specific application, several variants can arise, but in most cases the additional requirements are the following:

1) *Orientation.* The items may either have a fixed orientation or they can be rotated (by 90°).

2) *Guillotine cuts.* It may or not be imposed that the items are obtained through a sequence of edge-to-edge cuts parallel to the edges of the bin.

The guillotine constraint is frequently present in cutting problems, due to technological characteristics of automated cutting machines, whereas it is generally not imposed in packing applications. Rotation is not allowed in newspaper paging or when the items to be cut are decorated or corrugated, whereas orientation is free in the case of plain materials and in most packing contexts.

In this chapter we propose a simple typology for the class of bin packing problems defined by the four cases arising from the above two requirements. For each case, we present a new effective heuristic algorithm. We then introduce a unified metaheuristic framework for the whole class, which is adapted to a specific problem by simply changing the heuristic used to explore the neighborhood.

Formally, we have n items, and each item j is defined by a *width* w_j and a *height* h_j ($j = 1, \ldots, n$). An unlimited number of identical bins is available, each having width W and height H. We want to pack all items into the minimum number of bins, in such a way

[1]The results of this chapter appear in: A. Lodi, S. Martello, D. Vigo, "Heuristic and Meta-Heuristic Approaches for a Class of Two-Dimensional Bin Packing Problems", *INFORMS Journal on Computing* 11, 345–357, 1999, [109].

that no two items overlap and the edges of the items are parallel to those of the bins. We assume, without loss of generality, that all input data are positive integers. We consider four problems:

2BP|O|G: the items are oriented (O), i.e., they cannot be rotated, and guillotine cutting (G) is required;

2BP|R|G: the items may be rotated by 90° (R) and guillotine cutting is required;

2BP|O|F: the items are oriented and cutting is free (F);

2BP|R|F: the items may be rotated by 90° and cutting is free.

Although not all four variants are equally relevant in industrial contexts, a number of specific applications can be found in the bin packing or cutting stock literature. For example, 2BP|O|G has to be solved in the crepe-rubber cutting described by Schneider [143]; 2BP|R|G is related to the cutting stock problems arising in the steel industry, discussed by Vasko, Wolf and Stott [148]; 2BP|O|F models the problem of placing advertisements and other material in newspapers and yellow pages, studied by Lagus, Karanta and Yld-Jddski [102]; several applications of 2BP|R|F are mentioned by Bengtsson [21]. In the following, an asterisk will be used to denote both variants of a specific field. In order to ensure feasibility, we assume , without loss of generality, that:

i) $h_j \leq H$ and $w_j \leq W$ $(j = 1, \ldots, n)$ for 2BP|O|$*$;

ii) $\min\{w_j, h_j\} \leq \min\{W, H\}$ and $\max\{w_j, h_j\} \leq \max\{W, H\}$ $(j = 1, \ldots n)$ for 2BP|R|$*$.

In the well-known *one-dimensional bin packing problem* (1BP) one is required to partition n given elements having associated values $h_1, \ldots, h_n$ into the minimum number of subsets so that the sum of the values in each subset does not exceed a prefixed capacity H. Given any instance of this problem, consider the instance of the two-dimensional bin packing problem defined by n items j having height h_j and width $w_j = 1$ $(j = 1, \ldots, n)$, with bin height H and bin width $W = 1$. It is then clear that any of the four variants described above will solve the one-dimensional instance. It follows that our problems are strongly NP-hard, since it is known that the one-dimensional bin packing problem is such.

It is worth mentioning that most of the two-dimensional bin packing problems considered in the literature fall into the above cases. Theoretical contributions usually concern 2BP|O|F: Chung, Garey and Johnson [42], and Frenk and Galambos [76] have proposed upper bounds with asymptotic worst-case performance guarantee, whereas Martello and Vigo [119] have analyzed lower bounds and presented an exact branch-and-bound approach. The remaining literature is mostly devoted to applications and heuristics concerning one of the four cases. For 2BP|O|F, Berkey and Wang [23] have presented extensions of classical one-dimensional bin packing heuristics. Other heuristics have been proposed by Bengtsson [21] and El-Bouri, Popplewell, Balakrishnan and Alfa [65] for 2BP|R|F, by Lodi, Martello and Vigo [107] for 2BP|R|G, by Lodi, Martello and Vigo [108] for 2BP|O|G. The reader is also referred to Dyckhoff and Finke [63] and Dowsland and Dowsland [60] for general surveys on packing problems, and to Dyckhoff, Scheithauer and Terno [64] for an annotated bibliography.

We finally observe that Dyckhoff [62] has proposed a typology for cutting and packing problems, which is not however adequate in the present context, since all four problems we treat are classified there as 2/V/I/M.

In the next section we give basic definitions and briefly review relevant algorithms from the literature. In Section 5.3 we present a new heuristic algorithm for each of the four variants of 2BP. In Section 5.4 we introduce a general tabu search scheme which can virtually be used for the solution of any 2BP. The average performance of both heuristic and metaheuristic approaches is experimentally evaluated in Section 5.5.

5.2 Basic definitions and algorithms

Throughout this chapter, we denote an algorithm as A_{XY}, where A is the algorithm's name, whereas $X \in \{\mathrm{O}, \mathrm{R}\}$ and $Y \in \{\mathrm{G}, \mathrm{F}\}$ indicate the specific problem solved by A. It is worth mentioning that an algorithm for one of the four problems we consider may produce feasible solutions for other problems too. For example, an algorithm A_{OG} always produces solutions which are feasible for all four problems, whereas an algorithm A_{RF} may only be used for 2BP|R|F. The complete set of compatibilities between solutions and problems is depicted in the graph of Figure 5.1: an arc (v_i, v_j) implies that a solution feasible for the variant associated with vertex v_i is also feasible for the variant associated with vertex v_j.

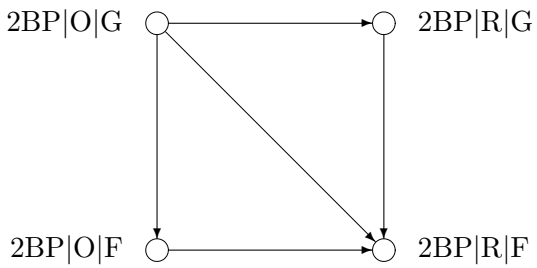


Figure 5.1: Compatibilities between solutions and problems.

Several heuristic algorithms from the literature obtain a feasible solution by packing the items in rows forming levels, called *shelves* (see Coffman, Garey, Johnson and Tarjan [43]). The first shelf of a bin coincides with its bottom; new shelves within the same bin are created along the horizontal line which coincides with the top of the tallest item packed on the highest shelf.

In many cases (see Chung, Garey and Johnson [42]) the solution is obtained in two phases, the first of which consists in solving an associated problem, known as the *Two-Dimensional Strip Packing Problem*. In this case one is given a single open-ended bin of width W and infinite height (*strip*), and the objective is to pack all the items such that the height to which the strip is filled is minimized. In the second phase the packed strip is subdivided into finite bins of height H. The strip packing is usually obtained through a shelf algorithm, so it is clear that the second phase consists in solving a one-dimensional bin packing problem in which each shelf is viewed as an element having value $\widetilde{h}_j$ $(j = 1, \ldots)$ equal to the height of the tallest item it contains.

The first approach of this type has been proposed and analyzed (from the worst-case behavior point of view) by Chung, Garey and Johnson [42]. Effective heuristics have then been derived by Berkey and Wang [23] for 2BP|O|G. Two of their algorithms, $\mathrm{FBS}_{\mathrm{OG}}$and $\mathrm{FFF}_{\mathrm{OG}}$, will be frequently referred to throughout the chapter, and are briefly described hereafter. Both initially sort the items by nonincreasing height.

Algorithm *Finite Best Strip* ($\mathrm{FBS_{OG}}$) starts by packing the strip according to a *best-fit decreasing* policy: if the current item does not fit into any existing shelf, then a new shelf is initialized for it; otherwise the current item is packed onto the shelf which minimizes the residual horizontal space. In the second phase the resulting shelves are packed into finite bins using the best-fit decreasing one-dimensional bin packing heuristic: the next (highest) shelf is packed into the bin which minimizes the residual vertical capacity, or into a new one, if no bin can accommodate it. The algorithm can be implemented so as to have time complexity $O(n \log n)$. As an example, consider the instance shown in Figure 5.2: the strip packing produced in the first phase is depicted in Figure 5.2 (i), and the corresponding finite bin solution is given in Figure 5.2 (ii).

Algorithm *Finite First Fit* ($\mathrm{FFF_{OG}}$) directly packs the items into finite bins (skipping the strip packing phase), according to a *first-fit decreasing* policy: the current item is packed into the first bin which can accommodate it, or on the bottom of a new one, if no such bin exists; in the former case the item is packed onto the first (lowest) existing shelf which can accommodate it, or by initializing a new one if no such shelf exists. The algorithm has time complexity $O(n^2)$. For the instance in Figure 5.2, algorithm $\mathrm{FFF_{OG}}$ produces the same solution as $\mathrm{FBS_{OG}}$.

It is easily seen that both algorithms above produce guillotine packings. A sequence of horizontal cuts separates the shelves. For each shelf, a vertical cut produces the leftmost item packed onto it, and a series of pairs (vertical cut, horizontal cut) produces the remaining items.

Lodi, Martello and Vigo [107] have considered adaptations of $\mathrm{FBS_{OG}}$ and $\mathrm{FFF_{OG}}$ to the case where rotation is allowed. We will say that an item is in *horizontal* (resp. *vertical*) orientation if its longest (resp. shortest) edge is parallel to the bottom of the bin or the strip. Both algorithms, denoted by $\mathrm{FBS_{RG}}$ and $\mathrm{FFF_{RG}}$ in the following, start by sorting the items according to nonincreasing value of their shortest edge, and by horizontally orienting them. Their iterative phase differs from that of $\mathrm{FBS_{OG}}$ and $\mathrm{FFF_{OG}}$ in two aspects: (a) when the current item initializes a new shelf, it is always packed horizontally, so as to minimize the height of the shelf; (b) when an existing shelf is considered for possible packing of the current item, if both orientations are feasible, then the vertical one is always considered, so as to favour future packings onto the same shelf. A different evolution of the two-phase shelf algorithms is the following approach proposed by Lodi, Martello and Vigo [107] for 2BP|R|G.

Algorithm *Floor-Ceiling* ($\mathrm{FC_{RG}}$) initially sorts the items by nonincreasing value of their shortest edge, and horizontally orients them. The main peculiarity is in the way the strip shelves are packed in the first phase. The algorithms described so far place the items, from left to right, with their bottom edge on the bottom line of the shelf (*floor*). Algorithm $\mathrm{FC_{RG}}$ may, in addition, place items, from right to left, with their top edge on the shelf *ceiling* (i.e., the horizontal line defined by the top edge of the tallest item packed in the shelf). Whenever an item is packed: (i) the rotated placing is also evaluated; (ii) the guillotine constraint is checked and the item placing is possibly modified accordingly. In the second phase, the shelves are packed into finite bins using an exact algorithm for the one-dimensional bin packing problem.

Note that the floor-ceiling approach allows for immediate adaptations to the three remaining problems:

$\mathrm{FC_{RF}}$: drop step (ii);

$\mathrm{FC_{OG}}$: initially sort the items according to nonincreasing height, and drop step (i);

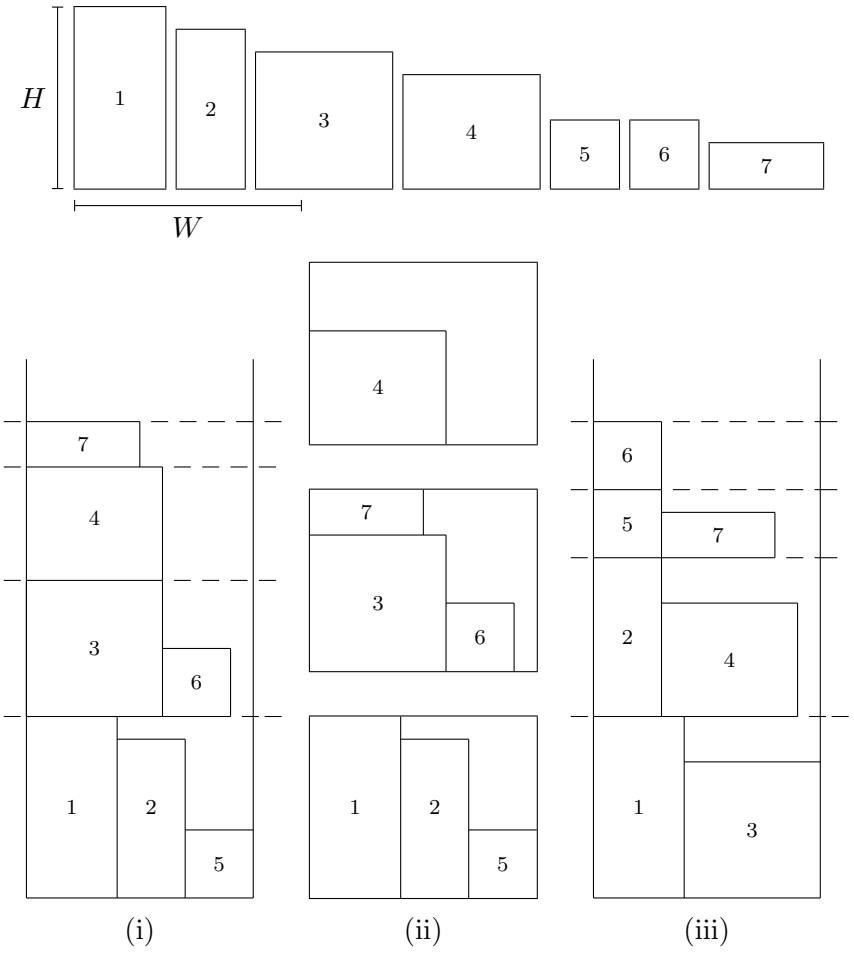


Figure 5.2: A two-dimensional bin packing instance with $n = 7$, and (i) strip packing produced by $\mathrm{FBS}_{\mathrm{OG}}$; (ii) finite bin solution found by $\mathrm{FBS}_{\mathrm{OG}}$ and $\mathrm{FFF}_{\mathrm{OG}}$; (iii) strip packing produced by $\mathrm{KP}_{\mathrm{OG}}$ (Section 5.3.1).

$\mathrm{FC}_{\mathrm{OF}}$: initially sort the items by nonincreasing height, and drop both steps (i) and (ii).

The computational experiments of Section 5.5.1 prove that the floor-ceiling algorithms outperform, on average, the other approaches from the literature. Better specific algorithms are introduced in the next sections.

We finally observe that the floor-ceiling algorithms above, as well as some of the new algorithms presented in the following sections, require the solution of NP-hard problems, hence their time complexity is non-polynomial. In practice, however, we always halted the execution of the codes for the NP-hard problems after a prefixed (small) number of iterations. The computational experiments showed that, in almost all cases, the optimal solution was obtained before the limit was reached.

5.3 New Heuristic Algorithms

In this section we introduce new heuristic algorithms for each of the four variants of the two-dimensional bin packing problem. The algorithms for 2BP$|*|$G are directly derived from the shelf approaches, by introducing a new way for packing the items on the shelves. The algorithms for 2BP$|*|$F are instead based on completely different ideas.

5.3.1 Oriented items, guillotine cutting (2BP|O|G)

In Section 5.2 we have seen algorithms FFF_{OG} and FBS_{OG} by Berkey and Wang [23], and the floor-ceiling algorithm FC_{OG}. In the present section we introduce a new two-phase algorithm in which the shelf packings are determined by solving a series of knapsack problems. In the 0-1 *knapsack problem* (KP01) one is given n elements, each having an associated profit p_j and cost c_j $(j = 1, \ldots, n)$, and a capacity q. The problem is to select a subset of elements whose total cost does not exceed q, and whose total profit is a maximum.

The proposed algorithm, denoted by KP_{OG}, at each iteration of the strip packing phase, initializes a new shelf with the tallest unpacked item, say j^*. The shelf packing is then completed by solving an instance of KP01 having an element for each unpacked item j, with profit $p_j = w_j h_j$ and cost $c_j = w_j$, and capacity $q = W - w_{j^*}$. The algorithm can be outlined as follows.

algorithm KP_{OG}:
sort the items according to nonincreasing h_j values;
comment: Phase 1;
repeat
open a new shelf in the strip by packing the first unpacked item;
solve the KP01 instance associated with the shelf;
pack the selected items onto the shelf
until all items are packed;
let $\widetilde{h}_1, \ldots, \widetilde{h}_s$ be the heights of the resulting shelves;
comment: Phase 2;
determine a finite bin solution by solving the 1BP instance having s elements,
with associated values $\widetilde{h}_1, \ldots, \widetilde{h}_s$, and capacity H
end.

Consider again the example in Figure 5.2: the strip packing produced by KP_{OG} is given in Figure 5.2 (iii); the resulting finite bin solution clearly packs the first two shelves in two bins, and the two remaining shelves in a third bin.

It is easily seen that the solutions produced by KP_{OG} always fulfil the guillotine constraint. As previously mentioned, an efficient implementation was obtained by halting the non-polynomial routines for the KP01 and 1BP instances after a prefixed (small) number of iterations.

5.3.2 Non-oriented items, guillotine cutting (2BP|R|G)

For this problem, we have already seen in Section 5.2 the adaptations FFF_{RG} and FBS_{RG} of the Berkey and Wang [23] algorithms, and the floor-ceiling algorithm FC_{RG} by Lodi,

Martello and Vigo [107]. We next describe a new effective algorithm, obtained by extending the knapsack-based approach introduced in the previous section.

The following variations to KP_{OG} are introduced in order to exploit the possibility of rotating the items. The items are initially sorted according to nonincreasing value of their shortest edge, and horizontally oriented: this orientation is always used for the shelf initializations (as in algorithm FC_{RG}).

For each shelf, say of height h^*, the instance of KP01 includes all unpacked items, either in vertical orientation, if no item size exceeds h^*, or in horizontal orientation, otherwise. Note that the profit associated with each item is independent of the orientation, while the cost is reduced whenever the vertical orientation is chosen. Therefore, this strategy ensures that the optimal solution value of the resulting KP01 instance, i.e., the total area packed in the strip, is maximum over the feasible item orientations.

Once a feasible finite bin solution has been obtained from the resulting shelves, as in KP_{OG}, an alternative solution is determined as follows. An instance of 2BP|O|G is built, which has a pseudo-item for each of the previous shelves, with sizes given by the height of the shelf and by the horizontal space actually used in it; for this pseudo-item, the vertical orientation is chosen whenever possible, i.e., if its longest edge does not exceed the finite bin height. Algorithm KP_{OG} is then executed for the resulting instance, and the best of the two final solutions is selected. The overall algorithm follows.

algorithm KP_{RG}:
sort the items by nonincreasing $\min\{w_j, h_j\}$ values, and horizontally orient them;
comment: Phase 1;
repeat
open a new shelf in the strip by horizontally packing the first unpacked item;
appropriately orient each unpacked item;
solve the resulting KP01 instance, and pack the selected items
until all items are packed;
let $\widetilde{w}_i$ and $\widetilde{h}_i$ $(i = 1, \ldots, s)$ be the sizes of the s resulting shelves;
comment: Phase 2;
determine a finite bin solution by solving the associated 1BP instance;
let z_1 be the solution value obtained for the 2BP|R|G instance;
comment: Phase 3;
define s pseudo-items having sizes $\widetilde{w}_i$, $\widetilde{h}_i$ $(i = 1, \ldots, s)$;
vertically orient each pseudo-item i such that $\max\{\widetilde{w}_i, \widetilde{h}_i\} \leq H$;
execute algorithm KP_{OG} for the 2BP|O|G instance induced by the pseudo-items;
let z_2 be the resulting solution value, and set $z := \min\{z_1, z_2\}$
end.

Once again consider the example in Figure 5.2, but interpret it as an instance of 2BP|R|G. The items are sorted by KP_{RG} as (3, 4, 1, 2, 5, 6, 7), with items 1 and 2 rotated by 90°. Figure 5.3 shows the solution found by each phase of KP_{RG}. Note that the solution at the end of Phase 2 still requires three bins, while the rotation of pseudo-items in Phase 3 produces an optimal two-bin solution.

As to the time complexity of KP_{RG}, the same considerations made for KP_{OG} obviously apply.

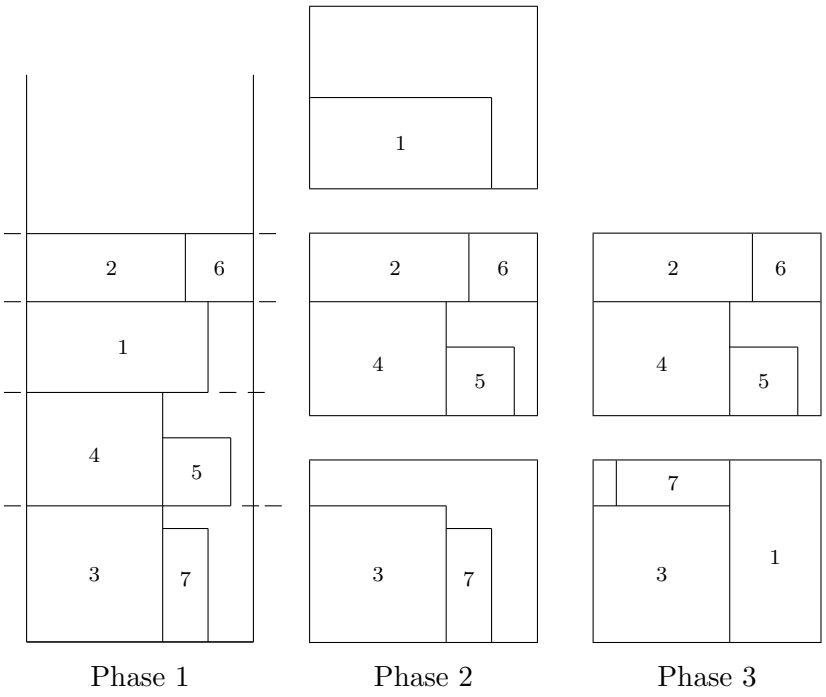


Figure 5.3: Solutions produced by Algorithm $\mathrm{KP}_{\mathrm{RG}}$.

5.3.3 Oriented items, free cutting (2BP|O|F)

To our knowledge, no specific heuristic for this case has been presented in the literature. In Section 5.2 we introduced the extension, $\mathrm{FC}_{\mathrm{OF}}$, of the floor-ceiling approach. In this section we propose a new effective heuristic, which no longer uses shelves, but exploits the feasibility of non-guillotine patterns by packing the items into the bins in alternate directions, i.e., from left to right, then from right to left in the lowest possible position, and so on.

More precisely, the *Alternate Directions* ($\mathrm{AD}_{\mathrm{OF}}$) algorithm starts by sorting the items according to nonincreasing heights, and by computing a lower bound L on the optimal solution value. (A trivial bound is of course the *continuous* lower bound $L_0 = \lceil \sum_{j=1}^{n} w_j h_j/(WH) \rceil$; better bounds can be found in Martello and Vigo [119].) Then, L bins are initialized by packing on their bottoms a subset of the items, following a best-fit decreasing policy. The remaining items are packed into *bands* according to the current *direction* associated with the bin. Namely, if the direction is "from left to right" (resp. "from right to left"): (i) the first item of the band is packed with its left (resp. right) edge touching the left (resp. right) edge of the bin, in the lowest possible position; (ii) each subsequent item is packed with its left (resp. right) edge touching the right (resp. left) edge of the previous item in the band, in the lowest possible position. Observe that the items packed (from left to right) by the initialization step constitute the first band of each bin. The direction associated with all the bins after this step is then "from right to left". In the iterative part of the algorithm, for each bin, we scan all the unpacked items, possibly packing them in the current direction, and changing direction when no further item can be packed in the current band. As soon as no item can be packed in either direction, we move to the next bin, or we initialize a new empty bin. The algorithm

can be formally stated as follows.

algorithm $\mathrm{AD_{OF}}$:
sort the items according to nonincreasing h_j values;
comment: Phase 1;
compute a lower bound L on the optimal solution value, and open L empty bins;
for $j := 1$ **to** n **do**
if item j can be packed on the bottom of some bin **then**
pack j, left justified, on the bottom of the bin whose residual horizontal space is a minimum;
comment: Phase 2;
$i := 0$;
repeat
$i := i + 1$, *right_to_left* := `true`, *n_fail* := 0;
repeat
let j be the first unpacked item which can be packed in bin i according to the current value of *right_to_left*, if any;
if $j =$ `nil` **then**
n_fail := *n_fail* + 1, *right_to_left* := **not** *right_to_left*
else
pack j into bin i according to *right_to_left*, *n_fail* := 0;
until *n_fail* = 2
until all items are packed
end.

As an example, consider the instance of 12 items shown in Figure 5.4 (i). The bin sizes are $W = 10$ and $H = 8$, and the item sizes are: $w_1 = 4$, $h_1 = 6$; $w_2 = h_2 = 4$; $w_3 = 8$, $h_3 = 3$; $w_4 = w_5 = w_6 = 4$, $h_4 = h_5 = h_6 = 3$; $w_7 = 1$, $h_7 = 3$; $w_8 = 6$, $h_8 = 2$; $w_9 = 2$, $h_9 = 2$; $w_{10} = w_{11} = 9$, $h_{10} = h_{11} = 2$; $w_{12} = 3$, $h_{12} = 1$. Lower bound L_0 gives value 2. Figure 5.4 (ii) shows the solution found by $\mathrm{AD_{OF}}$. In Phase 1, items 1, 2, 3, 7 and 9 are packed. In Phase 2, a band is added to the first bin (items $\{4, 8\}$), and two to the second one (items $\{5, 6\}$ and $\{10\}$). The third bin is then opened, and packs the remaining items, in two bands. Note that the pattern obtained in the first bin is not guillotine cuttable.

Algorithm $\mathrm{AD_{OF}}$ can be implemented so as to run in $O(n^3)$ time. Indeed, the heaviest of the lower bounds described by Martello and Vigo [119] requires $O(n^3)$ time. The main repeat-until loop considers $O(n)$ bins, and, for each of them, $O(n)$ items are examined in the inner loop. For each item, its possible packing can be evaluated in $O(n)$ time, since: (i) the horizontal coordinate at which the item can be placed is unique (see above); (ii) the lowest possible vertical coordinate can be determined in $O(n)$ time by checking the interaction of the item with at most $O(n)$ items currently packed in the bin.

5.3.4 Non-oriented items, free cutting (2BP|R|F)

We have already seen in Section 5.2 the adaptation, $\mathrm{FC_{RF}}$, of the floor-ceiling approach. In addition, algorithm $\mathrm{AD_{OF}}$ of the previous section can easily be adapted to exploit the possibility of rotating the items. The best computational results were however obtained by the following completely different approach.

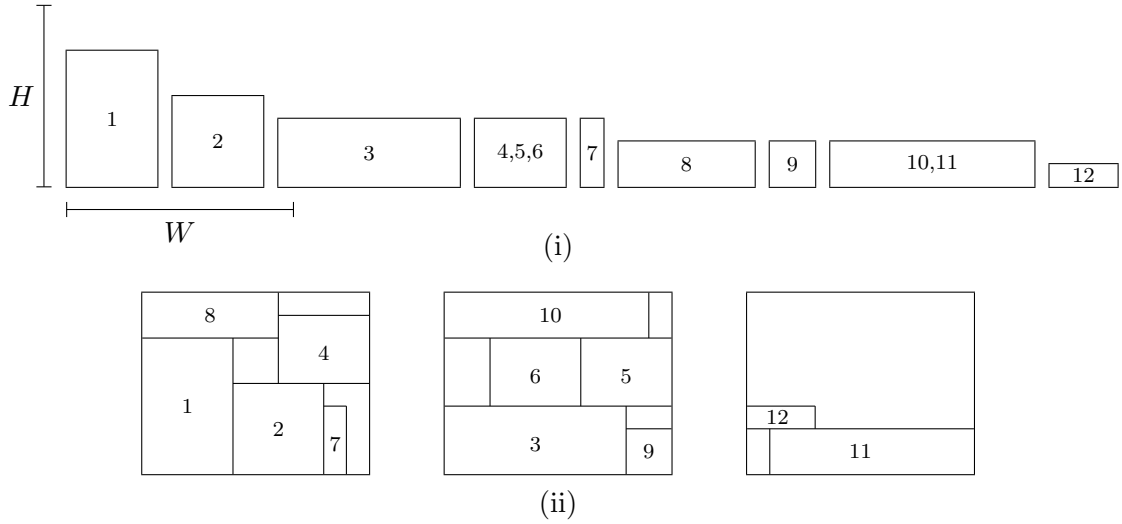


Figure 5.4: (i) two-dimensional bin packing instance with $n = 12$; (ii) solution found by Algorithm $\mathrm{AD_{OF}}$.

The algorithm, called *Touching Perimeter* ($\mathrm{TP_{RF}}$), starts by sorting the items according to nonincreasing area (breaking ties by nonincreasing $\min\{w_j, h_j\}$ values), and by horizontally orienting them. A lower bound L on the optimal solution value is then computed, and L empty bins are initialized. (The continuous lower bound L_0 defined in the previous section is obviously valid for 2BP|R|F as well; better bounds are proposed by Dell'Amico, Martello and Vigo [56].) The algorithm packs one item at a time, either in an existing bin, or by initializing a new one. The first item packed in a bin is always placed in the bottom-left corner. Each subsequent item is packed in a so-called *normal position* (see Christofides and Whitlock [41]), i.e., with its bottom edge touching either the bottom of the bin or the top edge of another item, and with its left edge touching either the left edge of the bin or the right edge of another item.

The choice of the bin and of the packing position is done by evaluating a *score*, defined as the percentage of the item perimeter which touches the bin and other items already packed. This strategy favours patterns where the packed items do not "trap" small areas, which may be hard to use for further placements. For each candidate packing position, the score is evaluated twice, for the two item orientations (if both are feasible), and the highest value is selected. Score ties are broken by choosing the bin having the maximum packed area. The overall algorithm is as follows.

algorithm $\mathrm{TP_{RF}}$:
sort the items by nonincreasing $w_j h_j$ values, and horizontally orient them;
comment: Phase 1;
compute a lower bound L on the optimal solution value, and open L empty bins;
comment: Phase 2;
for $j := 1$ **to** n **do**
$score := 0$;
for each normal packing position in an open bin **do**
let $score_1$ and $score_2$ be the scores associated with the two orientations;

```
            score := max{score, score_1, score_2}
        end for;
        if score > 0 then
            pack item j in the bin, position and orientation corresponding to score
        else
            open a new bin, and horizontally pack item j into it
    end for
end.
```

Consider again the example in Figure 5.4, but interpret it as an instance of 2BP|R|F. The items are sorted by $\mathrm{TP}_{\mathrm{RF}}$ as (1, 3, 10, 11, 2, 4, 5, 6, 8, 9, 7, 12), with items 1 and 7 rotated by 90°. Figure 5.5 shows the solution found by $\mathrm{TP}_{\mathrm{RF}}$. Note that the pattern in the second bin is not guillotine cuttable.

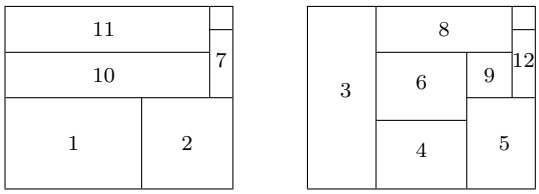


Figure 5.5: Solution found by Algorithm $\mathrm{TP}_{\mathrm{RF}}$.

The iterative part of the algorithm requires $O(n^3)$ time since, for each item, the number of normal positions is $O(n)$ and, for each position, the touching perimeter is computed in $O(n)$ time.

5.4 A Unified Tabu Search Framework

A metaheuristic approach is an attractive way for guiding the operations of a subordinate heuristic in order to obtain high-quality solutions to a difficult combinatorial optimization problem. Among the different metaheuristic techniques, tabu search recently proved to be particularly effective for two-dimensional bin packing problems (see, e.g., Lodi, Martello and Vigo [107, 108]). The reader is referred to Aarts and Lenstra [2] and to Glover and Laguna [82] for an introduction to metaheuristic and tabu search algorithms. In this section we introduce a general tabu search scheme which can virtually be used for the solution of any two-dimensional finite bin packing problem by simply changing the subordinate inner heuristic.

The main feature of our approach is the use of a unified parametric neighborhood, whose size and structure are dynamically varied during the search. The scheme and the neighborhood are independent of the specific problem to be solved. The peculiarities of the problem are taken into account only in the choice of a specific deterministic algorithm to be used, within the neighborhood search, for the evaluation of the moves. In the following, we will denote by A such an algorithm, and by $A(S)$ the solution value it produces when applied to the (sub)instance of 2BP induced by item set S. In Section 5.5.2, the effectiveness of the proposed scheme is computationally evaluated, on the four packing problems we are considering, using for A the four heuristics introduced in the previous section, and algorithms $\mathrm{FBS}_{\mathrm{OG}}$ and $\mathrm{FBS}_{\mathrm{RG}}$ described in Section 5.2.

Given a current solution, the neighborhood is searched through *moves* which consist in modifying the solution by changing the packing of a subset of items S, in an attempt to empty a specific *target bin*. To this end, subset S always includes one item, j, from the target bin and the current contents of k other bins. The new packing for S is obtained by executing algorithm A on S. In this way, parameter k defines the size and the structure of the current neighborhood. Its value is automatically increased or decreased during the search, and the algorithm maintains k distinct tabu lists. The target bin is selected as the one which is more likely to be emptied by the moves. We adopted the same policy used in the specialized tabu search algorithms presented by Lodi, Martello and Vigo [107, 108], i.e., the target bin t is determined as the one minimizing, over all current bins i, the *filling function*

$$\varphi(S_i) = \alpha \frac{\sum_{j \in S_i} w_j h_j}{WH} - \frac{|S_i|}{n} \tag{5.1}$$

where S_i denotes the set of items currently packed into bin i, and α is a pre-specified positive weight (equal to 20 in our computational experiments). The resulting choice favours the selection of target bins packing a small area, breaking ties by bins packing a relatively large number of items.

The overall algorithm, 2BP_TABU, is formally stated below. An initial incumbent solution is obtained by executing algorithm A on the complete instance, while the initial tabu search solution consists of packing one item per bin. At each iteration, a target bin is selected, and a sequence of moves, each performed within a procedure SEARCH, tries to empty it. Procedure SEARCH also updates the value of parameter k and, in special situations, may impose performing *diversification actions* (to be discussed later). The execution is halted as soon as a proven optimal solution is found, or a time limit is reached.

algorithm 2BP_TABU:
  $z^* := A(\{1, \ldots, n\})$ (**comment:** incumbent solution value);
  compute a lower bound L on the optimal solution value;
  if $z^* = L$ **then stop**;
  initialize all tabu lists to empty;
  pack each item into a separate bin;
  $z := n$ (**comment:** tabu search solution value);
  while time limit is not reached **do**
    determine the target bin t;
    diversify := `false`; $k := 1$;
    while *diversify* = `false` **and** $z^* > L$ **do**
      $k_{in} := k$;
      call SEARCH(t,k,*diversify*,z);
      $z^* := \min\{z^*, z\}$;
      if $k \leq k_{in}$ **then** determine the new target bin t
    end while;
    if $z^* = L$ **then stop else** perform a diversification action
  end while
end.

Given the target bin t, procedure SEARCH explores the neighborhood defined by the current value of parameter k. For each item j in bin t we evaluate the candidate moves by

executing algorithm A on the subinstances induced by all possible sets S defined by j and by all k-tuples of other bins. In two special situations the move is immediately performed, and the control returns to the main algorithm: (i) when a move decreases the current number of used bins; (ii) when a non-tabu move removes j from t by packing the subinstance in exactly k bins. In addition, in these cases, the neighborhood is changed by decreasing the current value of parameter k by one unit.

When neither (i) nor (ii) apply, a *penalty* is associated with the move. The penalty is infinity if the move is tabu, or if algorithm A used at least two extra bins (i.e., $A(S) > k+1$), or if $k = 1$. Otherwise, the penalty is computed as follows. We determine, among the $k+1$ bins produced by A, the one, say $\bar{t}$, which minimizes the filling function, and execute algorithm A on the subinstance induced by the items in bin $\bar{t}$ plus the residual items in the target bin. If a single bin solution is obtained, the penalty of the overall move is set to the minimum among the filling function values computed for the $k+1$ resulting bins; otherwise, the penalty is set to infinity.

When the neighborhood has been entirely searched without detecting case (i) or (ii) above, the move having the minimum finite penalty (if any) is performed and the control returns to 2BP_TABU. If, instead, the minimum penalty is infinity, i.e., no acceptable move has been found, the neighborhood is changed by increasing the current value of parameter k by one unit, or, if k already reached a maximum prefixed value $k_{\max}$, by imposing a diversification action. The overall procedure can be outlined as follows.

procedure SEARCH(t,k,*diversify*,z);
  $penalty^* := +\infty$;
  for each $j \in S_t$ **do**
    for each k-tuple K of bins not including t **do**
      $S := \{j\} \cup (\bigcup_{i \in K} S_i)$;
      $penalty := +\infty$;
      case
        $A(S) < k$:
          execute the move and update the current solution value z;
          $k := \max\{1, k-1\}$;
          return;
        $A(S) = k$:
          if the move is not tabu **or** $S_t \equiv \{j\}$ **then**
            execute the move and update the current solution value z;
            if $S_t \equiv \{j\}$ **then** $k := \max\{1, k-1\}$;
            return
          end if;
        $A(S) = k+1$ **and** $k > 1$:
          let I be the set of $k+1$ bins used by A;
          $\bar{t} := \arg\min_{i \in I}\{\varphi(S_i)\}$, $T := (S_t \setminus \{j\}) \cup S_{\bar{t}}$;
          if $A(T) = 1$ **and** the move is not tabu **then**
            $penalty := \min\{\varphi(T), \min_{i \in I \setminus \{\bar{t}\}}\{\varphi(S_i)\}\}$
      end case;
      $penalty^* := \min\{penalty^*, penalty\}$;
    end for;
  end for;

if *penalty** $\neq +\infty$ **then** execute the move corresponding to *penalty**
else if $k = k_{\max}$ **then** $diversify :=$ `true` **else** $k := k + 1$
return.

The diversification actions and the tabu lists remain to be described. The algorithm performs two kinds of diversification, controlled by a counter d, initially set to one. Whenever a diversification is imposed, d is increased by one and the target bin is determined as the one having the d-th smallest value of the filling function. If however $d > z$ or $d = d_{\max}$ (where $d_{\max}$ is a prefixed limit) a stronger diversification is performed, namely: (i) the $\lfloor z/2 \rfloor$ bins with smaller value of the filling function are removed from the tabu search solution; (ii) a new solution is obtained by packing alone in a separate bin each item currently packed into a removed bin; (iii) all tabu lists are reset to empty, and the value of d is reset to one.

As previously mentioned, there are a tabu list and a tabu tenure τ_k ($k = 1, \ldots, k_{\max}$) for each neighborhood. For $k > 1$, each list stores the *penalty** values corresponding to the last τ_k moves performed, in the corresponding neighborhood, on exit from the two for-each loops of SEARCH. For $k = 1$ instead, since the first and third case of SEARCH can never occur, the moves are only performed when the second case occurs, i.e., without computing a penalty. Hence, the tabu list stores the values of the filling function, $\varphi(S)$, corresponding to the last τ_1 sets S for which a move has been performed. The attributes stored in the tabu lists are real values, considerably varying with the solutions, see (5.1), so this choice is likely to prevent short-term cycling through moves.

5.5 Computational Experiments

All the algorithms presented in Section 5.3 and the tabu search framework were coded in FORTRAN 77 and run on a Silicon Graphics INDY R10000sc 195Mhz on test instances from the literature. The solutions of the 1BP and KP01 instances, needed by some of the deterministic algorithms, were obtained, respectively, through FORTRAN codes MTP (with a limit of 300 backtrackings) and MT1 (with a limit of 500 backtrackings) included in the diskette accompanying the book by Martello and Toth [118].

We considered ten classes of randomly generated problems. The first six classes have been proposed by Berkey and Wang [23]:

Class 5 : w_j and h_j uniformly random in [1,10], $W = H = 10$;
Class 6 : w_j and h_j uniformly random in [1,10], $W = H = 30$;
Class 7 : w_j and h_j uniformly random in [1,35], $W = H = 40$;
Class 8 : w_j and h_j uniformly random in [1,35], $W = H = 100$;
Class 9 : w_j and h_j uniformly random in [1,100], $W = H = 100$;
Class 10 : w_j and h_j uniformly random in [1,100], $W = H = 300$.

In each of the above classes, all the item sizes are generated in the same interval. Martello and Vigo [119] have proposed the following classes, where a more realistic situation is considered. The items are classified into four types:

Type 1 : w_j uniformly random in $[\frac{2}{3}W, W]$, h_j uniformly random in $[1, \frac{1}{2}H]$;

Type 2 : w_j uniformly random in $[1, \frac{1}{2}W]$, h_j uniformly random in $[\frac{2}{3}H, H]$;

Type 3 : w_j uniformly random in $[\frac{1}{2}W, W]$, h_j uniformly random in $[\frac{1}{2}H, H]$;

Type 4 : w_j uniformly random in $[1, \frac{1}{2}W]$, h_j uniformly random in $[1, \frac{1}{2}H]$.

The bin sizes are $W = H = 100$ for all classes, while the items are as follows:

Class 1 :type 1 with probability 70%, type 2, 3, 4 with probability 10% each;

Class 2 :type 2 with probability 70%, type 1, 3, 4 with probability 10% each;

Class 3 :type 3 with probability 70%, type 1, 2, 4 with probability 10% each;

Class 4 :type 4 with probability 70%, type 1, 2, 3 with probability 10% each.

For each class[2], we considered five values of n: 20, 40, 60, 80, 100. For each class and value of n, ten instances were generated. In the following two sections we present the computational results obtained for the deterministic algorithms and for the tabu search approach.

Worth is mentioning that test instances where the items sizes are drawn from uniform distributions are usually quite easy to solve for the one-dimensional bin packing problem (see, e.g., Martello and Toth [118], Ch. 8). This is however not true for the two-dimensional case. For example, an effective branch-and-bound algorithm for 2BP|O|F (see Martello and Vigo [119]) was not able to solve about one third of the 500 instances described above. Even small size instances are far from being trivial: fifteen percent of the instances with $n = 40$ were not solved to optimality.

5.5.1 Results for deterministic algorithms

The results are presented in Tables 5.1 and 5.2. The first two columns give the class and the value of n. The next two pairs of columns refer to the algorithms by Berkey and Wang [23], and give the results for 2BP|O|$*$ and 2BP|R|$*$, respectively: the algorithms are the original ones in [23] (FFF_{OG} and FBS_{OG}) and the variants allowing item rotation described in Section 5.2 (FFF_{RG} and FBS_{RG}). The following four pairs of columns refer to the four problems considered: for each problem we give the results obtained by the corresponding variant of the floor-ceiling approach described in Section 5.2 and by the specific heuristic in Section 5.3.

For each algorithm, the entries report the average ratio (heuristic solution value)/(lower bound), computed over the ten generated instances. For 2BP|O|$*$ we used the lower bound by Martello and Vigo [119], and for 2BP|R|$*$ the bound by Dell'Amico, Martello and Vigo [56]. For each class, the final line gives the average over all values of n. We do not give the CPU times, as they are negligible (never exceeding 0.5 seconds).

By considering, for each class, the average values computed over all values of n, we see that the new algorithms proposed in the present chapter outperform, in general, all the other heuristics from the literature. For all cases where this does not happen, the best is the floor-ceiling approach. The most difficult problems, with average errors exceeding 10%, turn out to belong to Classes 7, 9 and 10 for all problem types, to Classes 1 and 2 for 2BP|R|$*$, and to Class 4 for 2BP|O|$*$.

[2]Due to historical reason Classes 5-10 are considered before Classes 1-4, despite to the numeration which is according to Chapter 2.

Table 5.1: Deterministic algorithms: (heuristic solution value)/(lower bound). Random problem instances proposed by Berkey and Wang.

		Berkey and Wang				Lodi, Martello and Vigo							
		2BP|O|∗		2BP|R|∗		2BP|O|G		2BP|R|G		2BP|O|F		2BP|R|F	
Class	n	$\frac{FFF_{\mathrm{OG}}}{LB}$	$\frac{FBS_{\mathrm{OG}}}{LB}$	$\frac{FFF_{\mathrm{RG}}}{LB}$	$\frac{FBS_{\mathrm{RG}}}{LB}$	$\frac{FC_{\mathrm{OG}}}{LB}$	$\frac{KP_{\mathrm{OG}}}{LB}$	$\frac{FC_{\mathrm{RG}}}{LB}$	$\frac{KP_{\mathrm{RG}}}{LB}$	$\frac{FC_{\mathrm{OF}}}{LB}$	$\frac{AD_{\mathrm{OF}}}{LB}$	$\frac{FC_{\mathrm{RF}}}{LB}$	$\frac{TP_{\mathrm{RF}}}{LB}$
	20	1.17	1.14	1.09	1.06	1.14	1.13	1.06	1.06	1.12	1.12	1.06	1.05
	40	1.12	1.09	1.10	1.08	1.09	1.10	1.08	1.07	1.08	1.09	1.08	1.06
5	60	1.10	1.07	1.12	1.09	1.07	1.07	1.09	1.07	1.07	1.07	1.09	1.05
	80	1.08	1.06	1.10	1.09	1.06	1.06	1.09	1.08	1.06	1.06	1.09	1.06
	100	1.07	1.06	1.08	1.08	1.06	1.05	1.07	1.05	1.06	1.05	1.07	1.03
Average		1.108	1.084	1.098	1.080	1.084	1.082	1.078	1.066	1.078	1.078	1.078	1.050
	20	1.10	1.10	1.00	1.00	1.10	1.00	1.00	1.00	1.10	1.00	1.00	1.00
	40	1.10	1.10	1.10	1.10	1.10	1.10	1.10	1.10	1.10	1.10	1.10	1.10
6	60	1.15	1.15	1.10	1.10	1.15	1.15	1.05	1.15	1.10	1.10	1.05	1.00
	80	1.07	1.07	1.07	1.07	1.07	1.07	1.03	1.07	1.07	1.07	1.03	1.07
	100	1.06	1.06	1.03	1.06	1.03	1.03	1.03	1.03	1.03	1.03	1.03	1.00
Average		1.096	1.096	1.060	1.066	1.090	1.070	1.042	1.070	1.080	1.060	1.042	1.034
	20	1.20	1.18	1.20	1.20	1.18	1.18	1.18	1.12	1.18	1.20	1.18	1.06
	40	1.18	1.14	1.21	1.16	1.14	1.15	1.16	1.16	1.14	1.15	1.16	1.11
7	60	1.14	1.11	1.20	1.18	1.11	1.12	1.19	1.12	1.11	1.13	1.19	1.11
	80	1.13	1.10	1.20	1.15	1.10	1.10	1.15	1.12	1.10	1.10	1.15	1.10
	100	1.12	1.09	1.16	1.14	1.09	1.09	1.13	1.10	1.09	1.09	1.13	1.08
Average		1.154	1.124	1.194	1.166	1.124	1.128	1.162	1.124	1.124	1.134	1.162	1.092
	20	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	40	1.10	1.10	1.10	1.10	1.00	1.10	1.00	1.00	1.00	1.00	1.00	1.00
8	60	1.20	1.20	1.10	1.10	1.10	1.20	1.10	1.10	1.10	1.15	1.10	1.10
	80	1.10	1.10	1.10	1.10	1.10	1.13	1.10	1.10	1.10	1.10	1.10	1.07
	100	1.10	1.10	1.07	1.07	1.10	1.10	1.07	1.07	1.10	1.03	1.07	1.03
Average		1.100	1.100	1.074	1.074	1.060	1.106	1.054	1.054	1.060	1.056	1.054	1.040
	20	1.14	1.14	1.08	1.08	1.14	1.13	1.08	1.08	1.14	1.14	1.08	1.06
	40	1.11	1.11	1.14	1.12	1.11	1.09	1.10	1.11	1.11	1.11	1.10	1.11
9	60	1.11	1.10	1.13	1.11	1.10	1.10	1.11	1.11	1.10	1.10	1.11	1.08
	80	1.12	1.09	1.13	1.10	1.09	1.09	1.11	1.10	1.09	1.09	1.11	1.08
	100	1.12	1.09	1.13	1.10	1.09	1.09	1.10	1.09	1.09	1.09	1.10	1.08
Average		1.120	1.106	1.122	1.102	1.106	1.100	1.100	1.098	1.106	1.106	1.100	1.082
	20	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	40	1.40	1.40	1.40	1.40	1.40	1.50	1.40	1.40	1.40	1.40	1.40	1.40
10	60	1.10	1.10	1.05	1.05	1.10	1.10	1.05	1.05	1.10	1.05	1.05	1.05
	80	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	100	1.13	1.10	1.10	1.07	1.10	1.10	1.07	1.10	1.10	1.07	1.07	1.07
Average		1.126	1.120	1.110	1.104	1.120	1.140	1.104	1.110	1.120	1.104	1.104	1.104

As to the percentage errors, it is also noteworthy that they are computed with respect to a lower bound value, hence they give a pessimistic estimate of the actual quality of the heuristic solution. According to our experience, the gap between lower bound and exact solution can be quite large. Consider for example the instances of 2BP|O|F, for which the branch-and-bound algorithm by Martello and Vigo [119] is available: the average errors computed by using the best available solution (not always optimal) are 1.035, 1.050, 1.005, 1.034, 1.040, 1.000, 1.062, 1.000, 1.049, and 1.000, respectively for Classes 1 to 10, i.e., about one third, on average, of those reported in the tables.

Table 5.2: Deterministic algorithms: (heuristic solution value)/(lower bound). Random problem instances proposed by Martello and Vigo.

		Berkey and Wang				Lodi, Martello and Vigo							
		2BP|O|∗		2BP|R|∗		2BP|O|G		2BP|R|G		2BP|O|F		2BP|R|F	
Class	n	$\frac{FFF_{OG}}{LB}$	$\frac{FBS_{OG}}{LB}$	$\frac{FFF_{RG}}{LB}$	$\frac{FBS_{RG}}{LB}$	$\frac{FC_{OG}}{LB}$	$\frac{KP_{OG}}{LB}$	$\frac{FC_{RG}}{LB}$	$\frac{KP_{RG}}{LB}$	$\frac{FC_{OF}}{LB}$	$\frac{AD_{OF}}{LB}$	$\frac{FC_{RF}}{LB}$	$\frac{TP_{RF}}{LB}$
	20	1.10	1.10	1.19	1.19	1.10	1.10	1.19	1.17	1.08	1.10	1.19	1.13
	40	1.11	1.11	1.17	1.17	1.11	1.07	1.17	1.17	1.09	1.10	1.17	1.10
1	60	1.08	1.08	1.18	1.18	1.08	1.06	1.18	1.16	1.07	1.07	1.18	1.12
	80	1.07	1.06	1.19	1.17	1.06	1.06	1.17	1.17	1.06	1.06	1.17	1.11
	100	1.04	1.04	1.17	1.17	1.04	1.04	1.17	1.16	1.04	1.04	1.17	1.11
Average		1.080	1.078	1.180	1.176	1.078	1.066	1.176	1.166	1.068	1.074	1.176	1.114
	20	1.17	1.16	1.18	1.16	1.16	1.12	1.16	1.16	1.16	1.13	1.16	1.16
	40	1.09	1.08	1.19	1.19	1.08	1.07	1.19	1.19	1.07	1.08	1.19	1.16
2	60	1.06	1.06	1.18	1.18	1.06	1.06	1.18	1.18	1.06	1.06	1.18	1.11
	80	1.07	1.06	1.17	1.16	1.06	1.05	1.16	1.15	1.06	1.06	1.16	1.11
	100	1.06	1.06	1.17	1.17	1.06	1.04	1.17	1.17	1.06	1.06	1.17	1.12
Average		1.090	1.084	1.178	1.172	1.084	1.068	1.172	1.170	1.082	1.078	1.172	1.132
	20	1.01	1.01	1.00	1.00	1.01	1.01	1.00	1.00	1.01	1.01	1.00	1.01
	40	1.02	1.02	1.01	1.01	1.02	1.02	1.01	1.01	1.02	1.02	1.01	1.02
3	60	1.02	1.02	1.01	1.01	1.02	1.01	1.01	1.01	1.02	1.02	1.01	1.01
	80	1.02	1.02	1.01	1.01	1.02	1.02	1.01	1.01	1.02	1.02	1.01	1.01
	100	1.02	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01
Average		1.018	1.016	1.008	1.008	1.016	1.014	1.008	1.008	1.016	1.016	1.008	1.012
	20	1.14	1.14	1.15	1.15	1.14	1.16	1.15	1.12	1.14	1.10	1.15	1.20
	40	1.14	1.09	1.09	1.09	1.09	1.10	1.09	1.09	1.09	1.09	1.09	1.08
4	60	1.15	1.12	1.11	1.09	1.10	1.10	1.09	1.08	1.08	1.11	1.09	1.09
	80	1.15	1.13	1.09	1.07	1.12	1.12	1.06	1.06	1.11	1.10	1.06	1.06
	100	1.14	1.10	1.07	1.07	1.10	1.08	1.07	1.05	1.09	1.10	1.07	1.06
Average		1.144	1.116	1.102	1.094	1.110	1.112	1.092	1.080	1.102	1.100	1.092	1.098

The use of lower bounds in the algorithms' evaluation needs an additional observation. Since the same instances are solved with different constraints, one could expect a dominance between the algorithms. Indeed, an algorithm for a less constrained variant produces, on average, better solutions in terms of number of bins. This does not always appear in the average ratios reported in the tables, due to the fact that we use different lower bounds for the four problems. The phenomenon is more evident if one considers the absolute solution values: for example, the total number of bins (over the 500 solved instances) is 7480, 7297, 7487 and 7184 for KP_{OG}, KP_{RG}, AD_{OF} and TP_{RF}, respectively. The slight anomaly in the

behavior KP_{OG} and AD_{OF} disappears when the algorithms are used within tabu search: the total number of bins becomes then 7433, 7101, 7373 and 7100, respectively.

We finally note that, quite surprisingly, the average ratios for algorithms FC_{RG} and FC_{RF} are identical. Indeed, they always produced solutions using the same number of bins. However, in 164 out of 500 instances the packing patterns were actually different, and those produced by FC_{RF} were not guillotine cuttable.

5.5.2 Results for tabu search

The tabu search approach was implemented in a straightforward way from its description in Section 5.4, i.e., with no additional tailoring or adaptation to the particular problem variant, but the call to the specific inner heuristic algorithm (denoted by A in the pseudo-code). The resulting code is quite compact, consisting of just 500 FORTRAN statements. Tables 5.3 and 5.4 present the results it gives, for the four problem variants, when different inner heuristics are invoked.

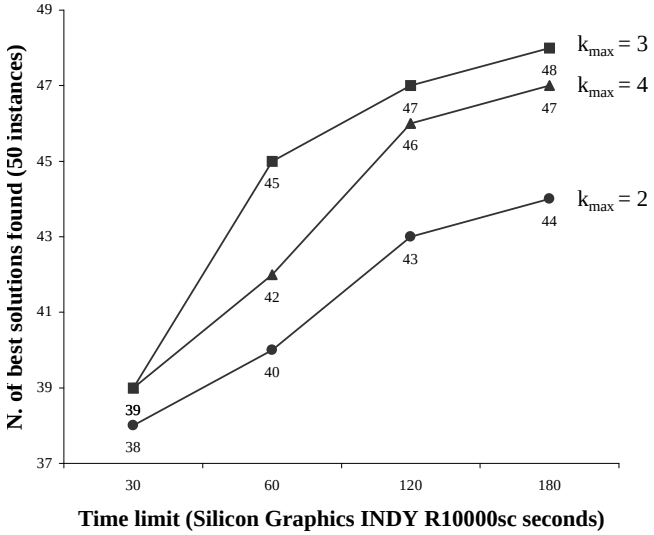


Figure 5.6: Results of the tabu search for different values of $k_{\max}$ and $T_{\lim}$ on the instances of Class 1 for 2BP|R|F (heuristic TP_{RF}).

Good values for the parameters described in Section 5.4 were experimentally determined as: $\alpha = 20$ (see equation 5.1), $d_{\max} = 50$ (maximum value of the differentiation counter d), $\tau_k = 3$ for all k (tabu tenure). As for $k_{\max}$ (maximum number of distinct tabu lists), we performed experiments with values from 1 to 4, using different time limits: $T_{\lim} = 30, 60, 120, 180$ CPU seconds. The outcome was as follows: (i) for $k_{\max} = 1$ the results were always poor, very seldom improving the starting solution; (ii) for $k_{\max} = 2$ the results were quite good with $T_{\lim} = 30$, slightly improving with higher time limits; (iii) for $k_{\max} = 3$ and $T_{\lim} = 30$ the results were slightly worse than for $k_{\max} = 2$, considerably improved with $T_{\lim} = 60$, and very marginally improved with higher time limits; (iv) for $k_{\max} = 4$ we had practically no improvement with respect to $k_{\max} = 3$. Hence, we adopted the best tradeoff, i.e., $k_{\max} = 3$ and $T_{\lim} = 60$. As an example, we report in Figure 5.6 the results obtained on the fifty instances of 2BP|R|F for Class 1, which turned out to be quite sensitive to the parameter variations.

Table 5.3: Tabu search with different inner heuristics: (tabu search solution value)/(lower bound), CPU time in Silicon Graphics INDY R10000sc seconds. Random problem instances proposed by Berkey and Wang.

		Berkey and Wang				Lodi, Martello and Vigo							
		2BP|O|∗ :		2BP|R|∗ :		2BP|O|G:		2BP|R|G:		2BP|O|F:		2BP|R|F:	
		heur.	FBS_{OG}	heur.	FBS_{RG}	heur.	KP_{OG}	heur.	KP_{RG}	heur.	AD_{OF}	heur.	TP_{RF}
Class	n	$\frac{TS}{LB}$	time	$\frac{TS}{LB}$	time	$\frac{TS}{LB}$	time	$\frac{TS}{LB}$	time	$\frac{TS}{LB}$	time	$\frac{TS}{LB}$	time
	20	1.09	36.00	1.06	24.00	1.11	42.00	1.03	12.39	1.06	24.00	1.05	18.00
	40	1.08	54.00	1.06	45.90	1.08	54.17	1.05	36.06	1.06	36.11	1.04	30.02
5	60	1.05	54.05	1.08	54.24	1.05	54.07	1.05	38.60	1.04	48.93	1.04	34.00
	80	1.04	44.18	1.07	55.44	1.04	42.32	1.07	54.59	1.05	48.17	1.06	48.08
	100	1.04	60.37	1.05	60.24	1.05	60.19	1.04	55.65	1.04	60.81	1.03	47.92
Average		1.060	49.72	1.064	47.96	1.066	50.55	1.048	39.46	1.050	43.60	1.044	35.60
	20	1.10	0.01	1.00	0.01	1.00	0.01	1.00	0.01	1.00	0.01	1.00	0.01
	40	1.10	0.01	1.10	0.01	1.10	0.01	1.10	0.01	1.10	0.01	1.10	0.01
6	60	1.15	0.02	1.10	0.01	1.15	0.06	1.15	0.06	1.10	0.09	1.00	0.01
	80	1.07	12.00	1.07	12.00	1.07	12.00	1.03	6.24	1.07	12.00	1.03	6.10
	100	1.06	12.00	1.03	6.03	1.03	6.00	1.03	6.00	1.03	6.00	1.00	0.01
Average		1.096	4.80	1.060	3.61	1.070	3.61	1.062	2.46	1.060	3.62	1.026	1.22
	20	1.18	48.00	1.12	30.00	1.18	48.00	1.09	24.00	1.20	54.00	1.06	18.00
	40	1.12	60.00	1.15	60.00	1.12	60.00	1.11	48.19	1.11	54.02	1.09	42.17
7	60	1.08	48.03	1.11	60.02	1.07	49.48	1.10	60.02	1.05	45.67	1.08	54.15
	80	1.07	54.03	1.12	60.03	1.08	57.24	1.07	60.05	1.08	54.31	1.07	60.07
	100	1.09	60.10	1.10	60.11	1.09	60.09	1.08	60.19	1.09	60.10	1.07	60.18
Average		1.108	54.03	1.120	54.03	1.108	54.96	1.090	50.49	1.106	53.62	1.074	46.91
	20	1.00	0.01	1.00	0.01	1.00	0.01	1.00	0.01	1.00	0.01	1.00	0.01
	40	1.10	0.01	1.10	0.01	1.10	0.01	1.00	0.01	1.00	0.01	1.00	0.01
8	60	1.20	0.03	1.10	0.01	1.20	0.09	1.10	0.04	1.15	0.14	1.10	0.09
	80	1.10	18.00	1.10	18.00	1.10	18.06	1.03	6.28	1.10	18.00	1.07	12.00
	100	1.10	18.00	1.07	12.00	1.10	18.00	1.03	6.10	1.03	6.00	1.03	6.00
Average		1.100	7.21	1.074	6.00	1.100	7.23	1.032	2.48	1.056	4.83	1.040	3.62
	20	1.13	42.00	1.06	18.00	1.13	42.00	1.04	12.01	1.11	36.02	1.04	12.01
	40	1.09	48.00	1.10	42.00	1.09	48.00	1.07	42.01	1.04	27.07	1.07	42.00
9	60	1.06	55.59	1.09	60.01	1.07	58.26	1.07	48.72	1.06	56.77	1.06	45.23
	80	1.06	60.05	1.10	60.05	1.08	60.07	1.08	54.60	1.06	56.18	1.07	54.14
	100	1.08	60.14	1.10	60.12	1.09	60.17	1.07	60.26	1.08	60.34	1.07	60.12
Average		1.084	53.16	1.090	48.04	1.092	53.70	1.066	43.52	1.070	47.28	1.062	42.70
	20	1.00	0.01	1.00	0.01	1.00	0.01	1.00	0.01	1.00	0.01	1.00	0.01
	40	1.40	0.01	1.40	0.01	1.50	0.02	1.40	0.02	1.40	0.03	1.40	0.03
10	60	1.10	0.01	1.05	0.01	1.10	0.05	1.05	0.02	1.05	0.04	1.05	0.05
	80	1.00	0.01	1.00	0.01	1.00	0.01	1.00	0.01	1.00	0.01	1.00	0.01
	100	1.10	18.00	1.07	12.00	1.10	18.00	1.07	12.11	1.07	12.00	1.07	12.00
Average		1.120	3.60	1.104	2.40	1.140	3.61	1.104	2.43	1.104	2.41	1.104	2.42

Table 5.4: Tabu search with different inner heuristics: (tabu search solution value)/(lower bound), CPU time in Silicon Graphics INDY R10000sc seconds. Random problem instances proposed by Martello and Vigo.

		Berkey and Wang				Lodi, Martello and Vigo							
		2BP|O|∗ : heur. FBS_{OG}		2BP|R|∗ : heur. FBS_{RG}		2BP|O|G: heur. KP_{OG}		2BP|R|G: heur. KP_{RG}		2BP|O|F: heur. AD_{OF}		2BP|R|F: heur. TP_{RF}	
Class	n	$\frac{TS}{LB}$	time	$\frac{TS}{LB}$	time	$\frac{TS}{LB}$	time	$\frac{TS}{LB}$	time	$\frac{TS}{LB}$	time	$\frac{TS}{LB}$	time
	20	1.08	24.01	1.15	42.06	1.08	24.00	1.11	30.00	1.04	12.02	1.11	30.00
	40	1.07	42.06	1.17	60.00	1.07	42.00	1.07	44.97	1.06	37.01	1.08	48.06
1	60	1.05	36.68	1.16	60.01	1.05	36.49	1.06	54.30	1.05	36.44	1.06	59.45
	80	1.05	60.08	1.17	60.06	1.05	60.09	1.08	60.20	1.04	54.52	1.10	60.12
	100	1.03	48.40	1.16	60.18	1.04	49.31	1.07	60.71	1.03	47.43	1.08	60.36
	Average	1.056	42.25	1.162	56.46	1.058	42.38	1.078	50.04	1.044	37.48	1.086	51.60
	20	1.12	36.00	1.16	42.00	1.12	36.00	1.10	30.04	1.06	18.04	1.10	30.01
	40	1.04	25.34	1.19	60.00	1.04	24.68	1.08	51.22	1.03	18.72	1.10	54.22
2	60	1.03	30.90	1.17	60.04	1.03	30.61	1.07	48.41	1.02	20.99	1.07	56.17
	80	1.03	44.02	1.16	60.13	1.03	45.87	1.08	60.23	1.02	37.95	1.08	60.11
	100	1.04	54.36	1.17	60.18	1.04	54.22	1.08	60.48	1.04	52.66	1.09	60.14
	Average	1.052	38.12	1.170	56.47	1.052	38.28	1.082	50.08	1.034	29.67	1.088	52.13
	20	1.00	0.01	1.00	0.01	1.00	0.01	1.00	0.01	1.00	0.01	1.00	0.06
	40	1.01	24.01	1.01	18.00	1.01	24.02	1.01	18.02	1.01	24.05	1.01	18.85
3	60	1.01	24.09	1.01	18.14	1.01	24.15	1.01	18.10	1.01	24.26	1.01	18.03
	80	1.02	54.44	1.01	30.24	1.01	48.61	1.01	30.52	1.01	54.31	1.01	30.51
	100	1.01	31.97	1.01	30.68	1.01	30.60	1.01	30.93	1.01	34.11	1.01	36.86
	Average	1.010	26.90	1.008	19.41	1.008	25.48	1.008	19.51	1.008	27.35	1.008	20.86
	20	1.14	24.00	1.12	6.00	1.14	24.00	1.12	6.00	1.10	12.00	1.12	6.01
	40	1.09	36.00	1.07	32.32	1.09	36.03	1.08	30.87	1.06	25.18	1.06	24.01
4	60	1.08	48.03	1.07	40.44	1.08	48.05	1.07	36.73	1.07	42.13	1.06	30.44
	80	1.10	60.02	1.06	48.03	1.10	60.02	1.06	45.68	1.06	47.30	1.05	39.04
	100	1.08	60.10	1.06	54.57	1.07	60.13	1.05	42.28	1.08	60.10	1.05	43.38
	Average	1.098	45.63	1.076	36.27	1.096	45.65	1.076	32.31	1.074	37.34	1.068	28.58

The computational results are presented in Tables 5.3 and 5.4. As in the previous section, the first two columns give the class and the value of n. The next pairs of columns refer to algorithms FBS_{OG} (Berkey and Wang [23]), FBS_{RG} (Section 5.2), and to the four heuristics in Section 5.3, when used within the tabu search. For each algorithm the entries give the average ratio (tabu search solution value)/(lower bound), computed over the ten generated instances, and the average CPU time. In this case too, for each class, the final line gives the average over all values of n.

The results are satisfactory. By considering, for each class, the average values computed over all values of n, we can observe that the tabu search generally improves the initial deterministic solution produced by the inner heuristic, with almost all exceptions occurring for Classes 6, 8 and 10. This is not surprising since, for the instances in these classes, depending on the value of n, it is either very easy or very hard to prove the optimality of a presumably optimal heuristic solution. Consider for example Class 6, for which the average area of an item is 30.25, hence, on average, 29.75 items are packed in each bin: for $n = 20$, 40, 80 and

100 it is then easy to find an optimal solution using one, two, three and four bins, respectively, while for $n = 60$ it is very difficult to find a feasible solution using two bins (if any).

In this case, too, the percentage errors could be much smaller if computed with respect to the optimal solution values. For the instances of 2BP|O|F, the average errors computed by using the best available solution (not always optimal) are 1.004, 1.004, 1.000, 1.008, 1.009, 1.000, 1.035, 1.000, 1.014 and 1.000, , respectively for Classes 1 to 10, i.e., about one eighth, on average, of those in the tables.

The computational results presented in this section show that tabu search is effective in all cases, regardless of the inner deterministic algorithm used in the search. This proves, for 2BP, the quality of a unified tabu search approach, capable to further improve the performance of effective deterministic heuristics.

Chapter 6

Other Packing Problems

6.1 Tabu Search: the Three-Dimensional Case

The *Three-Dimensional Bin Packing Problem* (3BP), already defined in Section 2.6.1, is the direct extension of 2BP (and, obviously, of BPP) to higher dimensional space. In 3BP the items and the bins are rectangular boxes, and the objective is always to orthogonally pack the items in the minimum number of bins without overlapping.

The main goal of this section is to show how the Tabu Search scheme discussed in the previous chapters, and presented for a wide set of two-dimensional packing problems, can be applied without substantial changes, and with satisfactory results, also to packing problems in three dimensions.

In order to use the presented Tabu Search metaheuristic we obviously need a subordinate heuristic (recall the discussion of Section 5.4) to obtain the partial solutions which are combined by the Tabu Search. In Section 6.1.1 a new heuristic algorithm for 3BP is presented, whereas in Section 6.1.2 the very slight adaptation of the Tabu Search to the three-dimensional case is discussed together with the final set of parameters used in the computational experiments. Finally, Section 6.1.3 presents the computational results on the set of instances proposed by Martello, Pisinger and Vigo [116]. The heuristic and metaheuristic approaches are compared with both the heuristic and exact algorithms in [116].

6.1.1 A New Heuristic Algorithm for 3BP

In this section we introduce a new heuristic algorithm for 3BP called *Height first-Area second* (HA). Algorithm HA packs one item at a time into layers, i.e., three-dimensional shelves in which (i) the height of the layer is defined by the height of the tallest item packed into it; (ii) all the items are packed with their basis on the floor of the layer. Thus, in order to obtain an "effective" packing into layers one must solve two main problems: packing in the same layer items with similar height (vertical component), and the two-dimensional problem on the basis of the layer (2BP component). Algorithm HA works by taking into account at the same time these two problems. However, it generates two possibly different solutions, say z_1 and z_2, depending on which component has been considered with "major emphasis".

As algorithm HA packs one item at a time, it is easy to see that a simple way to emphasize a component with respect to the other is to consider the items with a suitable order. In particular, sorting the items according to nonincreasing height values emphasizes the vertical packing since it tries to create layers in which the items have similar heights. On the other

hand, sorting the items by nonincreasing values of $w_j d_j$ is crucial to obtain good packing on the floor of the layers (see Section 5.3.4).

The packing strategy of algorithm HA is obtained by adapting the policy used for algorithm TP_{RF} (see Section 5.3.4, with a first difference that in 3BP case the rotation of the items' basis is not allowed). In particular, given an item j, a *score* is computed for each *normal* packing position h on the basis of each layer t ($score = 0$ if j cannot be packed in position h on the basis of t), and the packing position with maximum score is selected. The score is computed by taking into account three different factors: (i) the percentage of the perimeter of the basis of j touching the edges of the basis of t or the edges of items already packed into t; (ii) the percentage of the area of the basis of t already packed, and (iii) the relative difference between the height of t and the height of j. Let H_t indicate the height of layer t, S_t be the set of items currently packed into t, and $TP(t, S_t, j, h)$ the "touching perimeter": then the score of the packing of item j into layer t in the normal position h is computed as

$$score^t_{jh} = \tau \frac{TP(t, S_t, j, h)}{2w_j + 2d_j} + \mu \frac{\sum_{l \in S_t} w_l d_l}{WD} - (1 - \tau - \mu) \frac{H_t - h_j}{H_t} \tag{6.1}$$

where τ and μ are prefixed real values such that $\tau, \mu \in [0, 1]$, and $\tau + \mu \leq 1$.

Again, the three terms in (6.1) measure the effectiveness of one packing with respect to vertical and 2BP components, and are considered with different weights: the values of parameters τ and μ change depending on which component of 3BP is emphasized.

Note that the third term of (6.1) is positive if $H_t > h_j$, whereas is negative if $H_t < h_j$. In the first case, layer t is said to be *feasible* for item j, whereas in the second case t is *infeasible* for j. Anyway, it is sometimes allowed to pack item j into layer t even if t is infeasible (to be discussed later), and in those cases the term is computed as absolute value, and the height of layer t is updated as $H_t = h_j$.

In the following we will separately present the two *Steps* of algorithm HA in which solutions z_1 and z_2 are obtained.

Step 1. For the first solution, the items are partitioned into clusters, each characterized by a different height. In particular, after a preliminary sorting by nonincreasing height values, the first item j (the tallest one) defines the first cluster with height h_j, and each of the following items with height h_i within a prefixed range ($h_i \geq \beta h_j$, where $\beta \in [0, 1]$) belongs to the cluster. The tallest item s for which $h_s < \beta h_j$ defines a new cluster with height equal to h_s, and so on. As soon as the instance has been partitioned, the items in each cluster are sorted by nonincreasing values of $w_j d_j$ and re-numbered.

After these ordering phase, the items are packed one at a time into the *open* layers (an open layer does not necessarily have items packed into), in the normal packing position (if any) with maximum score. Among these layers the *feasible* ones are preferred, hence only in the case where no feasible layer can accommodate one item, say j, *infeasible* layers are considered (if any). A new layer with height equal to h_j is opened if neither feasible nor infeasible layer can accommodate item j, and j is packed left-justified into the new layer. A finite bin packing solution is finally obtained by packing the layers into the bins using an exact algorithm for the 1BP instance in which each layer t defines an item of height H_t, and the size of the bin is H.

The overall step 1. is as follow.

algorithm HA, step 1:
sort the items, define the clusters, sort and re-number the items in each cluster;
for $j := 1$ **to** n **do**
$score := 0$;
for each normal packing position h in an open *feasible* layer t **do**
compute the corresponding score $score^t_{jh}$;
$score := \max\{score, score^t_{jh}\}$
end for;
if $score > 0$ **then**
pack item j in the layer and position corresponding to *score*
else
for each normal packing position h in an open *infeasible* layer t **do**
compute the corresponding score $score^t_{jh}$;
$score := \max\{score, score^t_{jh}\}$
end for;
if $score > 0$ **then**
pack item j in the layer and position corresponding to *score*,
and update the height of the layer
else
open a new layer with height h_j, and pack item j into it
end for
let $H_1, \ldots, H_g$ be the heights of the resulting g layers;
determine a finite bin solution z_1 by solving the 1BP instance having g elements,
with associated values $H_1, \ldots, H_g$, and capacity H
end.

(Note that in Step 1. no layer is opened without packing one item into it.)

Preliminary experiments suggested for parameter τ and μ the values $\tau = 0.3$ and $\mu = 0.7$ which have been used in the computational testing of Section 6.1.3. This choice implies that the third term of (6.1) is not taken into account: the vertical component has been already emphasized through the use of the mentioned partition into clusters. Finally, the partition into cluster is guided by parameter $\beta = 0.75$.

Step 2. For the second solution, the items are directly sorted and re-numbered by nonincreasing values of $w_j d_j$. As mentioned, this ordering policy emphasizes the 2BP component. Anyway, the vertical component of 3BP is carefully considered by opening g empty layers with "suggested" heights $H_1, \ldots, H_g$, i.e., the heights of the layers originating z_1.

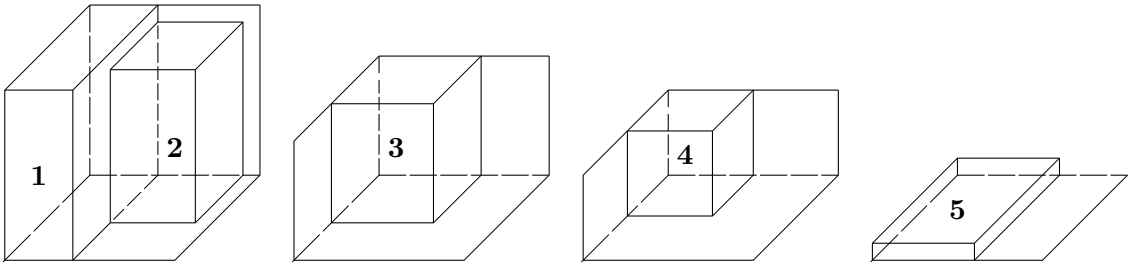
A part of the above differences (the initial sorting and the g empty open layers), Step 1. and Step 2. work exactly in the same way. Note that, even if the ordering by nonincreasing area could generate layers with items of very different heights, this is, in general, avoided by the score computation. In fact, due to the presence of empty but already open layers and to the third term of (6.1), the packing of an item j into a layer t such that $H_t \gg h_j$ (or $H_t \ll h_j$) has smaller score than the packing of j into an empty layer with more closer height. Hence, the values of parameters τ and μ are updated in Step 2., and, for the computational testing of Section 6.1.3, we selected $\tau = 0.2$ and $\mu = 0.3$ (thus weighting the third term of (6.1) as 0.5).

Finally, consider the 1BP instance defined at the end of Step 2., whose solution is the

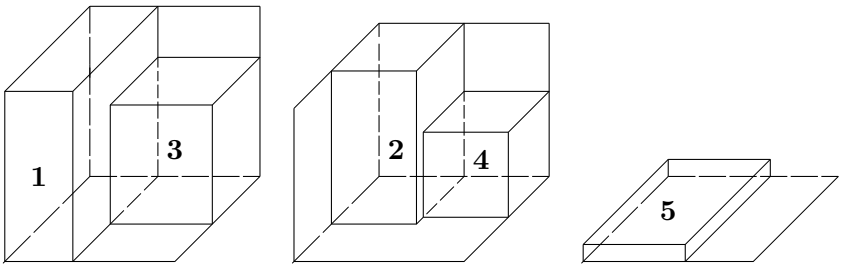
finite bin packing solution of value z_2. It is obvious that a preliminary step checking if some of the layers remained empty is needed, and in that case, they are eliminated.

The overall solution of algorithm HA is $z = \min\{z_1, z_2\}$. An example showing the effectiveness of performing Step 2. after Step 1. is the following.

Example
We are given a small instance with $n = 5$: $W = H = D = 10$, $w_1 = 4, h_1 = d_1 = 10$, $w_2 = 5, h_2 = 9, d_2 = 6$, $w_3 = 6, h_3 = 7, d_3 = 6$, $w_4 = h_4 = d_4 = 5$, and $w_5 = 6, h_5 = 1, d_5 = 10$. The items are already sorted by nonincreasing height values, and, by considering $\beta = 1$, each item defines a cluster, and the packing into layers obtained by Step 1. of algorithm HA is depicted in Figure 6.1(a).



(a): Algorithm HA, Step 1.



(b): Algorithm HA, Step 2.

Figure 6.1: Algorithm HA: (a) layers obtained in Step 1.; (b) layers obtained in Step 2.

Otherwise, in Step 2. the items are sorted by nonincreasing values of $w_j d_j$, the new order is: 5, 1, 3, 2, 4, and the packing is depicted in Figure 6.1(b).
It is easy to see that the layers of Figure 6.1(a) define a solution $z_1 = 3$, whereas $z_2 = 2$.□

In both steps a 1BP instance is generated, thus requiring the solution of an NP-hard problem. This solution is however obtained by halting a non-polynomial routine after a prefixed (small) number of iterations.

The last remark concerns an obvious extension of algorithm HA. The above steps can be easily applied by considering in turn the width and the depth instead of the height. In this way we obtain three different solutions composed by layers, and we select the best one with respect to the number of used bins. In Section 6.1.3 algorithm HA is applied using this rule.

6.1.2 The Tabu Search in the 3D Case

The adaptation needed by the Tabu Search framework presented in Section 5.4 to work in the three-dimensional case is not heavy. In particular, we adapted the concept of *filling function* by extending equation (5.1) to take into account also the third dimension, i.e., the depth. Hence, for each bin i in a 3BP solution, the new filling function is defined by

$$\varphi(S_i) = \alpha \frac{\sum_{j \in S_i} w_j h_j d_j}{WHD} - \frac{|S_i|}{n} \tag{6.2}$$

where again S_i denotes the set of items currently packed into bin i, and α is a prefixed positive parameter.

The overall scheme of the algorithm is the same presented in Section 5.4 and can be summarized as follows. We consider an incumbent solution obtained by performing algorithm HA on the instance, a lower bound on the optimal solution value, and an initial naive solution in which each item is packed into a separate bin. Then, the iterative part of the algorithm is started and the idea is to define subsets of the instance to run the subordinate heuristic on, in an attempt of re-packing the corresponding items in a smaller number of bins, and emptying the bin with the smallest value of the filling function. The definition of these subsets is the core of the method, and it is automatically performed following the guideline described in procedure SEARCH of Section 5.4: in particular, the size of the subset (defined as the number of bins involved) is parametrically updated depending on the results of the previous iterations.

The algorithm performs in a very similar way in the two- and three-dimensional case. The main difference is due to the extension of the filling function: the addition of the third dimension results in a splitting of the φ value of the moves, thus the tuning of the parameters leads to very different values.

The computational testing reported in the next section was performed with $k_{\max} = 4$, $\alpha = 1.5$, and with tabu tenure $\tau_k = 12$ ($k = 1, \ldots, k_{\max}$). By comparing these values with those used for the two-dimensional case, it is clear that the aim was to have more frequent updatings of the neighborhood size (k value), thus obtaining a balanced search strategy. The only situation in which the value of k could originally be increased was the case where no feasible move was found. This rule was powerful (together with the chosen set of values for the parameters) in two dimensions, whereas in the three-dimensional case even the new values of the parameters are not enough and a shortcut to obtain the increasing of k has been introduced. As soon as ℓ calls to procedure SEARCH have been performed with the same k value without improving the incumbent solution, k is increased by one if $k < k_{\max}$; otherwise, a diversification action is performed. (This is the only difference introduced in the code of the Tabu Search, and we used $\ell = 2 * n$ in the computational testing.)

A final remark concerns parameter $d_{\max}$. The value of 50 chosen in Section 5.5.2 denoted the preference of using the soft diversification (changing of the target bin) whenever possible, instead of the strong one in which the current packing is partially destroyed. This was justified by the fact that the number of diversification actions was naturally high, which is not the case in three dimensions. In the computational testing of the next section we set $d_{\max} = 2$.

6.1.3 Computational Experiments

The algorithms of the two previous sections were coded in FORTRAN 77 and run on a Digital Alpha 533 MHz. The heuristic algorithm of Section 6.1.1 solves 1BP instances through the FORTRAN code MTP by Martello and Toth [118] with a limit of 1000 backtrackings. The

algorithms have been tested on eight classes of instances proposed by Martello, Pisinger and Vigo [116], and we compare the results with both heuristic codes `dfirst3_heuristic` and `mcut3_heuristic` (H1 and H2 in Tables 6.1 and 6.2, respectively), and the exact code `binpack3D` (BB in tables) of Martello, Pisinger and Vigo [115].

Table 6.1: Random problem instances proposed by Martello, Pisinger and Vigo. $n < 50$, time limit for Tabu Search of 30 CPU seconds on Digital Alpha 533 MHz, average values over ten instances.

n	Class	$\frac{H1}{V}$	$\frac{H2}{V}$	$\frac{HA}{V}$	$\frac{TS}{V}$	$\frac{BB}{V}$
	1	1.14	1.00	1.08	1.05	1.00
	2	1.13	1.00	1.00	1.00	1.00
	3	1.03	1.00	1.00	1.00	1.00
	4	1.00	1.00	1.00	1.00	1.00
10	5	1.15	1.00	1.00	1.00	1.00
	6	1.13	1.00	1.05	1.05	1.00
	7	1.33	1.05	1.05	1.05	1.00
	8	1.22	1.00	1.10	1.05	1.00
Average		**1.14**	**1.01**	**1.04**	**1.03**	**1.00**
	1	1.12	1.03	1.00	1.00	1.00
	2	1.04	1.00	1.02	1.00	1.00
	3	1.10	1.04	1.02	1.00	1.00
	4	1.02	1.00	1.00	1.00	1.00
20	5	1.25	1.05	1.08	1.08	1.00
	6	1.12	1.03	1.00	1.00	1.00
	7	1.35	1.05	1.08	1.03	1.00
	8	1.09	1.00	1.00	1.00	1.00
Average		**1.14**	**1.03**	**1.02**	**1.01**	**1.00**
	1	1.12	1.03	1.01	1.00	1.00
	2	1.17	1.05	1.04	1.00	1.00
	3	1.10	1.01	1.03	1.02	1.00
	4	1.01	1.01	1.00	1.00	1.00
30	5	1.19	1.04	1.02	1.02	1.00
	6	1.18	1.04	1.03	1.00	1.00
	7	1.56	1.18	1.20	1.12	1.13
	8	1.21	1.05	1.04	1.00	1.00
Average		**1.19**	**1.05**	**1.05**	**1.02**	**1.02**
	1	1.17	1.09	1.07	1.01	1.01
	2	1.15	1.07	1.07	1.03	1.03
	3	1.13	1.08	1.04	1.02	1.02
	4	1.02	1.00	1.00	1.00	1.00
40	5	1.37	1.24	1.17	1.13	1.19
	6	1.17	1.10	1.07	1.01	1.00
	7	1.60	1.30	1.26	1.17	1.29
	8	1.30	1.17	1.10	1.06	1.02
Average		**1.24**	**1.13**	**1.10**	**1.05**	**1.07**

n	Class	$\frac{H1}{V}$	$\frac{H2}{V}$	$\frac{HA}{V}$	$\frac{TS}{V}$	$\frac{BB}{V}$
	1	1.07	1.03	1.00	1.00	1.00
	2	1.15	1.00	1.05	1.03	1.00
	3	1.09	1.00	1.02	1.02	1.00
	4	1.02	1.01	1.00	1.00	1.00
15	5	1.15	1.07	1.03	1.00	1.00
	6	1.13	1.02	1.00	1.00	1.00
	7	1.47	1.10	1.18	1.18	1.00
	8	1.22	1.07	1.05	1.05	1.00
Average		**1.16**	**1.04**	**1.04**	**1.03**	**1.00**
	1	1.13	1.02	1.03	1.00	1.00
	2	1.17	1.05	1.06	1.03	1.00
	3	1.09	1.05	1.03	1.00	1.00
	4	1.02	1.01	1.00	1.00	1.00
25	5	1.21	1.06	1.05	1.03	1.00
	6	1.15	1.04	1.02	1.00	1.00
	7	1.46	1.10	1.07	1.07	1.07
	8	1.27	1.06	1.04	1.03	1.00
Average		**1.19**	**1.05**	**1.04**	**1.02**	**1.01**
	1	1.20	1.09	1.07	1.01	1.00
	2	1.11	1.03	1.03	1.01	1.00
	3	1.14	1.03	1.03	1.00	1.00
	4	1.01	1.01	1.00	1.00	1.00
35	5	1.24	1.09	1.08	1.02	1.09
	6	1.21	1.10	1.07	1.05	1.00
	7	1.49	1.26	1.20	1.13	1.19
	8	1.35	1.20	1.09	1.08	1.16
Average		**1.22**	**1.10**	**1.07**	**1.04**	**1.05**
	1	1.17	1.11	1.07	1.03	1.06
	2	1.16	1.07	1.07	1.04	1.04
	3	1.17	1.10	1.10	1.05	1.05
	4	1.02	1.00	1.00	1.00	1.00
45	5	1.24	1.13	1.06	1.05	1.09
	6	1.18	1.12	1.08	1.02	1.00
	7	1.69	1.45	1.37	1.30	1.42
	8	1.32	1.18	1.13	1.06	1.06
Average		**1.24**	**1.15**	**1.11**	**1.07**	**1.09**

As mentioned in Section 2.6.1 the instances are generalizations of the 2BP instances of Martello and Vigo [119] (Classes 1–5), and Berkey and Wang [23] (Classes 6–8)[1].

For each class a set of 140 instances were solved, ten for each value of $n \in \{10, 15, 20, 25, 30, 35, 40, 45, 50, 60, 70, 80, 90, 100\}$, and in Tables 6.1 and 6.2, for each algorithm, we report the

[1]Class 9 in [116] has not been considered here due to the special structure of the instances: algorithms producing packing into layers (like HA) cannot find good solutions because the items are obtained by cutting the complete bin with non-guillotine cuts.

Table 6.2: Random problem instances proposed by Martello, Pisinger and Vigo. $n > 50$, time limit for Tabu Search of 180 CPU seconds on Digital Alpha 533 MHz, average values over ten instances.

n	Class	$\frac{H1}{V}$	$\frac{H2}{V}$	$\frac{HA}{V}$	$\frac{TS}{V}$	$\frac{BB}{V}$
	1	1.19	1.14	1.08	1.04	1.06
	2	1.25	1.14	1.11	1.08	1.08
	3	1.20	1.14	1.11	1.05	1.07
	4	1.02	1.01	1.00	1.00	1.00
50	5	1.34	1.28	1.13	1.11	1.24
	6	1.20	1.12	1.07	1.01	1.00
	7	1.49	1.32	1.27	1.17	1.28
	8	1.40	1.29	1.21	1.13	1.24
Average		**1.26**	**1.18**	**1.12**	**1.07**	**1.12**
	1	1.24	1.15	1.13	1.08	1.11
	2	1.22	1.15	1.11	1.06	1.09
	3	1.22	1.16	1.13	1.08	1.11
	4	1.05	1.02	1.02	1.02	1.02
70	5	1.30	1.27	1.16	1.12	1.25
	6	1.22	1.20	1.11	1.05	1.05
	7	1.65	1.47	1.33	1.25	1.46
	8	1.36	1.27	1.15	1.09	1.17
Average		**1.28**	**1.21**	**1.14**	**1.10**	**1.16**
	1	1.18	1.14	1.10	1.06	1.09
	2	1.18	1.14	1.11	1.07	1.09
	3	1.17	1.15	1.11	1.07	1.09
	4	1.04	1.04	1.03	1.03	1.03
90	5	1.32	1.34	1.20	1.14	1.30
	6	1.15	1.18	1.06	1.02	1.02
	7	1.49	1.46	1.29	1.19	1.45
	8	1.35	1.32	1.20	1.15	1.24
Average		**1.24**	**1.22**	**1.14**	**1.09**	**1.16**

n	Class	$\frac{H1}{V}$	$\frac{H2}{V}$	$\frac{HA}{V}$	$\frac{TS}{V}$	$\frac{BB}{V}$
	1	1.26	1.15	1.14	1.07	1.10
	2	1.21	1.15	1.11	1.06	1.09
	3	1.22	1.14	1.12	1.08	1.09
	4	1.02	1.02	1.01	1.01	1.01
60	5	1.45	1.31	1.25	1.17	1.31
	6	1.15	1.13	1.06	1.01	1.00
	7	1.56	1.41	1.28	1.20	1.38
	8	1.25	1.17	1.10	1.04	1.09
Average		**1.26**	**1.18**	**1.13**	**1.08**	**1.13**
	1	1.19	1.17	1.12	1.07	1.09
	2	1.19	1.15	1.12	1.07	1.10
	3	1.17	1.13	1.09	1.06	1.09
	4	1.03	1.03	1.02	1.02	1.02
80	5	1.36	1.33	1.18	1.12	1.33
	6	1.20	1.20	1.11	1.06	1.08
	7	1.45	1.39	1.24	1.18	1.38
	8	1.36	1.34	1.19	1.15	1.25
Average		**1.24**	**1.22**	**1.13**	**1.09**	**1.17**
	1	1.19	1.14	1.10	1.07	1.09
	2	1.19	1.16	1.12	1.08	1.10
	3	1.17	1.14	1.10	1.05	1.10
	4	1.05	1.04	1.02	1.02	1.03
100	5	1.37	1.40	1.22	1.17	1.36
	6	1.25	1.26	1.14	1.09	1.10
	7	1.46	1.41	1.23	1.16	1.41
	8	1.23	1.25	1.14	1.09	1.16
Average		**1.24**	**1.22**	**1.13**	**1.09**	**1.17**

average ratio (solution value)/V, where V is the optimal solution value if known, or the highest lower bound value (from [116]) otherwise.

Table 6.1 contains the results for the instances with $n < 50$, and the Tabu Search is considered with a time limit of 30 CPU seconds, whereas, as mentioned in Section 2.6.1, the branch-and-bound algorithm received a time limit of 1000 CPU seconds on a HP9000/C160 (160 MHz). (The computing times of algorithm HA, as well as the ones of `dfirst3_heuristic` and `mcut3_heuristic`, are negligible, and are not reported here.) These results show a satisfactory behavior of HA with respect to `dfirst3_heuristic` and `mcut3_heuristic`: although HA could only obtain solutions composed by layers (thus solving a 3BP more constrained problem), it is always better, on average, than the other heuristics (`mcut3_heuristic` does not pack into layers), but for the case $n = 10$. In addition, the Tabu Search approach is effective in improving the solution of HA, and this improvement becomes very important for $n > 25$, thus leading to solutions better, on average, than the ones obtained by branch-and-bound[2]. This behavior is confirmed and emphasized by the results of Table 6.2 for n between 50 and 100, for which the time limit for TS was 180 CPU seconds.

[2]Note that the tabu search solutions are composed by layers due to the inner heuristic, but, since HA is applied to subsets of items, then these layers possibly have different orientations in different bins. Hence, some of the 3BP optimal solutions cannot be obtained by TS.

Finally, in Table 6.3 we report some aggregated results clarifying the behavior of the Tabu Search. In particular, for each value of n we give four entries: the number of times the Tabu Search obtains a solution at least as good as the branch-and-bound ($TS \leq BB$ in Table 6.3), the number of times the solution of the Tabu Search is strictly better than the one of the branch-and-bound ($TS < BB$), the number of times the Tabu Search finds the optimal solution (TS `opt`); and the number of times the solution of the Tabu Search is better than the one of the branch-and-bound, and it is proved to be optimal (TS `close`).

Table 6.3: Random problem instances proposed by Martello, Pisinger and Vigo. Tabu Search vs. Branch-and-Bound over the 1120 considered instances.

n	10	15	20	25	30	35	40	45	50	60	70	80	90	100	*Total*	*Percent*
$TS \leq BB$	76	73	77	75	77	74	77	76	79	79	77	78	79	78	**1075**	**95.98%**
$TS < BB$	0	0	0	1	1	10	8	13	21	30	38	41	44	48	**255**	**22.77%**
TS `opt`	76	73	77	74	74	66	61	51	43	30	21	15	15	9	**685**	**61.16%**
TS `close`	0	0	0	0	1	3	1	2	4	4	4	5	3	4	**31**	**2.77%**

Very recently Færø, Pisinger and Zachariasen [66] proposed a new metaheuristic approach for 3BP based on the Guided Local Search method [149]. The idea is to iteratively solve constraint satisfaction problems in order to improve a starting solution obtained by `mcut3_heuristic`.

The preliminary results of computational experiments on some of the previous instances are reported in Table 6.4.

Table 6.4: Random problem instances proposed by Martello, Pisinger and Vigo. Tabu Search vs. Guided Local Search.

n	Class	*LB*	*BB*	*GLS*	*TS*
	1	12.50	13.60	13.40	13.40
	4	28.70	29.40	29.40	29.40
50	5	7.30	9.20	**8.30**	8.40
	6	8.70	9.80	**9.80**	9.90
	7	6.30	8.10	**7.40**	7.50
	8	8.00	10.10	**9.20**	9.30
	1	25.10	27.30	26.70	**26.60**
	4	57.60	59.10	59.00	59.00
100	5	12.90	17.50	15.10	**15.00**
	6	17.50	19.40	19.10	19.10
	7	10.90	15.30	**12.40**	12.50
	8	17.50	20.20	18.90	18.90

We consider experiments with $n \in \{50, 100\}$, for classes 1, 4, 5, 6, 7, and 8 above, and we compare our Tabu Search (TS in Table 6.4) with the best known lower bound (LB), the Guided Local Search method (GLS), and with the branch-and-bound in [115] (BB). Each entry in the table gives the average number of bins over ten instances. The same time limit

of 1,000 CPU seconds was given to TS and GLS on very similar machines, Digital Alpha 533 MHz. and Digital workstation 500au 500 Mhz., respectively.

Although again the solutions obtained by TS, due to HA, are composed by layers, the results show an effective behavior. In fact, TS and GLS find the same solution in 113 out of the 120 instances, 5 times GLS is better than TS (4 for $n = 50$, and 1 for $n = 100$), whereas 2 times TS is better than GLS (for $n = 100$).

6.2 2BP, Compatibility Graphs and Stable Sets

Given any 2BP instance I it is straightforward to define the associated *compatibility* undirected *Graph* $G = (V, E)$. Formally, for each item $j \in I$ ($j = 1, \ldots, n$), we define a node $j \in V$, and for each pair of items $i, j \in I$ (hence $i, j \in V$), $(i, j) \in E$ if and only if:

$$w_i + w_j \leq W \quad \textbf{or} \quad h_i + h_j \leq H \tag{6.3}$$

where W and H are the weight and height of the bin. Two items for which (6.3) holds are said to be *compatible*, i.e., they can be packed into the same bin (otherwise they are said to *incompatible*).

A simple observation is that the cardinality of each *stable set* of G defines a valid *lower bound* for 2BP. Then, the best of these lower bounds can be obtained by computing the *Maximum Cardinality Stable Set Problem* (or Maximum Stable Set Problem, MSSP for short) which is NP-hard for general graphs (see, Garey and Johnson [78]).

MSSP is known to be polynomially solvable for perfect graphs [86], and Hammer and Mahadev [92] showed that *bithreshold graphs* are (strongly) perfect and gave a polynomial time algorithm to solve the problem with time complexity $O(n^2)$. It is quite easy to see that 2BP compatibility graphs are bithreshold, hence MSSP is polynomial also for these graphs.

In the next section, we propose an algorithm which is able to solve MSSP with time complexity $O(n \log n)$ for 2BP compatibility graphs.

6.2.1 An $O(n \log n)$ Algorithm

The algorithm presented in this section is based on the partition of the items of any instance I of 2BP into four sets:

- set M is such that $w_i > \frac{1}{2}W \quad \textbf{and} \quad h_i > \frac{1}{2}H \quad \forall i \in M$;
- set MW is such that $w_i > \frac{1}{2}W \quad \textbf{and} \quad h_i \leq \frac{1}{2}H \quad \forall i \in MW$;
- set MH is such that $w_i \leq \frac{1}{2}W \quad \textbf{and} \quad h_i > \frac{1}{2}H \quad \forall i \in MH$;
- set N is such that $w_i \leq \frac{1}{2}W \quad \textbf{and} \quad h_i \leq \frac{1}{2}H \quad \forall i \in N$.

(Obviously, $M \bigcup MW \bigcup MH \bigcup N$ defines the overall set of the items of any instance I.)

Proposition 6.1. *Given any instance of 2BP, and the associated compatibility graph* $G = (V, E)$*, the maximum stable set* G *has cardinality* $|M| + c$*, with* $c \in \{0, 1, 2\}$.

Proof. Since M is by definition a stable set, we immediately have $c \geq 0$. Note that there cannot exist three pairwise incompatible items in $MW \bigcup MH \bigcup N$. In fact, an item in MW can

be incompatible only with items in MH, but two items in MH are obviously compatible, and vice-versa. In addition, no item in N is incompatible either with items in $MW \bigcup MH \bigcup N$. It follows that at most two items in $MW \bigcup MH \bigcup N$ can belong to the maximum stable set, and since a stable set of cardinality $|M|$ is already defined without these items, then c can be at most equal to 2. □

The above proposition suggests the algorithm called *2BP_Stable* (2BP_St) to detect the maximum stable set on 2BP compatibility graphs. From the proof of Proposition 6.1. it is clear that the case $|M|+2$ has to be tested by searching for a pair (i, j) with $i \in MW, j \in MH$ such that:

(i) i, j are incompatible;

(ii) i, k are incompatible **and** j, k are incompatible $\forall k \in M$.

In order to find such a pair, we define the subset $S_1 \subseteq MW$ (resp. $S_2 \subseteq MH$) of the items i such that $h_i + h_t > H$ (resp. $w_i + w_s > W$), where $t = \arg\min_{k \in M} h_k$ (resp. $s = \arg\min_{k \in M} w_k$). Only the items in S_1 and S_2 may belong to the pair (if any). Before giving the overall algorithm we need three additional observations:

1. the initial partition of the items can be done in $O(n)$ time, and, without additional effort, we can store $w_{min} = \min_{k \in M} w_k$, $h_{min} = \min_{k \in M} h_k$, $w_{2min} = \min_{k \in M \setminus \{s\}} w_k$ and $h_{2min} = \min_{k \in M \setminus \{t\}} h_k$ for set M, and $h_{max} = \max_{k \in MW} h_k$ and $w_{max} = \max_{k \in MH} w_k$ for sets MW and MH, respectively;

2. once sets S_1 and $S_2$are defined, the case $|M| + 2$ is reduced to the problem of finding two incompatible items in $S_1 \bigcup S_2$, which can be done by procedure CHECK_INC (to be discussed later);

3. a possible way to obtain a stable set of cardinality $|M| + 1$ is to remove one item from M and add a pair (i, j) of the above type, and the only two items which one can in turn try to remove are s and t. It is easy to see that if item s (resp. t) is removed, a subset S_2' (resp. S_1') can be defined by using w_{2min} (resp. h_{2min}) instead of w_{min} (resp. h_{min}), and we obtain a problem of the same type as the one described by 2. above for the set $S_1 \bigcup S_2'$ (resp. $S_1' \bigcup S_2$).

   The other way is, instead, to add a single item which can belong to MW, MH or N (see, algorithm 2BP_S below).

Then, by indicating the maximum stable set by SS, algorithm 2BP_S is the following.

algorithm 2BP_S:
partition the items into the sets M, MW, MH, N and
store the values s, t, w_{min}, h_{min}, w_{2min}, h_{2min}, h_{max}, and w_{max};
comment: case $|M| + 2$;
$INC :=$ `false`;
define sets $S_1 \subseteq MW$, and $S_2 \subseteq MH$;
call CHECK_INC($S_1 \bigcup S_2$,H,W,INC,k,ℓ);
if $INC =$ `true` **then**
begin

```
            SS := M ⋃{k} ⋃{ℓ};
            stop
        end
    comment: case |M| + 1, subcase 1.;
    define set S'_1;
    call CHECK_INC(S'_1 ⋃ S_2,H,W,INC,k,ℓ);
    if INC = true then
        begin
            SS := M \ {t} ⋃{k} ⋃{ℓ};
            stop
        end
    else
        begin
            define set S'_2;
            call CHECK_INC(S_1 ⋃ S'_2,H,W,INC,k,ℓ);
            if INC = true then
                begin
                    SS := M \ {s} ⋃{k} ⋃{ℓ};
                    stop
                end
        end
    comment: case |M| + 1, subcase 2.;
    if w_max + w_min > W then
        begin
            SS := M ⋃{arg max_{k∈MH} w_k};
            stop
        end
    if h_max + h_min > H then
        begin
            SS := M ⋃{arg max_{k∈MW} h_k};
            stop
        end
    comment: case |M| + 1, subcase 3.;
    for each k ∈ N do
        begin
            if w_k + w_s > W and h_k + h_t > H then
                begin
                    SS := M ⋃{k};
                    stop
                end
        end
    comment: case |M|;
    SS := M
end.
```

The complexity of algorithm 2BP_S strongly depends on procedure CHECK_INC. As mentioned above, the initial partition is done in $O(n)$ time, whereas subcase 2. is tested in

constant time, and subcase 3. requires $O(n)$. An obvious way of testing incompatibility for the pairs of items in a set requires $O(n^2)$ time, but by using an appropriate data structure this can be done in $O(n \log n)$, thus determining the overall complexity of 2BP_S. In particular, procedure CHECK_INC is an adaptation of procedure CHECK INCOMPATIBILITY proposed by Caprara and Toth [32] to test the incompatibility of all the items in a set in the two-dimensional vector packing context.

Procedure CHECK_INC receives as *input* a set S of items (with the corresponding w and h values), and the bin dimensions W and H, and gives as *output* a flag INC which is `true` if two items in S are incompatible, and, in such a case, the indices (in the original set) of these two items. The procedure is based on a simple observation and on an ordered data structure D. The observation is that given a pair of items $i, j \in MW$ (or $i, j \in MH$), if $w_i \leq w_j$ and $h_i \leq h_j$, then item j *dominates* item i, i.e., each item k incompatible with i is obviously incompatible with j. The data structure, D contains the subset $C \subseteq S$ of the non-dominated items of S. Initially, D contains the first element of S, and the items are iteratively added by preserving D ordered by decreasing values of w and by increasing values of h.

Thus, the procedure CHECK_INC is as follows:

```
procedure CHECK_INC(S,W,H,INC,k,ℓ);
begin
    insert the first item j ∈ S in D;
    S := S \ {j};
    for each j ∈ S do
    begin
        find the item s in D such that w_s is minimum and w_s > W − w_j;
        let s := 0 if no such item exists;
        if s ≠ 0 and h_s + h_j > H then
        begin
            INC := true;
            k := j;
            let ℓ be the index of item s in the original set;
            return
        end
        comment: item j is compatible with all the previous items: update D;
        find the first item s in D such that w_s < w_j;
        find the last item p in D such that w_p ≥ w_j;
        let s := 0 and p := 0 if the corresponding items do not exist;
        if p ≠ 0 and h_p < h_j then
        begin
            if s ≠ 0 and h_s > h_j then insert item j in D;
            else
                if s = 0 then insert item j in D;
                else
                    begin
                        delete item s;
                        insert item j in D
                    end
        end
```

```
        else    if p = 0 then insert item j in D;
    end
end.
```

The implementation of D by means of *red-black trees*, see, e.g., [45], requires $O(\log k)$ time both for searching for the first element s with w_s not greater than a given threshold, and for inserting/deleting an element, where k is the number of elements stored. Since the number of searches, insertions and deletions during the procedure is $O(n)$, the overall complexity is $O(n \log n)$, and the same holds for the overall algorithm 2BP_S for which testing the incompatibility of the items in a set is the bottleneck.

Theorem 6.1. *MSSP for 2BP compatibility graphs can be solved in $O(n \log n)$ time.*

Proof. It follows from the above discussion. □

An idea of the performance of the above bound is given by the results of Table 6.5. We consider the 500 instances already introduced in Chapter 2 (and used in the others), and we compare the new lower bound (L_{st} in Table 6.5), with the continuous lower bound (L_0), and the lower bounds L_1, L_2, and L_3 proposed by Martello and Vigo [119]. For each class we report in Table 6.5 the average value (over the 50 instances of the class) of the ratio of each lower bound over the best known upper bound (UB) computed with the constructive heuristics of Chapter 5.

Table 6.5: 2BP random instances by Martello and Vigo (Classes 1-4), and by Berkey and Wang (Classes 5-10). Comparison of lower bounds.

Class	$\frac{L_0}{UB}$	$\frac{L_1}{UB}$	$\frac{L_2}{UB}$	$\frac{L_3}{UB}$	$\frac{L_{st}}{UB}$
1	0.85	0.92	0.95	0.85	0.42
2	0.83	0.90	0.94	0.83	0.41
3	0.65	0.99	0.99	0.96	0.99
4	0.90	0.91	0.92	0.90	0.73
5	0.90	0.91	0.94	0.90	0.80
6	0.97	0.97	0.97	0.97	0.49
7	0.85	0.89	0.90	0.86	0.81
8	0.97	0.97	0.97	0.97	0.51
9	0.84	0.90	0.92	0.85	0.85
10	0.94	0.94	0.94	0.94	0.53

Not surprisingly, L_{st} is quite poor for all the considered classes, with the exception of Class 3 for which it has the best performances. In fact, for this class L_{st} obtains the highest value in 48 cases over 50 (instead of the 38 of both L_1 and L_2 which have anyway the same ratios). This behavior is due to the structure of the instances in this class for which w_j is uniformly random in $[\frac{1}{2}W, W]$ and h_j uniformly random in $[\frac{1}{2}H, H]$ with a probability of 70%.

6.2.2 The 3BP Case

The same discussion of Section 6.2 for 2BP can be repeated for three-dimensional bin packing by the obvious substitution of the compatibility test (6.3) with the following one:

$$d_i + d_j \leq D \quad \textbf{or} \quad w_i + w_j \leq W \quad \textbf{or} \quad h_i + h_j \leq H \tag{6.4}$$

where i, j are a pair of items, thus a pair of nodes of the compatibility graph $G = (V, E)$. Again, $(i, j) \in E$ iff (6.4) holds.

Also for 3BP, it is useful to partition the set of items. In particular, we obtain eight different sets:

- set M such that $d_i > \frac{1}{2}D$ **and** $w_i > \frac{1}{2}W$ **and** $h_i > \frac{1}{2}H \quad \forall i \in M$;
- set DW such that $d_i > \frac{1}{2}D$ **and** $w_i > \frac{1}{2}W$ **and** $h_i \leq \frac{1}{2}H \quad \forall i \in DW$;
- set DH such that $d_i > \frac{1}{2}D$ **and** $w_i \leq \frac{1}{2}W$ **and** $h_i > \frac{1}{2}H \quad \forall i \in DH$;
- set WH such that $d_i \leq \frac{1}{2}D$ **and** $w_i > \frac{1}{2}W$ **and** $h_i > \frac{1}{2}H \quad \forall i \in WH$;
- set MD such that $d_i > \frac{1}{2}D$ **and** $w_i \leq \frac{1}{2}W$ **and** $h_i \leq \frac{1}{2}H \quad \forall i \in MD$;
- set MW such that $d_i \leq \frac{1}{2}D$ **and** $w_i > \frac{1}{2}W$ **and** $h_i \leq \frac{1}{2}H \quad \forall i \in MW$;
- set MH such that $d_i \leq \frac{1}{2}D$ **and** $w_i \leq \frac{1}{2}W$ **and** $h_i > \frac{1}{2}H \quad \forall i \in MH$;
- set N such that $d_i \leq \frac{1}{2}D$ **and** $w_i \leq \frac{1}{2}W$ **and** $h_i \leq \frac{1}{2}H \quad \forall i \in N$.

Not surprisingly, Proposition 6.1. can be extended to 3BP. Formally, the following proposition holds.

Proposition 6.2. *Given any instance of 3BP, and the associated compatibility graph $G = (V, E)$, the maximum stable set G has cardinality $|M| + c$, with $c \in \{0, 1, 2, 3\}$.*

Proof. The fact that $c \geq 0$ is obvious because M is a stable set by construction. Consider now set $R = V \setminus M$: no subset $T \subseteq R$ of cardinality four (or more) exists such that each pair of items in T is incompatible. Take set DW. Any item $i \in DW$ could be incompatible with items in sets DH, WH and MH, but any item $j \in MH$ is obviously compatible with any item $k \in DH$ because both $w_j, w_k \leq \frac{1}{2}W$. The same holds for sets DH and WH, whereas any item in set MD could be incompatible only with an item in set WH (as MW with DH, and MH with DW). It follows that at most three items in R can belong to the maximum stable set, whose cardinality is therefore at most $|M| + 3$. □

We propose in the following an algorithm, which we call 3BP_Stable (3BP_S), for finding the maximum stable set in the 3BP context. By denoting again the stable set as SS, and by initializing it as $SS = M$, the algorithm exploits the proposition above, and needs to test three cases with a large number of subcases:

1. case $|M|+3$: it can be obtained only by adding to set SS a triple (i, j, k) of incompatible items such that $i \in DW$, $j \in DH$, and $k \in WH$, and pairwise incompatible with all the items in SS;
2. case $|M| + 2$: it can be obtained in two ways:

   (a) by removing an item of SS, and adding a triple as in case 1 above;
   (b) by adding a pair of items to M;

3. case $|M|+1$: it can be obtained in three ways:
   (a) by removing a pair of items of SS, and adding a triple as in case 1 above;
   (b) by removing an item of SS, and adding a pair as in case (2b) above;
   (c) by adding a single item to SS.

(It will be clear in the following that several subsubcases correspond to each of these subcases.)

As mentioned in Section 6.2.1, the initial partition can be done in $O(n)$ time, and, without additional effort, we are able to store the following information:

- for set M: $r = \arg\min_{k\in M} d_k$, $d_{min} = \min_{k\in M} d_k$, $s = \arg\min_{k\in M} w_k$, $w_{min} = \min_{k\in M} w_k$, $t = \arg\min_{k\in M} h_k$, $h_{min} = \min_{k\in M} h_k$, $r' = \arg\min_{k\in M\setminus\{r\}} d_k$, $d_{2min} = \min_{k\in M\setminus\{r\}} d_k$, $s' = \arg\min_{k\in M\setminus\{s\}} w_k$, $w_{2min} = \min_{k\in M\setminus\{s\}} w_k$, $t' = \arg\min_{k\in M\setminus\{t\}} h_k$, $h_{2min} = \min_{k\in M\setminus\{t\}} h_k$, $d_{3min} = \min_{k\in M\setminus\{r,r'\}} d_k$, $w_{3min} = \min_{k\in M\setminus\{s,s'\}} w_k$, and $h_{3min} = \min_{k\in M\setminus\{t,t'\}} h_k$;
- for set DW: $h_{max} = \max_{k\in DW} h_k$;
- for set DH: $w_{max} = \max_{k\in DH} w_k$;
- for set WH: $d_{max} = \max_{k\in WH} d_k$.

In addition, we need to define the sets of type S corresponding to the ones of Section 6.2.1. In particular:

- $S_1 \subseteq DW$ such that $h_i + h_t > H \quad \forall i \in S_1$;
- $S_2 \subseteq DH$ such that $w_i + w_s > W \quad \forall i \in S_2$;
- $S_3 \subseteq WH$ such that $d_i + d_r > D \quad \forall i \in S_3$;
- $S'_1 \subseteq DW$ such that $h_i + h_{t'} > H \quad \forall i \in S'_1$;
- $S'_2 \subseteq DH$ such that $w_i + w_{s'} > W \quad \forall i \in S'_2$;
- $S'_3 \subseteq WH$ such that $d_i + d_{r'} > D \quad \forall i \in S'_3$;
- $S''_1 \subseteq DW$ such that $h_i + h_{3min} > H \quad \forall i \in S''_1$;
- $S''_2 \subseteq DH$ such that $w_i + w_{3min} > W \quad \forall i \in S''_2$;
- $S''_3 \subseteq WH$ such that $d_i + d_{3min} > D \quad \forall i \in S''_3$;
- $S_4 \subseteq MD$ such that $h_i + h_t > H$ **and** $w_i + w_s > W \quad \forall i \in S_4$;
- $S_5 \subseteq MW$ such that $h_i + h_t > H$ **and** $d_i + d_r > D \quad \forall i \in S_5$;
- $S_6 \subseteq MH$ such that $w_i + w_s > W$ **and** $d_i + d_r > D \quad \forall i \in S_6$;
- $\bar{S}'_4 \subseteq MD$ such that $h_i + h_{t'} > H$ **and** $w_i + w_s > W \quad \forall i \in \bar{S}'_4$;

- $\tilde{S}'_4 \subseteq MD$ such that $h_i + h_t > H$ **and** $w_i + w_{s'} > W \quad \forall i \in \tilde{S}'_4$;
- $\bar{S}'_5 \subseteq MW$ such that $h_i + h_{t'} > H$ **and** $d_i + d_r > D \quad \forall i \in \bar{S}'_5$;
- $\tilde{S}'_5 \subseteq MW$ such that $h_i + h_t > H$ **and** $d_i + d_{r'} > D \quad \forall i \in \tilde{S}'_5$;
- $\bar{S}'_6 \subseteq MH$ such that $w_i + w_{s'} > W$ **and** $d_i + d_r > D \quad \forall i \in \bar{S}'_6$;
- $\tilde{S}'_6 \subseteq MH$ such that $w_i + w_s > W$ **and** $d_i + d_{r'} > D \quad \forall i \in \tilde{S}'_6$.

Finally, we discuss in detail the cases above and in particular at least one of the subcases (or subsubcases).

Case 1 The problem consists in finding a triple in the set $S_1 \bigcup S_2 \bigcup S_3$. It can be trivially solved in $O(n^3)$ time by simply trying each triple.

Case 2a The problem can be reduced to the previous one by removing item t from SS and testing the set $S'_1 \bigcup S_2 \bigcup S_3$, or item s and testing the set $S_1 \bigcup S'_2 \bigcup S_3$, or item r and testing the set $S_1 \bigcup S_2 \bigcup S'_3$.

Case 2b We define set $S_1 \bigcup S_2$. Since any item in this set is incompatible on the third dimension (the depth) both with the other items in the set and with any item in SS, the problem is to find a pair of incompatible items in the set, and can be solved in $O(n \log n)$ time by procedure CHECK_INC. The same holds for sets $S_1 \bigcup S_3$ and $S_2 \bigcup S_3$. Other sets for which the problem consists in finding a pair of incompatible items are $S_1 \bigcup S_6$, $S_2 \bigcup S_5$ and $S_3 \bigcup S_4$: for these sets, however, it is not possible to apply CHECK_INC because the three dimensions have to be tested. A trivial way of performing these tests requires $O(n^2)$ time (two nested loops).

Case 3a The problem is again the computation of a triple. The corresponding set is $S''_1 \bigcup S_2 \bigcup S_3$ and items t and t' are removed from SS. The same holds for five other sets: $S'_1 \bigcup S'_2 \bigcup S_3$, $S_1 \bigcup S''_2 \bigcup S_3$, $S_1 \bigcup S'_2 \bigcup S'_3$, $S'_1 \bigcup S_2 \bigcup S'_3$, and $S_1 \bigcup S_2 \bigcup S''_3$.

Case 3b This is the case where we remove an item from SS and we add a pair. The items that can be removed are again r, s, or t, and the following subcases arise. By removing r, we clearly define S'_3, hence we have to test $S'_3 \bigcup S_1$, $S'_3 \bigcup S_2$, and $S'_3 \bigcup S_4$. In addition, we also define $\tilde{S}'_5$ and $\tilde{S}'_6$, hence we need to test $\tilde{S}'_5 \bigcup S_2$ and $\tilde{S}'_6 \bigcup S_1$. As for Case 2b, some of these sets, namely $S'_3 \bigcup S_1$ and $S'_3 \bigcup S_2$, can be tested through CHECK_INC in $O(n \log n)$ time, whereas for the remaining ones $O(n^2)$ time is required. For the sake of completeness, these are the remaining ten subcases: $S'_2 \bigcup S_1$, $S'_2 \bigcup S_3$, $S'_2 \bigcup S_5$, $\tilde{S}'_4 \bigcup S_3$, and $\bar{S}'_6 \bigcup S_1$ (corresponding to the removing of s), and $S'_1 \bigcup S_2$, $S'_1 \bigcup S_3$, $S'_1 \bigcup S_6$, $\bar{S}'_4 \bigcup S_3$, and $\bar{S}'_5 \bigcup S_2$ (corresponding to the removing of t).

Case 3c We have seven subsubcases. For set DW we simply test in constant time condition $h_{max} + h_{min} > H$ (and analogously for DH and WH). For any item $i \in MD$ we test conditions $h_i + h_{min} > H \quad$ and $\quad w_i + w_{min} > W$ (and analogously for MW and MH) with an overall complexity of $O(n)$. Finally, for any item $i \in N$ we test conditions $h_i + h_{min} > H \quad$ and $\quad w_i + w_{min} > W \quad$ and $\quad d_i + d_{min} > D$, again in $O(n)$ time.

We do not report the overall algorithm but the idea is quite simple. Algorithm 3BP_S tests one at a time each of the described cases, and stops as soon as in one of these it is able to improve the initial stable set $SS = M$. It is easy to see that by exploring the cases in this order no additional improvement is possible. The actual bottleneck of the computation is the recognition of a triple, which determines the overall time complexity of 3BP_SP which is $O(n^3)$.

Theorem 6.2. *MSSP for 3BP compatibility graphs can be solved in $O(n^3)$ time.*

Proof. It follows from the above discussion. □

6.3 Two-Dimensional Shelf Packing Problems

In two-dimensional packing problems one is given a set of n rectangular *items*, each having *width* w_j and *height* h_j $(j = 1, \ldots, n)$, and the object is to orthogonally allocate them, without overlapping, to rectangular containers by minimizing the unused space. It is assumed that the items have fixed orientation, i.e., they cannot be rotated. Two main specific problems are considered in the literature:

- the *two-dimensional bin packing problem* (2BP), where an infinite number of identical finite containers (*bins*), having width W and height H, is available, and the object is to minimize the number of bins used.
- the *two-dimensional strip packing problem* (2SP), where a single container (*strip*), having width W and infinite height, is available, and the object is to minimize the height to which the strip is used.

These problems have industrial applications in cutting and packing contexts. The reader is referred to Lodi, Martello and Vigo [110] for a recent survey on two-dimensional packing, and to Dyckhoff, Scheithauer and Terno [64] for an annotated bibliography on cutting and packing.

Both 2BP and 2SP can be viewed as generalizations of the *one-dimensional bin packing problem* (1BP), where n elements, having size w_j $(j = 1, \ldots, n)$, have to be partitioned into the minimum number of subsets so that the sum of the sizes in each subset does not exceed a given capacity W. It is known that 1BP is strongly NP-hard. Given any instance of 1BP, we can construct both an equivalent instance of 2BP, by defining $h_j = H$ for $j = 1, \ldots, n$, and an equivalent instance of 2SP, by defining $h_j = 1$ for $j = 1, \ldots, n$. It follows that 2BP and 2SP are strongly NP-hard.

Most of the approximation algorithms for 2BP and 2SP (see, e.g., Coffman, Garey, Johnson and Tarjan [43], Chung, Garey and Johnson [42], Berkey and Wang [23] Frenk and Galambos [76], Lodi, Martello and Vigo [109]) find a bin/strip solution by packing the items, from left to right, in rows forming levels (*shelves*). The first shelf is the bottom of the bin/strip, and subsequent shelves are created on the horizontal line coinciding with the top of the tallest item packed on the shelf below. This kind of packing has also practical relevance: in most cutting applications it is required that the patterns are such that the items can be obtained through a sequence of edge-to-edge cuts parallel to the edges of the bin (*guillotine* cuts), and it is easily seen that shelf packings fulfil this constraint.

In this section we consider 2BP and 2SP with the additional constraint that the items are packed by shelves. We denote the resulting problems as *two-dimensional shelf bin/strip packing problem* (2SBP/2SSP).

To our knowledge, the mathematical models presented in the literature for two-dimensional cutting and packing problems (see, e.g., Beasley [18], Hadjiconstantinou and Christofides [89] Hifi [94]), are all based on a discrete representation of the geometrical space and the explicit use of coordinates at which items may be allocated. As a consequence these models require a huge, non-polynomial, number of binary variables, such as, for example, $x_{ipq} = 1$ iff item i is placed with its bottom left corner at coordinate (p, q) $(i = 1, \ldots, n,\ p = 1, \ldots, W,\ q = 1, \ldots, H$. In the next sections we show that the shelf restriction can be exploited so as to obtain mathematical models involving a polynomial number of variables. The quality of the models is evaluated through computational experiments.

We assume in the following that all input data are positive integers.

6.3.1 Mathematical Models

We start with a couple of simple observations, which will allow to obtain a more compact formulation. For any optimal shelf solution there exists an equivalent solution in which

(i) the first (leftmost) item packed in each shelf is the tallest item in the shelf;

(ii) the first (bottom) shelf packed in each bin/strip is the tallest shelf in the bin/strip.

Hence we will consider only solutions satisfying these conditions. If an item is the first in a shelf, we will say that it *initializes* the shelf; similarly, if a shelf is the first in a bin/strip, we will say that it *initializes* the bin/strip. We will also assume that

(iii) the items are sorted so that $h_1 \geq h_2 \geq \ldots \geq h_n$.

Let us first consider the two-dimensional shelf bin packing problem 2SBP. Our model uses four sets of variables: the first two sets refer to the packing of items into shelves, the remaining two to the packing of shelves into bins. The model assumes that n potential shelves are available, each associated with a different item i which initializes it, hence having the corresponding height h_i. The first decision variable is thus

$$y_i = \begin{cases} 1 & \text{if item } i \text{ initializes shelf } i \\ 0 & \text{otherwise} \end{cases} \qquad (i = 1, \ldots, n) \tag{6.5}$$

and observe that, by (i) and (iii) above, only items j satisfying $j > i$ may be packed in shelf i (if this shelf is actually used). Therefore the item packing is modeled by

$$x_{ij} = \begin{cases} 1 & \text{if item } j \text{ is packed into shelf } i \\ 0 & \text{otherwise} \end{cases} \qquad (i = 1, \ldots, n-1;\ j > i) \tag{6.6}$$

Similarly, we assume that n potential bins are available, each associated with a potential shelf k which initializes it. The third decision variable is thus

$$q_k = \begin{cases} 1 & \text{if shelf } k \text{ initializes bin } k \\ 0 & \text{otherwise} \end{cases} \qquad (k = 1, \ldots, n) \tag{6.7}$$

and observe that, by (ii) and (iii) above, only shelves i satisfying $i > k$ may be allocated to bin k (if this bin is actually used). Therefore the shelf packing is modeled by

$$z_{ki} = \begin{cases} 1 & \text{if shelf } i \text{ is allocated to bin } k \\ 0 & \text{otherwise} \end{cases} \qquad (k = 1, \dots, n-1; i > k) \tag{6.8}$$

The model follows:

$$\min \quad \sum_{k=1}^{n} q_k \tag{6.9}$$

$$\text{subject to} \quad \sum_{i=1}^{j-1} x_{ij} + y_j = 1 \qquad (j = 1, \dots, n) \tag{6.10}$$

$$\sum_{j=i+1}^{n} w_j x_{ij} \leq (W - w_i) y_i \qquad (i = 1, \dots, n-1) \tag{6.11}$$

$$\sum_{k=1}^{i-1} z_{ki} + q_i = y_i \qquad (i = 1, \dots, n) \tag{6.12}$$

$$\sum_{i=k+1}^{n} h_i z_{ki} \leq (H - h_k) q_k \qquad (k = 1, \dots, n-1) \tag{6.13}$$

$$y_i \in \{0, 1\} \qquad (i = 1, \dots, n) \tag{6.14}$$

$$x_{ij} \in \{0, 1\} \qquad (i = 1, \dots, n-1;\ j > i) \tag{6.15}$$

$$q_k \in \{0, 1\} \qquad (k = 1, \dots, n) \tag{6.16}$$

$$z_{ki} \in \{0, 1\} \qquad (k = 1, \dots, n-1;\ i > k) \tag{6.17}$$

The objective function (6.9) minimizes the number of bins used. Equations (6.10) impose that each item is packed exactly once, either by initializing a shelf or in a shelf initialized by a preceding (taller) item. Equations (6.11) impose the width constraint to each used shelf. Equations (6.12) impose that each used shelf is allocated exactly once, either by initializing a bin or in a bin initialized by a preceding (taller) shelf. Finally, equations (6.13) impose the height constraint to each used bin.

A couple of observations can be done on this model. First, from (6.10) and (6.12) we immediately have $y_1 = q_1 = 1$. Second, an equivalent model could replace variables y_i $(i = 1, \dots, n)$ with x_{ii} $(i = 1, \dots, n)$, and variables q_k $(k = 1, \dots, n)$ with z_{kk} $(k = 1, \dots, n)$. In both cases we feel that the given model has to be preferred for the sake of clarity.

It is easy to derive from (6.9)–(6.17) a mathematical model for the two-dimensional shelf strip packing problem 2SSP. Indeed, it is enough to introduce the appropriate objective function, and to drop all variables and constraints related to the packing of the shelves into the bins. We get

$$
\begin{aligned}
(6.18) \qquad & \min \quad \sum_{i=1}^{n} h_i y_i \\
& \text{subject to} \quad (6.10),\ (6.11),\ (6.14),\ (6.15)
\end{aligned}
$$

6.3.2 Computational Experiments

In this section the models of 2SBP and 2SSP are evaluated by using the branch-and-bound of Cplex 6.5 on the set of 500 two-dimensional instances already used in many points of this thesis. In particular, in the 2SSP case we consider only the width of the given bin to adapt each instance to be of strip packing type (thus obtaining a strip of width W and infinite height).

In Table 6.6 we consider the 2SBP on the four classes of instances proposed by Martello and Vigo [119] (Classes 1-4), and on the six classes proposed by Berkey and Wang [23] (Classes 5-10).

For each class and for each value of $n \in \{20, 40, 60, 80, 100\}$ we report for the continuous lower bound (L_0, computed as usual as $\lceil \sum_{i=1}^{n} w_i h_i/(WH) \rceil$, and obviously also valid for 2SBP) and for the continuous relaxation of the model (L_c) the average ratios (lower bound/best known solution), computed over ten instances (the best known solution is indicated as UB and is obtained by the branch-and-bound). In addition, we give the average computing times on a Digital Alpha 533 MHz of the standard branch-and-bound of Cplex 6.5, the number of problems solved to optimality within the time limit of 300 CPU seconds, and the average number of branch-and-bound nodes.

The behavior of the model is satisfactory: the performance of its continuous relaxation is better than that of the simple bound L_0 in all the 500 instances, and we are able to solve to optimality more than 80 % of the problems, by the standard branch-and-bound of Cplex 6.5 within a time limit of 300 CPU seconds (Digital Alpha 533 MHz.).

The same information is finally given for 2SSP in Table 6.7. In this case the continuous lower bound (L_0) is computed as $\lceil \sum_{i=1}^{n} w_i h_i/W \rceil$.

In this case too the proposed model obtains satisfactory results. In particular, we are able to solve to optimality 65 % of the 500 instances (always using the standard branch-and-bound of Cplex 6.5, and with a time limit of 300 CPU seconds). However, the behavior is much more dependent on the class of problems considered.

Table 6.6: 2SBP, random problem instances proposed by Martello and Vigo (Classes 1-4), and by Berkey and Wang (Classes 5-10).

Class	n	$\frac{L_0}{UB}$	$\frac{L_c}{UB}$	$TimeB\&B$	$Solved$	$Nodes$
	20	0.83	0.87	0.13	10	8.9
	40	0.84	0.86	1.79	10	44.8
1	60	0.87	0.87	30.35	10	606.7
	80	0.84	0.85	120.24	7	1682.8
	100	0.85	0.85	186.60	5	1101.1
	20	0.79	0.82	0.13	10	7.7
	40	0.84	0.86	5.05	10	158.4
2	60	0.86	0.87	51.62	9	997.4
	80	0.84	0.85	200.41	4	2010.8
	100	0.84	0.84	222.41	3	1073.0
	20	0.66	0.66	0.01	10	1.0
	40	0.65	0.66	0.07	10	1.6
3	60	0.63	0.64	0.12	10	1.0
	80	0.64	0.65	0.23	10	1.0
	100	0.65	0.65	0.38	10	1.0
	20	0.84	0.93	0.41	10	5.7
	40	0.90	0.93	13.54	10	206.6
4	60	0.87	0.92	130.37	6	1073.2
	80	0.89	0.90	193.98	5	865.5
	100	0.88	0.89	278.74	1	366.7
	20	0.88	0.88	0.11	10	2.8
	40	0.88	0.89	1.33	10	30.5
5	60	0.91	0.91	2.98	10	20.7
	80	0.91	0.91	11.01	10	132.0
	100	0.94	0.94	59.15	10	496.3
	20	1.00	1.00	0.23	10	13.3
	40	0.95	1.00	3.52	10	76.4
6	60	0.90	0.90	63.15	9	1566.5
	80	0.90	0.90	155.70	7	2076.4
	100	0.88	0.88	194.94	5	1458.3
	20	0.83	0.83	0.21	10	1.0
	40	0.84	0.86	2.87	10	21.6
7	60	0.90	0.91	36.78	10	279.8
	80	0.88	0.88	52.06	9	437.9
	100	0.88	0.89	207.85	4	484.3
	20	1.00	1.00	0.25	10	13.1
	40	0.95	0.95	3.90	10	60.0
8	60	0.83	0.90	126.24	7	2322.8
	80	0.80	0.83	188.68	5	2245.2
	100	0.87	0.87	174.74	5	1236.9
	20	0.81	0.83	0.17	10	1.0
	40	0.82	0.85	2.61	10	43.0
9	60	0.86	0.87	16.99	10	152.5
	80	0.86	0.86	33.70	10	318.2
	100	0.88	0.89	203.12	5	747.6
	20	1.00	1.00	0.18	10	10.6
	40	0.80	0.95	3.66	10	43.9
10	60	0.87	0.87	102.44	7	1726.2
	80	0.93	0.93	115.10	7	1222.1
	100	0.81	0.81	199.40	4	1327.7

Table 6.7: 2SSP, strip packing adaptation of the random problem instances proposed by Martello and Vigo (Classes 1-4), and by Berkey and Wang (Classes 5-10).

Class	n	$\frac{L_0}{UB}$	$\frac{L_c}{UB}$	$TimeB\&B$	$Solved$	$Nodes$
	20	0.82	0.86	0.01	10	2.0
	40	0.83	0.85	0.05	10	7.1
1	60	0.85	0.86	0.42	10	38.7
	80	0.85	0.86	0.63	10	45.0
	100	0.86	0.87	0.81	10	39.6
	20	0.87	0.90	17.40	10	9188.6
	40	0.92	0.93	166.08	5	14064.2
2	60	0.93	0.94	300.06	0	11535.0
	80	0.94	0.94	300.10	0	6363.5
	100	0.94	0.95	300.17	0	3827.1
	20	0.80	0.82	0.01	10	1.6
	40	0.80	0.81	0.03	10	2.1
3	60	0.79	0.80	0.06	10	2.0
	80	0.80	0.80	0.12	10	1.5
	100	0.82	0.82	0.33	10	7.9
	20	0.85	0.88	0.87	10	218.3
	40	0.89	0.92	33.68	10	2765.6
4	60	0.91	0.92	220.51	5	9865.6
	80	0.92	0.93	239.66	3	5531.2
	100	0.93	0.94	300.13	0	3514.4
	20	0.88	0.91	0.09	10	34.0
	40	0.91	0.92	1.09	10	265.4
5	60	0.92	0.93	31.17	9	5461.6
	80	0.93	0.93	31.43	9	7522.7
	100	0.96	0.96	91.05	8	5601.7
	20	0.81	0.89	0.74	10	193.8
	40	0.89	0.91	220.06	4	35329.0
6	60	0.91	0.92	300.04	0	20581.5
	80	0.92	0.93	300.07	0	15579.2
	100	0.93	0.93	300.11	0	11971.1
	20	0.84	0.88	0.14	10	31.2
	40	0.88	0.89	2.41	10	245.5
7	60	0.90	0.91	33.98	9	1265.9
	80	0.90	0.91	10.75	10	378.1
	100	0.92	0.92	144.97	6	2891.6
	20	0.79	0.89	0.87	10	112.8
	40	0.87	0.91	167.25	6	17464.2
8	60	0.89	0.91	300.05	0	14235.7
	80	0.90	0.91	300.10	0	7711.6
	100	0.90	0.91	300.13	0	5783.4
	20	0.84	0.87	0.11	10	16.8
	40	0.87	0.88	0.97	10	75.6
9	60	0.89	0.90	12.42	10	761.3
	80	0.89	0.90	23.57	10	1231.2
	100	0.92	0.93	136.12	6	2658.1
	20	0.77	0.88	0.88	10	120.2
	40	0.86	0.90	197.56	4	21029.3
10	60	0.88	0.91	300.04	0	11562.9
	80	0.89	0.91	300.08	0	8007.6
	100	0.89	0.90	300.13	0	5630.5

Part II

Algorithms for Assignment Problems

Chapter 7

Preliminaries and Outline of Part II

The well-known *Linear Assignment Problem* (AP) (see [55] for an annotated bibliography) states as follows. Given a square cost matrix c_{ij} of order n, the problem is to assign to each row a different column, and vice-versa in order to minimize the total sum of the row-column assignment costs.

This problem can be seen as the *Minimum Cost Perfect Matching* problem on bipartite graphs. Let $G = (V \cup T, A)$ be a bipartite graph where V and T are the vertex sets such that $|V| = |T| = n$, $A = \{(i,j)|i \in V, j \in T\}$ is the arc set, and c_{ij} is the cost of arc $(i,j) \in A$. Then, the minimum cost perfect matching in G is to select n arcs in A for which no two arcs share a vertex and the sum of the corresponding costs is a minimum. Vertex $i \in V$ corresponds to row i and vertex $j \in T$ to column j, thus the optimal solution of matching gives the solution to the AP in which, if arc $(i,j) \in A$ is selected, row i is assigned to column j.

A classic Integer Linear Programming (ILP) formulation for the AP is:

$$Z(AP) = \min \sum_{i \in V} \sum_{j \in T} c_{ij}\, x_{ij} \tag{7.1}$$

subject to

$$\sum_{i \in V} x_{ij} = 1, \qquad j \in T \tag{7.2}$$

$$\sum_{j \in T} x_{ij} = 1, \qquad i \in V \tag{7.3}$$

$$x_{ij} \in \{0,1\} \qquad i \in V, j \in T \tag{7.4}$$

where $x_{ij} = 1$ if and only if row i is assigned to column j in the optimal solution. Constraints (7.2) (resp. (7.3)) assure that each row (resp. column) is assigned to exactly one column (resp. row).

Alternatively, AP can also be defined on a digraph (of n vertices) as the graph theory problem of finding a set of *disjoint* sub-tours such that all the vertices in the digraph are visited and the sum of the costs of selected arcs is a minimum. Through this point of view constraints (7.2) and (7.3) impose in-degree and out-degree of each vertex equal to one.

It is well-known that, since the constraint matrix of AP is totally unimodular, the problem is polynomially solvable. Thus, an (integer) optimal solution of AP can be obtained by applying any linear programming algorithm to the continuous relaxation of the above model.

However, AP can be solved through special purpose algorithms, as for example the Hungarian Algorithm (see, Lawler [103]) with time complexity $O(n^3)$.

The great interest of the AP for combinatorial optimization is due to several reasons. The problem itself models real-world applications arising in a number of different domains like scheduling, location and routing, just to mention a few. In addition, many other problems, with both theoretical and practical relevance, contain AP as a subproblem (e.g., the most famous one is the *Travelling Salesman Problem*), thus solving AP provides a bound for the solution of such problems. Finally, AP is strictly related to linear programming, thus being a perfect "playground" to apply basic concepts like LP duality and many others.

Many of the projects I have been involved in during the PhD period have the assignment problem as a common denominator:

- In Chapter 8, we consider a generalization of AP, called *k-Cardinality Assignment Problem*, in which only a subset (with cardinality equal to k) of the rows has to be assigned. For this problem specialized algorithms and the corresponding computer implementations are presented.

- In Chapter 9, a new tabu search framework, called *eXploring Tabu Search*, is presented, and its effectiveness is computationally tested on another generalization of AP obtained by replacing the objective function (7.1) with a cumulative function (thus the name *Cumulative Assignment Problem*, CAP).

- In Chapter 10, AP is used as a bound in Constraint Programming (CP). In particular, the Hungarian algorithm is integrated into two CP global constraints (`IlcAllDiff` and `IlcPath`), thus adding to the constraints an *optimization component*. CP global constraints usually prune the solution space by feasibility reasoning, whereas the addition of AP allows also a pruning based on costs. The mechanism is perfectly general, and AP is used as an example, again very clever. In addition, the bound inside the constraint is used as a *black box* by the user (which is an important feature for CP).

- In Chapter 11, a branch-and-cut algorithm for the *Multiple Depot Vehicle Scheduling Problem* is presented. The initial lower bound, as well as the pricing strategy require the iterative solution of a third generalization of AP, called *Transportation Problem*, in which some of the vertices have in-degree and out-degree higher than one (see, equations (7.2)-(7.3)).

Finally, Chapter 12 presents algorithms for a pair of problems which belong to the wide category of assignment problems even if they are not directly related to the Linear Assignment Problem. In particular, a metahueristic algorithm for the *unconstrained Quadratic 0–1 Programming* (QP) is considered, whereas exact and heuristic approaches have been developed for the *Data sets Reconstruction Problem* (DRP).

Both QP and DRP can be seen as graph theory problems, and they have been included in this thesis because the algorithmic components are common to the ones used in other chapters for different problems, thus representing an interesting appendix to prove the effectiveness of these ideas.

Chapter 8

The k-Cardinality Assignment Problem

8.1 Introduction

A[1] well-known problem in telecommunications is the *Satellite-Switched Time-Division Multiple Access* (SS/TDMA) time slot assignment problem (see, e.g., Prins [134], Jain, Werth and Browne [96]). We are given m transmitting stations and n receiving stations, interconnected by a geostationary satellite through an on-board $k \times k$ switch ($k \leq \min(m, n)$), and an integer $m \times n$ *traffic matrix* $W = [w_{ij}]$, where w_{ij} is the time needed for the transmission from station i to station j. The problem is to determine a sequence of switching configurations which allow all transmissions to be performed in minimum total time, under the constraint that no preemption of a transmission occurs. A *switching configuration* is an $m \times n$ 0-1 matrix X^ℓ having exactly k ones, with no two in the same row or column. Matrix X^ℓ is associated with a transmission time slot of duration $t^\ell = \max_{ij}\{X^\ell_{ij} w_{ij}\}$. The objective is thus to determine an integer τ and X^ℓ ($\ell = 1, \ldots, \tau$) so that $\sum_{\ell=1}^{\tau} t^\ell$ is a minimum. The problem is known to be NP-hard. Several heuristic algorithms (see, e.g., Pomalaza-Ráez [133] and Dell'Amico, Maffioli and Trubian [53]) determine, at each iteration, a switching configuration which minimizes the sum of the selected transmission times. In the following we consider efficient algorithms for exactly solving this local problem.

Given an integer $m \times n$ cost matrix W and an integer k ($k \leq \min(m, n)$), the *k–Cardinality Assignment Problem* (k-AP) is to select k rows and k columns of W and to assign each selected row to exactly one selected column, so that the sum of the costs of the assignment is a minimum. Let x_{ij} ($i = 1, \ldots, m$, $j = 1, \ldots, n$) be a binary variable taking the value 1 if and only if row i is assigned to column j. The problem can then be formulated as the following integer linear program:

$$z = \min \sum_{i=1}^{m} \sum_{j=1}^{n} w_{ij} x_{ij} \tag{8.1}$$

$$\sum_{j=1}^{n} x_{ij} \leq 1, \quad (i = 1, \ldots, m) \tag{8.2}$$

[1] The results in this chapter appear in: M. Dell'Amico, A. Lodi, S. Martello, "Efficient Algorithms and Codes for k-Cardinality Assignment Problems", *Technical Report* OR/97/7, DEIS - Università di Bologna, *Discrete Applied Mathematics* to appear, [51].

$$\sum_{i=1}^{m} x_{ij} \leq 1, \quad (j = 1, \ldots, n) \tag{8.3}$$

$$\sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} = k \tag{8.4}$$

$$x_{ij} \in \{0, 1\} \quad (i = 1, \ldots, m, \ j = 1, \ldots, n) \tag{8.5}$$

In the special case where $m = n = k$, k-AP reduces to the well-known *Assignment Problem* (AP). (In this case the '$\leq$' sign in equations (8.2) and (8.3) can be replaced by the '=' sign, and equation (8.4) can be dropped.)

It has been shown by Dell'Amico and Martello [54] that the constraint matrix of k-AP is totally unimodular, hence the polyhedron defined by the continuous relaxation of (8.1)–(8.5) has integral vertices. Thus the optimal solution to the problem can be obtained in polynomial time through linear programming.

It is well known that AP is to find the minimum cost basis of the intersection of two partition matroids, hence k-AP is the optimum basis of the intersection of two matroids truncated to k. Since the two-matroid intersection algorithm runs in polynomial time, we know that k-AP is polynomial even for real valued cost matrices. This also gives a possible dual algorithm for k-AP: start with an empty solution (no row assigned) and apply for k times a *shortest augmenting path* technique (see, e.g., Lawler [103]) obtaining, at each iteration, a solution in which one more row is assigned. Another possible approach consists in solving a min-cost flow problem on an appropriate network (see Section 8.5).

In addition to the SS/TDMA context, k-AP has immediate applications in assigning workers to machines when there are multiple alternatives and only a subset of workers and machines has to be assigned. Although extensive literature exists on AP and related problems (surveys can be found, e.g., in Martello and Toth [117], Ahuja, Magnanti and Orlin [3], Dell'Amico and Martello [55]), to our knowledge the only specific result on k-AP has been presented by Dell'Amico and Martello [54].

The algorithm in [54] was developed for solving instances in which matrix W is complete. In this chapter we consider the case where W is sparse, i.e., many entries are empty, or, equivalently, they have infinite cost. The sparse case is relevant for applications. Consider, e.g., the SS/TDA time slot assignment previously discussed: it is clear that, in general, a transmitting station will not send messages to *all* receiving stations, so we can expect that real-world traffic matrices are sparse.

In the sparse case the problem is more conveniently formulated through the following graph theory model. Let $G = (U \cup V, A)$ be a bipartite digraph where $U = \{1, \ldots, m\}$ and $V = \{1, \ldots, n\}$ are the two vertex sets and $A \subseteq U \times V$ is the arc set: $A = \{(i, j) : w_{ij} < +\infty\}$. Let w_{ij} be the weight of an arc $(i, j) \in A$. We want to select k arcs of A such that no two arcs share a common vertex and the sum of the costs of the selected arcs is a minimum.

In the following we assume, without loss of generality, that $w_{ij} \geq 0 \ \forall \ (i, j) \in A$. Since k-AP does not change if we swap sets U and V, we also assume, without loss of generality, that $m \leq n$.

The problem in which the objective is to maximize the cost of the assignment, is solved by k-AP if each cost w_{hl} $((h, l) \in A)$ is replaced by the non-negative quantity $\widetilde{w}_{hl} = \max_{(i,j)\in A}\{w_{ij}\} - w_{hl}$.

The aim of this article is to provide a specialized algorithm, and the corresponding computer code, for the exact solution of k-AP.

8.2 Complete Matrices

The main phases of the approach in [54] are a heuristic preprocessing and an exact primal algorithm. Preprocessing finds a partial optimal solution, determines sets of rows and columns which must be assigned in an optimal solution and reduces the cost matrix. The partial solution is then heuristically completed, and the primal algorithm determines the optimum through a series of alternating paths.

The preprocessing phase can be summarized as follows.

Step 0. For each row i determine the first, second and third smallest cost in the row ($\rho_i, \rho'_i, \rho''_i$, respectively) and let $c(i), c'(i), c''(i)$ be the corresponding columns. For each column j determine the first, second and third smallest cost in the column ($\gamma_j, \gamma'_j, \gamma''_j$, respectively) and let $r(j), r'(j), r''(j)$ be the corresponding rows. Reorder the columns of W by nondecreasing γ_j values, and the rows by nondecreasing ρ_i values.

Step 1. Determine a value $g_1 \leq k$, as high as possible, such that a partial optimal solution in which g_1 rows are assigned can be heuristically determined. This is obtained through a heuristic algorithm which, for increasing values of j, assigns row $r(j)$ to column j if this row is currently unassigned; if instead row $r(j)$ is already assigned, under certain conditions the algorithm assigns column $j+1$ instead of j (and then proceeds to column $j+2$) or assigns the current column j to row $r'(j)$ instead of $r(j)$. If $g_1 = k$ then terminate (the solution is optimal); otherwise, execute a greedy heuristic to determine an approximate solution of cardinality k and let UB_1 be the corresponding value.

Step 2. Repeat Step 1 over the transposed cost matrix. Let g_2 and UB_2 be the values obtained.

Step 3. $UB := \min(UB_1, UB_2)$; $g := \max(g_1, g_2)$.

Step 4. Starting from the partial optimal solution of cardinality g, determine a set R of rows and a set C of columns that must be selected in an optimal solution to k-AP. Compute a lower bound LB on the optimal solution value.

Step 5. If $UB > LB$ then reduce the instance by determining pairs (i, j) for which x_{ij} must assume the value zero.

The overall time complexity of the above preprocessing phase is $O(mn + kn \log n)$.

Once Step 5 has been executed, the primal phase starts with the $k \times k$ submatrix of W induced by the sets of rows and columns selected in the heuristic solution of value UB, and exactly solves an AP over this matrix. Then $m + n - 2k$ iterations are performed by adding to the current submatrix one row or column of W at a time, and by re-optimizing the current solution through alternating paths on a particular digraph induced by the submatrix.

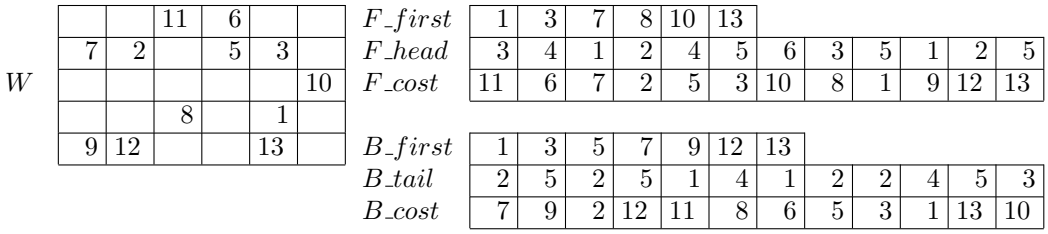
8.3 Sparse matrices

In this section we discuss the most relevant features of the algorithm for the sparse case.

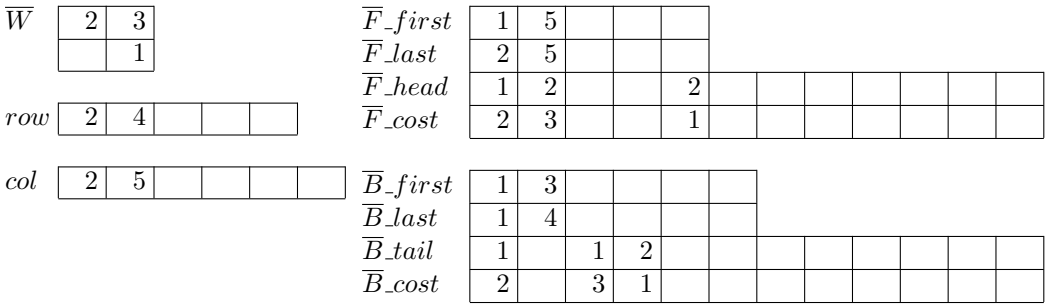
8.3.1 Data structures

Let μ be the number of entries with finite cost. We store matrix W through a *forward star* (see Figure 8.1(a)) consisting of an array, F_first, of $m+1$ pointers, and two arrays, F_head and F_cost, of μ elements each. The entries of row i, and the corresponding columns, are stored in F_cost and F_head, in locations $F\_first(i), \ldots, F\_first(i+1)-1$ (with $F\_first(1)=1$ and $F\_first(m+1)=\mu+1$). Since the algorithm needs to scan W both by row and by column, we duplicate the information into a *backward star*, similarly defined by arrays B_first (of $n+1$ pointers), B_tail and B_cost, storing the entries of column j, and the corresponding rows, in locations $B\_first(j), \ldots, B\_first(j+1)-1$.

In the primal phase a local matrix $\overline{W}$ is initialized to the $k \times k$ submatrix of W obtained from preprocessing, and iteratively enlarged by addition of a row or a column. Let $row(i)$ (resp. $col(j)$) denote the row (resp. column) of W corresponding to row i (resp. column j) of $\overline{W}$. Matrix $\overline{W}$ is stored through a modified forward star. In this case we have two arrays, $\overline{F}\_head$ and $\overline{F}\_cost$, of length μ, and two arrays, $\overline{F}\_first$ and $\overline{F}\_last$, of m pointers each. For each row i currently in $\overline{W}$: (a) the elements of the current columns, and the corresponding current column indices, are stored in $\overline{F}\_cost$ and $\overline{F}\_head$, in locations $\overline{F}\_first(i), \ldots, \overline{F}\_last(i)$; (b) locations $\overline{F}\_last(i)+1, \ldots, \overline{F}\_first(i)+L-1$ (where $L = F\_first(row(i)+1) - F\_first(row(i))$ denotes the number of elements in row $row(i)$ of W) are left empty for future insertions. The backward star for $\overline{W}$ is similarly defined by arrays $\overline{B}\_tail$, $\overline{B}\_cost$, $\overline{B}\_first$ and $\overline{B}\_last$. Figure 8.1(b) shows the structures for a 2×2 submatrix $\overline{W}$ of W.



(a) Input matrix.



(b) Local matrix.

Figure 8.1: Data structures.

8.3.2 Approximate solution

All computations needed by the preprocessing of Section 8.2 were efficiently implemented through the data structures described in the previous section, but the greedy solution of Step 1 (and 2) required a specific approach. Indeed, due to sparsity, in some cases a solution of cardinality k does not exist at all. Even if k-cardinality assignments exist, in many cases the procedure adopted for the complete case proved to be highly inefficient in finding one. For the same reason, a second greedy approach was needed at Step 3. These approaches are discussed in Section 8.3.3.

In certain cases the execution of the greedy algorithms can be replaced by the following transformation which provides a method to determine a k-cardinality assignment, or to prove that no such assignment exists. Let us consider the bipartite digraph $G = (U \cup V, A)$ introduced in Section 8.1, and define a digraph $G' = (\{s\} \cup \{\overline{s}\} \cup U \cup V \cup \{t\}, A')$, where $A' = A \cup \{(s, \overline{s})\} \cup \{(\overline{s}, i) : i \in U\} \cup \{(j, t) : j \in V\}$. Now assign unit capacity to all arcs of A' except $(s, \overline{s})$, for which the capacity is set to k, and send the maximum flow ϑ from s to t. It is clear that this will either provide a k-cardinality assignment or prove that no such assignment is possible. If $\vartheta < k$, we know that no solution to k-AP exists. If $\vartheta = k$, the arcs of A carrying flow provide a k-cardinality assignment, which is suboptimal since the arc costs have been disregarded.

Computational experiments proved that the above approach is only efficient when the instance is very sparse, i.e., $|A| \leq 0.01\ mn$, and $k > 0.9\ m$. For different cases, much better performance was obtained through the following greedy approaches.

8.3.3 Greedy approximate solutions

We recall that at steps 1 and 2 of the preprocessing phase described in Section 8.2 we have partial optimal solutions of cardinality g_1 and g_2, respectively. Starting from one of these solutions, the following algorithm tries to obtain an approximate solution of cardinality k by using the information computed at Step 0 (the first, second and third smallest cost in each row and column). In a first phase (steps a. and b. below) we try to compute the solution using only first minima. If this fails, we try using second and third minima (step c.). Parameter g_h can be either g_1 or g_2.

Algorithm GREEDY(g_h):

a. $\overline{g}_h := g_h$;
find the first unassigned row, i, such that column $c(i)$ is unassigned and set $\widetilde{\rho} := \rho_i$ ($\widetilde{\rho} := +\infty$ if no such row);
find the first unassigned column, j, such that row $r(j)$ is unassigned and set $\widetilde{\gamma} := \gamma_j$ ($\widetilde{\gamma} := +\infty$ if no such column);

b. **while** $\overline{g}_h < k$ **and** ($\widetilde{\rho} \neq +\infty$ **or** $\widetilde{\gamma} \neq +\infty$) **do**
$\overline{g}_h := \overline{g}_h + 1$;
if $\widetilde{\rho} \leq \widetilde{\gamma}$ **then**
assign row i to column $c(i)$;
find the next unassigned row, i, such that column $c(i)$ is unassigned
and set $\widetilde{\rho} := \rho_i$ ($\widetilde{\rho} := +\infty$ if no such row);
else

assign column j to row $r(j)$;
find the next unassigned column, j, such that row $r(j)$ is unassigned
and set $\widetilde{\gamma} := \gamma_j$ ($\widetilde{\gamma} := +\infty$ if no such column)
endif
endwhile;

c. **if** $\overline{g}_h < k$ **then**
execute steps similar to a. and b. above, but looking, at each search,
for the first or next row i (resp. column j) such that column $c'(i)$ or $c''(i)$
(resp. row $r'(j)$ or $r''(j)$) is unassigned, and setting
$\widetilde{\rho} := \rho'(i)$ if $c'(i)$ is unassigned, or $\widetilde{\rho} := \rho''(i)$ otherwise
(resp. $\widetilde{\gamma} := \gamma'(j)$ if $r'(j)$ is unassigned, or $\widetilde{\gamma} := \gamma''(j)$ otherwise).

d. **if** $\overline{g}_h = k$ **then** $UB_h :=$ solution value **else** $UB_h := +\infty$;
return $\overline{g}_h$ and UB_h.

Since it is possible that both $\overline{g}_1$ and $\overline{g}_2$ are less than k, Step 3 of Section 8.2 was modified so as to have in any case an approximate solution of cardinality k, either through elements other than the first three minima, or, if this fails, through dummy elements. The resulting Step 3 is as follows.

Step 3. $UB := \min(UB_1, UB_2)$; $g := \max(g_1, g_2)$; $\overline{g} := \max(\overline{g}_1, \overline{g}_2)$;
if $\overline{g} = k$ **then go to** Step 4;
for each unassigned column j **do**
if $\exists\ w_{ij}$ such that i is unassigned **then**
assign row i to column j, update UB and set $\overline{g} := \overline{g} + 1$;
if $\overline{g} = k$ **then go to** Step 4
endif;
while $\overline{g} < k$ **do**
let i be the first unassigned row and j the first unassigned column;
add to W a dummy element $w_{ij} = M$ (a very large value);
assign row i to column j, update UB and set $\overline{g} := \overline{g} + 1$
endwhile.

8.4 Implementation

The overall algorithm for sparse matrices was coded in Fortran 77.

The approximate solution of Section 8.3.2 was obtained using the FORTRAN implementation of the Dinic [59] algorithm proposed by Goldfarb and Grigoriadis [85].

The AP solution needed at the beginning of the primal phase was determined through a modified version of the Jonker and Volgenant [99] code SPJV for the sparse assignment problem. The original Pascal code was translated into FORTRAN and modified so as to: (i) handle our data structure for the current matrix $\overline{W}$ (see Section 8.3.1); (ii) manage the particular case arising when the new Step 3 (see Section 8.3.3) adds dummy elements to the cost matrix. These elements are not explicitly inserted in the data structure but implicitly considered by associating a flag to the rows having fictitious assignment.

The alternating paths needed, at each iteration of the primal phase, to re-optimize the current solution have been determined through the classical Dijkstra algorithm, implemented

with a binary heap for the node labels. Two shortest path routines were coded: RHEAP (resp. CHEAP) starts from a given row (resp. column) and uses the forward (resp. backward) data structure of Section 8.3.1. Both routines implicitly consider the possible dummy elements (see above).

The overall structure of the algorithm is the following.

Algorithm SKAP:
if $|A| \leq 0.01\ mn$ **and** $k > 0.9\ m$ **then**
  execute the Dinic algorithm (Section 8.3.2) and set $LB := -1$;
  if the resulting flow is less than k **then return** "no solution"
else
  perform the sparse version of Steps 0–2 of preprocessing (Section 8.2);
  execute the new Step 3 (Section 8.3.3) and let UB denote the resulting value;
  execute the sparse version of Step 4 and let LB denote the lower bound value;
  if $UB = LB$ **then return** the solution **else** reduce the instance (Step 5)
endif;
initialize the data structure for matrix $\overline{W}$ (Section 8.3.1);
execute the modified SPJV code for AP and let UB denote the solution value;
if $UB = LB$ **then return** the solution;
for $i := 1$ **to** $m - k$ **do**
  add a row to $\overline{W}$ and re-optimize by executing RHEAP;
  add a column to $\overline{W}$ and re-optimize by executing CHEAP;
endfor;
for $i := 1$ **to** $n - m$ **do**
  add a column to $\overline{W}$ and re-optimize by executing CHEAP;
if the solution includes dummy elements **then return** "no solution"
else return the solution.

8.4.1 Running the code

The algorithm was implemented as a FORTRAN 77 subroutine, called SKAP. The whole subroutine is completely self-contained and communication with it is achieved solely through the parameter list. The subroutine can be invoked with the statement

```
      CALL SKAP(M,N,F_FIRST,F_COST,F_HEAD,B_FIRST,B_COST,B_TAIL,K,
     +     OPT,Z,ROW,COL,U,V,MU,F_COST2,B_COST2,F_HEAD2,B_TAIL2)
```

The input parameters are:

`M, N` = number of rows and columns (m, n);

`F_FIRST, F_COST, F_HEAD` = forward star data structure (see Section 8.3.1): these arrays must be dimensioned at least at $m + 1, |A|$ and $|A|$, respectively;

`B_FIRST, B_COST, B_TAIL` = backward star data structure (see Section 8.3.1): these arrays must be dimensioned at least at $n + 1, |A|$ and $|A|$, respectively;

`K` = cardinality of the required solution;

The output parameters are:

`OPT` = return status: value 0 if the instance is infeasible; value 1 otherwise;

`Z` = optimal solution value;

`ROW` = the set of elements in the solution is {(`ROW(j),j`): `ROW(j)` $> 0, \mathtt{j} = 1, ..., \mathtt{N}$}: this array must be dimensioned at least at n;

`COL` = the set of entries in the solution is {(`i,COL(i)`): `COL(i)` $> 0, \mathtt{i} = 1, ..., \mathtt{M}$}: this array must be dimensioned at least at m;

Local arrays `U, V` (dimensioned at least at m and n, respectively) and scalar `MU` are internally used to store the dual variables (see, Dell'Amico and Martello [54]); local arrays `F_COST2, F_HEAD2, B_COST2` and `B_TAIL2` are internally used to store the restricted matrix (see Section 8.3.1), and must be dimensioned at least at $|A|$.

8.5 Computational Experiments

Algorithm SKAP was computationally tested on a Silicon Graphics Indy R10000sc 195 MHz.

We compared its performance with the following indirect approach. Consider digraph $G' = (\{s\} \cup \{\overline{s}\} \cup U \cup V \cup \{t\}, A')$ introduced in Section 8.3.2, and assign cost w_{ij} to each arc $(i, j) \in A$, and zero cost to the remaining arcs. It is then clear that sending a min-cost $s-t$ flow of value k on this network will give the optimal solution to the corresponding k-AP. This approach was implemented by using two different codes for min-cost flow. The first one is the FORTRAN code RELAXT IV of Bertsekas and Tseng, see [24], called RLXT in the sequel, and the second one is the C code CS2 developed by Cherkassky and Goldberg, see [84]. Parameter `CRASH` of RELAXT IV was set to one, since preliminary computational experiments showed that this value produces much smaller computing times for our instances.

Test problems were obtained by randomly generating the costs w_{ij} from the uniform distributions $[0, 10^2]$ and $[0, 10^5]$. This kind of generation is often encountered in the literature for the evaluation of algorithms for AP.

For increasing values of m and n (up to 3000) and various degrees of density (1%, 5% and 10%), ten instances were generated and solved for different values of k (ranging from $\frac{20}{100}m$ to $\left\lfloor \frac{99}{100}m \right\rfloor$).

Density d% means that the instance was generated as follows. For each pair (i, j), we first generate a uniformly random value r in $[0, 1)$: if $r \leq d$ then the entry is added to the instance by generating its value w_{ij} in the considered range; otherwise the next pair is considered. In order to ensure consistency with the values of m and n, we then consider each row i and, if it has no entry, we uniformly randomly generate a column j and a value w_{ij}. The same is finally done for columns.

Tables 8.1 to 8.3 show the results for sparse instances. In each block, each entry in the first six lines gives the average CPU time computed over ten random instances, while the last line gives the average over all k values considered. The results show that the best of the two min-cost flow approaches is faster than SKAP for some of the easy instances of Table 8.1, while both are clearly slower for all the remaining instances.

More specifically, for the 1% density (Table 8.1) RLXT is faster than SKAP in range $[0, 10^2]$ for m and n less than 3000, while CS2 is faster than SKAP in range $[0, 10^5]$ for $500 \leq m \leq 1000$ and $1000 \leq n \leq 2000$. It should be noted, however, that SKAP solves all these instances in less than one second on average, and that it is less sensitive to the cost range.

For the 5% instances (Table 8.2), SKAP is always the fastest code. For large values of m and n it is two-three times faster than its best competitor. This trend is confirmed by Table 8.3 (10% density), where, for large instances, the CPU times of SKAP are about one order of magnitude smaller than the others.

In Table 8.4 we consider complete matrices and compare the previous approaches and the specialized algorithm PRML, developed, for dense k-APs, by Dell'Amico and Martello [54]. The results show that SKAP is clearly superior to both min-cost flow approaches. In addition, in many cases, it is even faster than the specialized code. This is probably due to the new heuristic algorithm embedded in SKAP.

Tables 8.1-8.4 also give information about the relative efficiency of codes RELAXT IV and CS2, when used on our type of instances. It is quite evident that the two codes are sensitive to the cost range: RELAXT IV is clearly faster than CS2 for range $[0, 10^2]$, while the opposite holds for range $[0, 10^5]$.

Finally, Figures 8.2 and 8.3 show the behavior of the average CPU time requested by SKAP, for each value of k, as a function of the instance size. We consider the case of range $[0, 10^5]$ and 5% density, separately for rectangular and square instances. The functions show that SKAP has, for fixed k value, a regular behavior. There is instead no regular dependence of the CPU times on the value of k: small values produce easy instances, while the most difficult instances are encountered for k around $\frac{80}{100}m$ or $\frac{90}{100}m$.

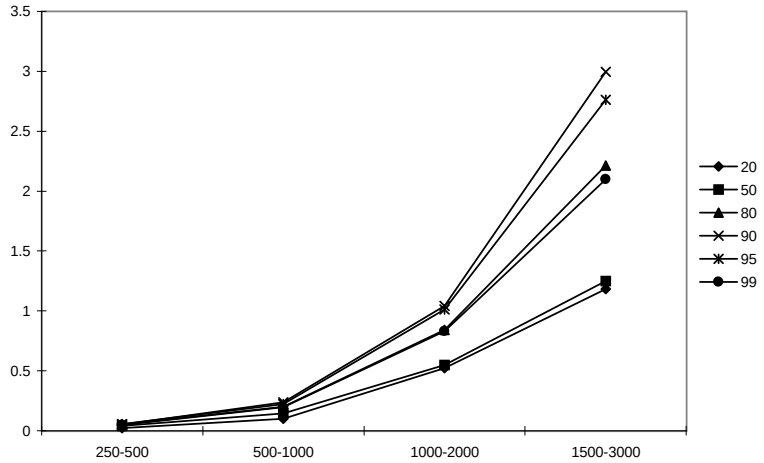


Figure 8.2: Density 5%, range $[0, 10^5]$, rectangular instances. Average CPU time (in seconds) as a function of the instance size.

Table 8.1: Density 1%. Average computing times over 10 problems, Silicon Graphics INDY R10000sc seconds.

			range $[0, 10^2]$			range $[0, 10^5]$		
m	n	$k\frac{100}{m}$	RLXT	CS2	SKAP	RLXT	CS2	SKAP
250	500	20	0.018	0.029	0.005	0.031	0.038	0.008
		50	0.020	0.029	0.032	0.053	0.038	0.037
		80	0.024	0.031	0.039	0.071	0.043	0.035
		90	0.023	0.030	0.039	0.067	0.042	0.038
		95	0.027	0.034	0.037	0.081	0.041	0.037
		99	0.023	0.031	0.041	0.068	0.045	0.039
		Average	**0.023**	**0.031**	**0.032**	**0.062**	**0.041**	**0.032**
500	500	20	0.033	0.049	0.056	0.082	0.062	0.063
		50	0.054	0.056	0.061	0.155	0.064	0.068
		80	0.070	0.057	0.078	0.218	0.068	0.087
		90	0.067	0.055	0.077	0.197	0.073	0.088
		95	0.076	0.058	0.073	0.165	0.074	0.077
		99	0.084	0.061	0.050	0.152	0.078	0.052
		Average	**0.064**	**0.056**	**0.066**	**0.162**	**0.070**	**0.073**
500	1000	20	0.045	0.090	0.029	0.189	0.120	0.061
		50	0.059	0.096	0.115	0.171	0.110	0.126
		80	0.075	0.095	0.118	0.227	0.125	0.141
		90	0.077	0.097	0.131	0.316	0.120	0.164
		95	0.072	0.100	0.182	0.364	0.122	0.202
		99	0.077	0.099	0.153	0.326	0.123	0.159
		Average	**0.068**	**0.096**	**0.121**	**0.266**	**0.120**	**0.142**
1000	1000	20	0.089	0.156	0.231	0.358	0.204	0.231
		50	0.178	0.171	0.227	0.605	0.205	0.256
		80	0.200	0.181	0.294	0.987	0.219	0.351
		90	0.199	0.188	0.271	1.109	0.232	0.326
		95	0.218	0.200	0.239	0.875	0.229	0.300
		99	0.237	0.217	0.142	0.796	0.244	0.202
		Average	**0.187**	**0.186**	**0.234**	**0.788**	**0.222**	**0.278**
1000	2000	20	0.191	0.399	0.323	1.695	0.489	0.448
		50	0.196	0.430	0.446	0.911	0.489	0.471
		80	0.330	0.435	0.455	1.344	0.497	0.571
		90	0.257	0.439	0.519	1.530	0.511	0.683
		95	0.270	0.461	0.735	1.688	0.506	0.779
		99	0.249	0.454	0.612	1.831	0.493	0.615
		Average	**0.249**	**0.436**	**0.515**	**1.500**	**0.498**	**0.595**
2000	2000	20	0.477	1.015	0.905	2.485	1.190	0.913
		50	0.599	1.047	0.858	3.264	1.234	1.029
		80	0.867	1.203	1.078	7.862	1.315	1.620
		90	0.876	1.256	1.007	6.087	1.405	1.435
		95	0.883	1.333	0.895	5.828	1.432	1.312
		99	0.900	1.427	0.503	4.274	1.547	0.893
		Average	**0.767**	**1.214**	**0.874**	**4.967**	**1.354**	**1.200**
1500	3000	20	0.928	1.078	0.038	7.412	1.410	1.026
		50	0.795	1.271	0.996	3.714	1.511	1.077
		80	0.613	1.307	0.983	1.723	1.490	1.262
		90	0.754	1.370	1.112	2.656	1.560	1.673
		95	0.752	1.363	1.476	3.448	1.570	1.853
		99	0.663	1.340	1.252	2.803	1.551	1.420
		Average	**0.751**	**1.288**	**0.976**	**3.626**	**1.515**	**1.385**
3000	3000	20	1.088	2.476	0.069	7.119	2.941	2.083
		50	1.431	2.743	2.012	5.509	3.220	2.387
		80	2.826	3.064	2.543	28.727	3.397	3.926
		90	2.239	3.207	2.328	13.689	3.594	3.428
		95	2.160	3.438	1.975	14.763	3.766	3.034
		99	2.096	3.656	1.194	17.572	3.950	1.973
		Average	**1.973**	**3.097**	**1.687**	**14.563**	**3.478**	**2.805**

Table 8.2: Density 5%. Average computing times over 10 problems, Silicon Graphics INDY R10000sc.

			range $[0, 10^2]$			range $[0, 10^5]$		
m	n	$k\frac{100}{m}$	RLXT	CS2	SKAP	RLXT	CS2	SKAP
250	500	20	0.035	0.076	0.005	0.168	0.091	0.022
		50	0.034	0.080	0.032	0.089	0.093	0.039
		80	0.040	0.082	0.042	0.080	0.107	0.051
		90	0.035	0.075	0.049	0.155	0.092	0.053
		95	0.040	0.080	0.049	0.102	0.107	0.057
		99	0.043	0.077	0.043	0.072	0.093	0.049
		Average	**0.038**	**0.078**	**0.037**	**0.111**	**0.097**	**0.045**
500	500	20	0.054	0.145	0.048	0.225	0.176	0.070
		50	0.063	0.147	0.078	0.176	0.186	0.088
		80	0.098	0.155	0.101	0.560	0.193	0.122
		90	0.111	0.168	0.107	0.489	0.199	0.125
		95	0.114	0.170	0.093	0.497	0.204	0.108
		99	0.114	0.176	0.066	0.388	0.211	0.085
		Average	**0.092**	**0.160**	**0.082**	**0.389**	**0.195**	**0.100**
500	1000	20	0.176	0.358	0.016	1.376	0.450	0.099
		50	0.165	0.405	0.137	0.721	0.501	0.144
		80	0.165	0.419	0.153	0.419	0.508	0.200
		90	0.170	0.432	0.168	0.326	0.499	0.237
		95	0.172	0.424	0.180	0.302	0.502	0.223
		99	0.171	0.435	0.173	0.320	0.488	0.198
		Average	**0.170**	**0.412**	**0.138**	**0.577**	**0.491**	**0.184**
1000	1000	20	0.175	0.928	0.028	1.360	1.135	0.263
		50	0.296	1.035	0.267	1.213	1.230	0.339
		80	0.529	1.126	0.365	4.883	1.285	0.584
		90	0.583	1.165	0.389	3.024	1.315	0.572
		95	0.586	1.215	0.361	4.574	1.365	0.486
		99	0.611	1.257	0.274	3.728	1.428	0.357
		Average	**0.463**	**1.121**	**0.281**	**3.130**	**1.293**	**0.434**
1000	2000	20	1.044	2.161	0.045	9.425	2.560	0.524
		50	0.735	2.430	0.052	4.206	2.862	0.549
		80	0.994	2.700	0.582	2.404	3.065	0.840
		90	1.068	2.752	0.621	1.841	3.070	1.042
		95	0.970	2.772	0.667	1.519	2.981	1.011
		99	0.912	2.652	0.675	1.900	2.918	0.830
		Average	**0.954**	**2.578**	**0.440**	**3.549**	**2.909**	**0.799**
2000	2000	20	1.914	4.775	0.085	6.455	5.419	1.054
		50	2.115	5.460	0.173	4.212	6.062	1.483
		80	2.676	6.048	1.502	33.195	6.475	3.074
		90	3.624	6.334	1.778	29.435	6.662	2.648
		95	3.661	6.598	1.646	35.922	6.759	2.261
		99	3.132	6.943	1.237	36.167	7.402	1.671
		Average	**2.854**	**6.026**	**1.070**	**24.231**	**6.463**	**2.032**
1500	3000	20	3.296	5.459	0.099	29.304	6.607	1.184
		50	1.406	5.958	0.098	13.933	7.282	1.249
		80	3.062	6.531	1.296	6.015	7.462	2.213
		90	2.207	6.898	1.383	4.832	7.594	2.997
		95	2.529	6.932	1.488	4.720	7.447	2.763
		99	2.345	6.613	1.488	5.935	7.179	2.097
		Average	**2.474**	**6.399**	**0.975**	**10.790**	**7.262**	**2.084**
3000	3000	20	9.503	11.850	0.171	14.187	13.428	2.395
		50	3.867	12.669	0.175	9.093	13.938	3.710
		80	12.990	15.402	3.757	74.084	16.270	8.534
		90	12.098	15.735	4.696	94.893	16.925	7.384
		95	10.224	16.366	4.446	96.993	17.938	5.911
		99	7.818	17.444	3.581	105.391	19.175	4.629
		Average	**9.417**	**14.911**	**2.804**	**65.774**	**16.279**	**5.427**

Table 8.3: Density 10%. Average computing times over 10 problems, Silicon Graphics INDY R10000sc.

			range $[0, 10^2]$			range $[0, 10^5]$		
m	n	$k\frac{100}{m}$	RLXT	CS2	SKAP	RLXT	CS2	SKAP
250	500	20	0.062	0.126	0.006	0.479	0.157	0.027
		50	0.062	0.140	0.044	0.243	0.172	0.045
		80	0.074	0.147	0.049	0.123	0.170	0.058
		90	0.071	0.141	0.062	0.103	0.166	0.069
		95	0.065	0.144	0.061	0.105	0.170	0.066
		99	0.066	0.150	0.062	0.118	0.171	0.064
		Average	**0.067**	**0.141**	**0.047**	**0.195**	**0.168**	**0.055**
500	500	20	0.069	0.338	0.011	0.380	0.421	0.065
		50	0.109	0.372	0.080	0.719	0.442	0.098
		80	0.180	0.380	0.129	1.194	0.450	0.183
		90	0.207	0.393	0.133	1.099	0.464	0.175
		95	0.211	0.414	0.130	1.104	0.476	0.166
		99	0.219	0.423	0.097	0.794	0.478	0.124
		Average	**0.166**	**0.387**	**0.097**	**0.882**	**0.455**	**0.135**
500	1000	20	0.362	0.848	0.027	3.299	1.068	0.116
		50	0.267	0.932	0.022	1.587	1.166	0.163
		80	0.399	1.029	0.168	0.894	1.188	0.236
		90	0.365	1.045	0.210	0.692	1.202	0.321
		95	0.356	1.039	0.222	0.550	1.209	0.313
		99	0.347	1.030	0.237	0.488	1.167	0.259
		Average	**0.349**	**0.987**	**0.148**	**1.252**	**1.167**	**0.235**
1000	1000	20	0.491	1.951	0.037	2.177	2.335	0.303
		50	0.642	2.155	0.069	2.122	2.548	0.382
		80	0.892	2.503	0.474	9.234	2.722	0.843
		90	1.085	2.578	0.569	8.973	2.789	0.817
		95	1.218	2.546	0.546	8.413	2.869	0.767
		99	1.107	2.748	0.436	6.904	3.049	0.577
		Average	**0.906**	**2.414**	**0.355**	**6.304**	**2.719**	**0.615**
1000	2000	20	1.980	4.610	0.070	25.197	5.472	0.501
		50	1.009	4.933	0.080	7.292	5.796	0.639
		80	1.898	5.400	0.583	4.349	5.962	0.996
		90	1.840	5.559	0.760	4.198	5.948	1.630
		95	1.596	5.518	0.839	4.648	5.832	1.587
		99	1.671	5.397	0.872	3.666	5.745	1.191
		Average	**1.666**	**5.236**	**0.534**	**8.225**	**5.793**	**1.091**
2000	2000	20	6.281	9.746	0.133	9.597	10.952	1.216
		50	2.105	10.495	0.149	7.715	11.145	1.670
		80	6.176	12.512	2.019	46.621	12.472	4.882
		90	7.773	12.580	3.071	66.271	13.377	4.491
		95	8.017	13.426	2.624	49.976	13.855	3.745
		99	5.773	14.087	2.317	67.158	14.363	2.762
		Average	**6.021**	**12.141**	**1.719**	**41.223**	**12.694**	**3.128**
1500	3000	20	5.124	11.413	0.155	54.703	12.920	1.232
		50	3.890	12.410	0.161	20.098	13.715	1.427
		80	7.012	13.495	0.168	12.327	14.986	2.510
		90	4.381	13.674	1.491	9.505	14.774	4.529
		95	5.027	14.092	1.880	14.312	14.470	4.390
		99	4.666	13.821	1.957	14.103	14.433	3.168
		Average	**5.017**	**13.151**	**0.969**	**20.841**	**14.216**	**2.876**
3000	3000	20	27.022	24.249	0.290	18.440	26.891	2.789
		50	10.271	25.223	0.312	14.506	26.967	3.907
		80	15.036	28.650	5.850	117.946	29.787	13.780
		90	73.373	31.067	10.134	141.552	33.177	12.530
		95	25.598	34.414	11.014	137.860	35.357	10.124
		99	19.422	34.894	7.139	158.249	36.366	7.952
		Average	**28.454**	**29.750**	**5.790**	**98.092**	**31.424**	**8.514**

Table 8.4: Complete matrices. Average computing times over 10 problems, Silicon Graphics INDY R10000sc.

			range $[0, 10^2]$				range $[0, 10^5]$			
m	n	$k\frac{100}{m}$	RLXT	CS2	PRML	SKAP	RLXT	CS2	PRML	SKAP
250	500	20	0.263	2.323	0.029	0.032	5.838	2.577	0.043	0.063
		50	0.347	2.431	0.040	0.039	2.594	2.806	0.117	0.128
		80	0.403	2.543	0.046	0.041	2.279	2.846	0.212	0.145
		90	0.482	2.585	0.049	0.052	2.678	2.905	0.297	0.175
		95	0.498	2.645	0.122	0.119	2.782	2.811	0.386	0.182
		99	0.503	2.693	0.132	0.136	2.777	2.938	0.457	0.243
		Average	**0.416**	**2.537**	**0.070**	**0.070**	**3.158**	**2.814**	**0.252**	**0.156**
500	500	20	1.255	5.042	0.074	0.068	2.923	5.370	0.131	0.221
		50	1.224	5.197	0.083	0.076	4.087	5.549	0.295	0.278
		80	1.451	5.424	0.262	0.253	16.758	6.003	0.860	0.609
		90	1.958	5.528	0.281	0.730	16.187	6.044	1.186	1.116
		95	2.732	5.876	0.292	0.775	17.136	6.631	1.318	1.062
		99	2.691	6.195	0.347	0.784	17.211	6.764	1.458	0.904
		Average	**1.885**	**5.544**	**0.223**	**0.448**	**12.384**	**6.060**	**0.875**	**0.698**

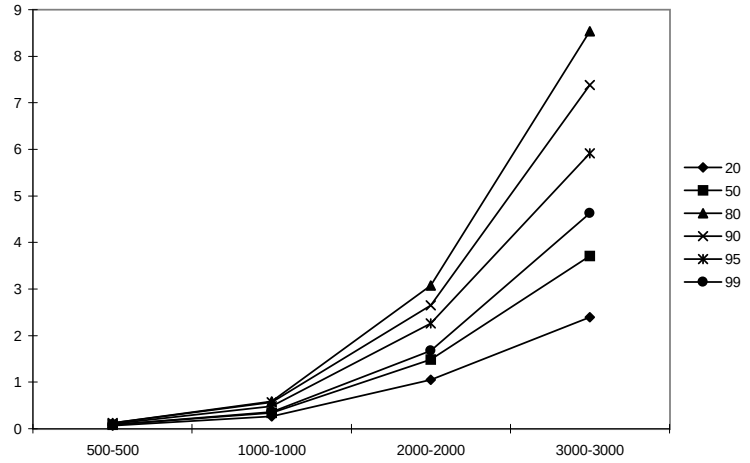


Figure 8.3: Density 5%, range $[0, 10^5]$, square instances. Average CPU time (in seconds) as a function of the instance size.

Chapter 9

E$\mathcal{X}$ploring Tabu Search: the CAP case

9.1 Introduction

Any[1] problem in *combinatorial optimization* can be described by a pair (S, f) where S is a finite set of feasible solutions and $f : S \to \mathcal{R}$ is a given objective function. The goal is to find the solution $s^* \in S$ which minimizes (or maximizes) the objective function over S. It is well known that the computational complexity of a combinatorial problem can change when we maintain the solutions set S, but we change the objective function. One of the simplest examples of such behavior is given by the problem of finding a spanning tree of a graph. If f is linear, then the problem is equivalent to finding a base of a single matroid, so it is solvable in polynomial time with a greedy algorithm (see e.g. Lawler [103]). However if we adopt different objective functions we often have NP-complete problems (see e.g. Camerimi, Galbiati and Maffioli, 1980; Dell'Amico, Labbé and Maffioli [49]). When the problem is a two matroid intersection, i.e. S contains all common subsets of two matroids, we can still solve the problem in polynomial time. A classic example of two matroids intersection is the *Linear Assignment Problem* (AP) which consists of finding n elements of a given $n \times n$ cost matrix $C = [c_{ij}]$, such that no two elements belong to the same row or column, and the sum of the elements chosen is a minimum. (Several polynomial time algorithms for solving AP have been given in the last thirty years, see e.g. Dell'Amico and Martello, [55].)

In this chapter we consider a generalization of AP obtained by replacing its linear objective function with a *cumulative function*. The resulting problem, called *Cumulative Assignment Problem* (CAP), is NP-complete and no specific technique has been proposed before now to solve it. To be more specific, let a be an n dimensional vector of *penalties*. CAP asks for an assignment of each row to exactly one column and for an ordering of the elements of C involved, such that the scalar product of a times the ordered vector of the chosen elements of C is minimized.

An immediate application of CAP is the following. We associate to each row i of matrix C an *operator* and to each column j a *job*. The value c_{ij} is the time spent by operator i to perform job j. Moreover, each penalty a_i $(i = 1, \ldots, n)$ is the cost of operator i for one time unit. Using these definitions, CAP is to determine the assignment of each operator to a job

[1] The results of this chapter appear in: M. Dell'Amico, A. Lodi, F. Maffioli, "Solution of the cumulative assignment problem with a well-structured tabu search method", *Journal of Heuristics* 5, 123–143, 1999, [50].

in such a way that the cost paid to perform all the jobs is minimized. CAP is also interesting since it is a relaxation of more general problems as the Delivery Man Problem (Fischetti, Laporte and Martello [67]). Lastly CAP is a special case of the Three-Dimensional Axial Assignment Problem (see, e.g. Balas and Saltzman [9]).

In this chapter we study the solution of CAP by means of metaheuristic techniques. We consider a Multistart heuristic, a Simulated Annealing algorithm and a new well-structured Tabu Search approach that we have called the *eXploring Tabu Search* ($\mathcal{X}$-TS). The new method is described in detail and the effect of each of the strategies adopted is studied by means of extensive computational experiments. A preliminary version of $\mathcal{X}$-TS has been used with success in Dell'Amico and Maffioli [52]; Dell'Amico and Trubian [58]; Dell'Amico, Maffioli and Trubian [53].

In Section 9.2 we give a mathematical model of CAP, we discuss its complexity status and overview results from the literature on related problems. In Section 9.3 we present lower bounds on the optimal solution value. Section 9.4 is devoted to introducing the neighborhood we have used for all the metaheuristic algorithms, and to presenting an efficient implementation of the exploration of the neighborhood. In Section 9.5 we introduce the metaheuristic techniques we adopted and, in particular, we describe the $\mathcal{X}$-TS approach. Extensive computational results are presented in Section 9.6, whilst the last section summarizes the work and suggests some interesting directions for future research.

9.2 Mathematical model and complexity

Let C be an integer square cost matrix with n rows and columns, and let a be an n dimensional integer vector of penalties. Without loss of generality we will assume that

$$a_1 \geq a_2 \geq \ldots \geq a_n. \tag{9.1}$$

CAP asks for two permutations of the integers $1, 2, \ldots, n$, say ϕ and ξ, which minimize

$$z(\phi, \xi) = \sum_{k=1}^{n} a_k c_{\phi(k),\xi(\phi(k))}. \tag{9.2}$$

Permutation ξ defines the assignment of each row to a column, whereas permutation ϕ gives the ordering of the elements chosen.

Observe that if we are given the "assignment" permutation ξ, then the optimal ordering ϕ is obtained by associating to the smallest $c_{i,\xi(i)}$ element the largest penalty, then associating to the second smallest element the second largest penalty, and so on, i.e.

$$\phi(k) = \text{argkmin}\{c_{i,\xi(i)}, i = 1, \ldots, n\} \quad k = 1, \ldots, n, \tag{9.3}$$

where argkmin denotes the argument of the k-th smallest element.

If we define the boolean variables

$$x_{ijk} = \begin{cases} 1 & \text{if row } i \text{ is assigned to column j} \\ & \text{and } c_{ij} \text{ is the } k\text{-th element chosen} \\ 0 & \text{otherwise} \end{cases} \qquad i, j, k = 1, \ldots, n$$

then we can give the following linear programming model for CAP:

$$(CAP)\ \min z = \sum_{k=1}^{n}\sum_{i=1}^{n}\sum_{j=1}^{n} a_k c_{ij} x_{ijk} \tag{9.4}$$

$$\sum_{k=1}^{n}\sum_{j=1}^{n} x_{ijk} = 1 \qquad i = 1,\dots,n, \tag{9.5}$$

$$\sum_{k=1}^{n}\sum_{i=1}^{n} x_{ijk} = 1 \qquad j = 1,\dots,n, \tag{9.6}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{n} x_{ijk} = 1 \qquad k = 1,\dots,n, \tag{9.7}$$

$$x_{ijk} \in \{0,1\} \quad i,j,k = 1,\dots,n. \tag{9.8}$$

Equations (9.5) and (9.6) are the classic "row" and "column" constraints of AP (for each fixed k value), whereas (9.7) impose that exactly one element of C is assigned to each ordering position k. Observe that (9.4)–(9.8) define a special case of the *Three-Dimensional Axial Assignment Problem* (3AP) which is given by the same constraints and by the more general objective function

$$\min z = \sum_{k=1}^{n}\sum_{i=1}^{n}\sum_{j=1}^{n} d_{ijk} x_{ijk}$$

3AP is well known to be NP-hard and many of its special cases remain NP-hard too. For example Garey and Johnson [78] have shown that 3AP is NP-hard even if the costs d_{ijk} can only assume two distinct values. Another interesting special case is 3AP with *decomposable* costs (D3AP) in which $d_{ijk} = \alpha_i \beta_j \gamma_k$ where α, β and γ are n dimensional vectors of nonnegative numbers. Burkard, Rudolf and Woeginger [29] have shown that also 3DAP is NP-hard.

It is immediately seen that CAP is more general than D3AP. Indeed, any instance of D3AP can be polinomially transformed into an equivalent instance of CAP by setting $a_k = \gamma_k$, for $k = 1,\dots,n$ and $c_{ij} = \alpha_i \beta_j$ for $i,j = 1,\dots,n$, thus proving CAP to be NP-hard.

3AP has been attacked with implicit enumeration methods (Pierskalla [132]; Burkard and Rudolf [28]) and with cutting plane techniques (Balas and Saltzman [9, 10]; Qi, Balas and Gwan [136]). The most effective method is a cutting plane algorithm which solves instances with up to 28 rows and columns in about 2.000 seconds on a Sequent Hydra multiprocessor. It seems therefore to be quite unlikely that the general techniques developed for the exact solution of 3AP can solve large instances of CAP, and hence heuristic techniques are needed to obtain good solutions within reasonable running times.

9.3 Lower bounds

In this section we describe lower bounding procedures from the literature and our adaptation to CAP of a Lagrangean bound developed for 3AP.

The first two lower bounds for CAP have been introduced by Haas [87] and consist of relaxations by constraints elimination.

The first lower bound LE1 considers two problems obtained from CAP by removing, respectively, constraints (9.5) and (9.7), and constraints (9.6) and (9.7). In the first case the elements of matrix C chosen are the minima of each column, whereas in the second case the

elements are the minima of each row. The lower bound value is given by the maximum of the two solution values.

The second lower bound LE2 is obtained with a reformulation of the problem. We first define the "differences" of the penalties

$$d_k = a_k - a_{k+1}, \quad k = 1, \ldots, n$$

where $a_{n+1} = 0$. Then we define the n matrices $C^k = [c^k_{ij}]$ with $c^k_{ij} = d_k c_{ij}$, for $k = 1, \ldots, n$ and we ask for an assignment of exactly k elements from each matrix C^k, with the additional constraint that if an element (i, j) is chosen in matrix C^k it must be chosen also in the next matrices C^l with $l > k$. More formally:

$$(CAP') \quad \min \sum_{k=1}^{n} \sum_{i=1}^{n} \sum_{j=1}^{n} c^k_{ij} x_{ijk} \tag{9.9}$$

$$\sum_{j=1}^{n} x_{ijk} \leq 1 \qquad i, k = 1, \ldots, n, \tag{9.10}$$

$$\sum_{i=1}^{n} x_{ijk} \leq 1 \qquad j, k = 1, \ldots, n, \tag{9.11}$$

$$\sum_{i=1}^{n} \sum_{j=1}^{n} x_{ijk} = k \qquad k = 1, \ldots, n, \tag{9.12}$$

$$x_{ijk} - x_{ijl} \leq 0 \qquad i, j = 1, \ldots, n,\ 1 \leq k < l \leq n \tag{9.13}$$

$$x_{ijk} \in \{0, 1\} \qquad i, j, k = 1, \ldots, n. \tag{9.14}$$

Constraints (9.10) and (9.11) ensure that at most one element from each row and column is chosen from each matrix C^k. Equations (9.12) impose that exactly k elements be chosen from each matrix C^k, whereas (9.13) impose that the elements chosen in a matrix must also be chosen in all the following matrices.

Note that due to constraints (9.12) and (9.13) exactly one element is chosen for the first time from each matrix C^k, whereas the other $k-1$ elements have been chosen in the previous matrices. Moreover from the definition of the differences d_k, the element chosen for the first time in the k-th matrix "cumulates" the penalties $d_k, d_{k+1}, \ldots, d_n$. But $\sum_{h=k}^{n} d_h = a_k$, so choosing a new element from a matrix is equivalent to defining its ordering in the permutation ϕ (see (9.3)). Looking at this model, the meaning of the attribute "cumulative" given to our problem is clear.

Lower bound LE2 is obtained by eliminating constraints (9.13), thus the problem separates into n independent assignment problems with cardinalities $1, 2, \ldots, n$. Each AP with fixed cardinality k (with $k \leq n$) has been efficiently solved with the procedure of Dell'Amico and Martello [54].

Another effective lower bound can be obtained by adapting to CAP the Lagrangean bound proposed by Balas and Saltzman [10], for 3AP. This bound is obtained by embedding in a Lagrangean fashion constraints (8.4) in the objective function. Applying this technique to CAP we obtain the following:

$$\text{LCAP}(\lambda) = \min \sum_{k=1}^{n} \sum_{i=1}^{n} \sum_{j=1}^{n} a_k c_{ij} x_{ijk} + \sum_{k=1}^{n} \lambda_k (1 - \sum_{i=1}^{n} \sum_{j=1}^{n} x_{ijk}) \tag{9.15}$$

subject to (9.5), (9.6) and (9.8). Observing that the above objective function can be rewritten as

$$\text{LCAP}(\lambda) = \sum_{k=1}^{n}\sum_{i=1}^{n}\sum_{j=1}^{n}(a_k c_{ij} - \lambda_k)x_{ijk} + \sum_{k=1}^{n}\lambda_k, \tag{9.16}$$

one can see that the optimal solution to LCAP(λ) can be obtained by solving a linear assignment on the reduced costs $\overline{c}_{ij} = \min_k\{a_k c_{ij} - \lambda_k\}$, for $i, j = 1, \ldots, n$.

The Lagrangean dual LAG $= \max_\lambda$ LCAP(λ) is then solved with the modified subgradient technique introduced by Camerini, Fratta and Maffioli [30].

Our computational experiments with these three lower bounds show that bounds LE1 and LE2 give better results than LAG only when few iterations of the subgradient optimization are allowed. Since our final aim is to use the better lower bound value only to evaluate the quality of the solutions obtained with the heuristic algorithms, we therefore gave LAG a large time limit (one hour of CPU time on a Sun Sparc Ultra 2 workstation) so that the subgradient optimization converged, in almost all cases, to its maximum. With this time limit LAG always turned out to be the winner versus LE1 and LE2. Therefore the comparisons of Section 9.6 are made with the lower bound value computed by LAG. (We do not report our computational experiments with the three lower bounding procedures, since we are only interested in the final lower bound value and not in studying the relative performances of the three methods, when the time limit changes.)

9.4 A Neighborhood

In this section we introduce and discuss one of the basic elements, common to the metaheuristic methods: the neighborhood.

A metaheuristic algorithm is a strategy based on a *local search* (LS). Any LS presupposes the definition of a *neighborhood function*, $\mathcal{N} : S \to 2^{|S|}$, i.e. a mapping of the solution space which associates with each solution $s \in S$ a subset $\mathcal{N}(s) \subset S$. The LS method starts with a solution s, moves to an *adjacent* solution $s' \in \mathcal{N}(s)$, then defines $s = s'$ and iterates the process until a given stopping criterion holds. Roughly speaking, the various metaheuristics differ in the choice of the neighborhood and in the strategy used to select the next solutions.

We are interested in studying the effect of different strategies, when the same neighborhood is used. Therefore we need a 'good' neighborhood and an efficient procedure for its exploration, which can be used for all the metaheuristics we consider.

One of the problems in the design of an algorithm based on LS is the trade-off between the width of the neighborhood and the time used for its exploration. The larger a neighborhood, the more accurate the search, for a single iteration, but the longer the time spent in the exploration. Both the accuracy of the search and the total number of iterations help in finding good solutions, therefore one has to determine a suitable compromise between accuracy and speed. In order to have both the above advantages we adopted a wide neighborhood, but we carefully studied the implementation of its exploration, so that the resulting code is fast.

More precisely, let $\xi = (\xi(i), \ldots, \xi(n))$ be the permutation defining the current assignment. Since any permutation defines a unique solution of CAP, let us call ξ a "solution". Given two integers i, j with $1 \le i < j \le n$ we can define a new permutation ξ' by *swapping* $\xi(i)$ with $\xi(j)$, i.e. setting $\xi'(i) = \xi(j)$, $\xi'(j) = \xi(i)$ and $\xi'(l) = \xi(l)$ for $l \neq i, l \neq j$. Our neighborhood $\mathcal{N}(\xi)$ consists of all the permutations generated with all the possible choices of pair i, j.

The value of the given solution ξ can be computed in $O(n \log n)$ time by reordering the elements $c_{l,\xi(l)}$ $(l = 1, \ldots, n)$ by nondecreasing weights and defining $\phi(l) = l$ for $l = 1, \ldots, n$. The next solution values can be computed in $O(n)$ time, since only two elements are changed, at each iteration. Each neighborhood contains $O(n^2)$ different solutions, thus exploring $\mathcal{N}$ with a standard implementation requires $O(n^3)$ computational time. But this is a very long time, for any effective heuristic algorithm, so we need to speed up the exploration with an improved implementation. In particular we describe how to reduce the time required to compute each solution value, from $O(n)$ to $O(\log n)$.

When a swap is performed the assignments $(i, \xi(i))$ and $(j, \xi(j))$ are removed from the current solution and the two new assignments $(i, \xi(j))$ and $(j, \xi(i))$ are added to the solution. This determines the substitution of two values from the set $A = \{c_{l,\xi(l)}, l = 1, \ldots, n\}$ of the assigned elements with two new values. Let us store the elements of A into an n-dimensional vector e such that

$$e_k \leq e_{k+1}, \quad k = 1, \ldots, n-1 \tag{9.17}$$

Then the value of the current solution is obtained with the scalar product ea (recall that the penalties vector a is ordered according to (9.1)). Let *out* be the smallest index of an element of e which contains the value $M^o = \max(c_{i,\xi(i)}, c_{j,\xi(j)})$ and let *in* be the smallest index of an element of e such that $e_{in} > M^i = \max(c_{i,\xi(j)}, c_{j,\xi(i)})$ (see Example 1 below). The indices *out* and *in* can be identified in $O(\log n)$ with a binary search. We now consider the variation of the solution obtained by removing the element with value M^o and adding the element of value M^i. We call e' the resulting vector, reordered by nondecreasing values. According to the relative positions of *in* and *out* several cases arise: we describe in detail only the case $in < out$; the other cases are similar and a complete description of them could be rather boring.

The new partial solution value can be efficiently computed by observing that: (a) $e'_k = e_k$ for $k = 1, \ldots, in-1$ and for $k = out+1, \ldots, n$; (b) the elements $e_{in}, \ldots, e_{out-1}$ are shifted one position right when M^i is inserted, i.e. $e'_{k+1} = e_k$ for $k = in, \ldots, out-1$; (c) $e'_{in} = M^i$. Therefore if z denotes the current solution value, then the new partial solution assumes value:

$$z^p = z - \sum_{k=in}^{out} e_k a_k + \sum_{k=in}^{out-1} e_k a_{k+1} + M^i a_{in} \tag{9.18}$$

Let us define the vectors:

$$\begin{aligned} \sigma_k &= \sum_{l=1}^{k} e_l a_l \quad k = 1, \ldots, n \\ \widehat{\sigma}_k &= \sum_{l=1}^{k} e_l a_{l+1} \quad k = 1, \ldots, n-1 \end{aligned}$$

then $\sum_{k=in}^{out} e_k a_k = (\sigma_{out} - \sigma_{in-1})$ and $\sum_{k=in}^{out-1} e_k a_{k+1} = (\widehat{\sigma}_{out-1} - \widehat{\sigma}_{in-1})$. Thus if we have already stored σ and $\widehat{\sigma}$ the computation of (9.18) can be done in constant time.

Example 1. Let us consider an instance with $n = 10$, penalties a =(25,22,19,13,10,8,6,4,1,0) and the vector of the values currently assigned being $e = (1, 5, 9, 10, 13, 15, 16, 17, 22, 26)$. The current solution value is $z = 872$. Assume that a swap removes the values 9 and 17, and adds the new values 11 and 14. Our implementation first identifies the indices *out* (= 8) and

in (= 6) associated with the two values M^o (= 17) and M^i (= 14). Vector e' is obtained by removing value 17, by shifting e_6 and e_7 one position right, and by inserting the value 14 into e'_6.

$$\begin{array}{rcccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ e = (& 1 & 5 & 9 & 10 & 13 & 15 & 16 & 17 & 22 & 26) \\ & & & & & & & \searrow & \searrow & & \\ e' = (& 1 & 5 & 9 & 10 & 13 & & 15 & 16 & 22 & 26) \\ & & & & & & \Uparrow & & & & \\ & & & & & & 14 & & & & \end{array}$$

Using vectors σ and $\widehat{\sigma}$ (9.18) becomes

$$\begin{aligned} z^p &= z - (\sigma_8 + \sigma_5) + (\widehat{\sigma}_7 - \widehat{\sigma}_5) + M^i a_{in} \\ &= 872 - (850 - 566) + (592 - 438) + 112 = 854 \quad \square \end{aligned}$$

Applying a procedure similar to the above, we can compute the complete value of the new solution ξ', by determining the change of z^p due to the other two elements involved in the swap: $m^o = \min(c_{i,\xi(i)}, c_{j,\xi(j)})$ and $m^i = \min(c_{i,\xi(j)}, c_{j,\xi(i)})$). We need to consider the vector e' and to find the smallest index out' such that $e'_{out'} = m^o$ and the smallest index in' such that $e'_{in'} > m^i$. Using a straightforward implementation one could determine the two indices in $O(n)$ time, by defining and scanning vector e', but we can speed up the search by avoiding the explicit definition of e'. Let us consider first in': since $m^i \leq M^i$, then $in' \in \{1, \ldots, in\}$. But in this interval e' and e coincide, so we can search the required value in e instead than in e'. The definition of out' is a little more complicated. If $m^o \leq M^i$, then we can search again the required value in $e_1, \ldots, e_{in-1}$, (which coincides with $e'_1, \ldots, e'_{in-1}$). Otherwise ($m^o > M^i$) we know that $m^o \in \{e_{in}, \ldots, e_{out-1}\}$, but these elements should be stored in e' one position righter than in e. Hence we can simply search m^o in e (instead than in e') and define out' as the index of the element we found, plus one.

Lastly, to compute the complete solution value we should define the final vector e'' obtained from e' by removing the element of value m^o, by inserting the element of value m^i and by shifting the elements between in' and out'. Once again we want to avoid the explicit definition of e'', so we need to know the exact positioning of the elements of e in e''. If $out' \leq in$ then the elements of e with indices between in' and $out' - 1$ are shifted one position to the right when e'' is defined. Hence the final value z can be computed efficiently using vectors σ and $\widehat{\sigma}$. If otherwise $out' > in$ then the elements of e with indices between positions in and out' should be shifted two positions right to obtain vector e''. Therefore to compute z efficiently we need to define a new vector $\overline{\sigma}$ in which a value e_l is associated with the penalty a_{l+2}, i.e.

$$\overline{\sigma}_k = \sum_{l=1}^{k} e_l a_{l+2} \quad k = 1, \ldots, n-2$$

Summarizing, the computational time required to explore a neighborhood is as follows. We need $O(n \log n)$ time to compute the value of the starting solution ξ and to define the vectors $\sigma, \widehat{\sigma}, \overline{\sigma}$ and other similar vectors needed for the cases we have not described explicitly. Then for each solution in $\mathcal{N}(\xi)$ we need $O(\log n)$ to identify the indices in and out and a constant time to compute the value z^p of the partial solution. Then we need again $O(\log n)$ to define in' and out' and a constant time to compute the value of the complete solution ξ'. We have thus proved the following theorem.

Theorem 9.1. *Neighborhood $\mathcal{N}$ can be explored in $O(n^2 \log n)$ time.*

A further reduction of the average computing time was obtained by applying several simple criteria which, in many cases, allow one to determine if a solution is not improving with respect to the current best solution of the neighborhood, without computing the exact solution value.

9.5 Metaheuristic algorithms

We considered three metaheuristic methods based on local search: *Multi-Start* algorithm, a *Simulated Annealing* algorithm and an implementation of the *Tabu Search* method, called *eXploring Tabu Search*. As already observed, we will use the neighborhood described in the previous section for all the algorithms we consider. According to the general description of an algorithm based on local search, given at the beginning of Section 9.4, it remains to present only the search strategy adopted for each method (see e.g. Aarts and Lenstra [2] for a comprehensive description of the different metaheuristic algorithms).

0.1. Multi-Start

A Multi-Start algorithm basically consists of two nested loops. At each iteration of the external loop we simply generate a random feasible solution, which is improved by the operations performed in the internal loop. Given a current solution s at each iteration of the inner loop, the best solution $s' \in \mathcal{N}(s)$ is selected. If $z(s') < z(s)$ then we set $s = s'$ and we start a new iteration of the internal loop, otherwise a local optimum has been found and the inner loop terminates. The most common criteria used to stop the algorithm consist of giving a global time limit, or of fixing the number of iterations of the external loop. The algorithm returns the best solution identified during the search.

The procedure we used to generate the random feasible solutions is an implementation of a *greedy randomized* (GR) algorithm. We start by ordering the entries of the cost matrix C by non-decreasing values, then we build a feasible solution by performing n times the following operations. At each iteration we enlarge a current partial solution by adding a pair (i, j) such that both row i and column j are not assigned in the partial solution. More specifically, given a parameter $K > 0$, the pair to be added is randomly selected among the K pairs with smallest c_{ij} which can be feasibly added to the partial solution. If $K = 1$ this method is a pure (deterministic) greedy algorithm, otherwise it is a randomized method in which the effect of the randomization on the final solution is evident to the extent that the value of K increases. It is well known that the average value of the solutions obtained with a GR algorithm first decreases when K increases, but after a small threshold value it grows rapidly with K. We performed preliminary computational experiments by generating an instance of Class A (see Section 9.6) with $n = 200$ and solving this instance 1000 times with algorithm GR, with parameter K set to 2,3,5,6,8 and 10, respectively. In Figure 9.1 we report, for each value of K, the average solution values over the 1000 runs. According to the results of these experiments we set the value of K to five.

0.2. Simulated annealing

The Simulated Annealing method (SA) we implemented starts with a random feasible solution s, generated with procedure GR, and iteratively applies the following steps:

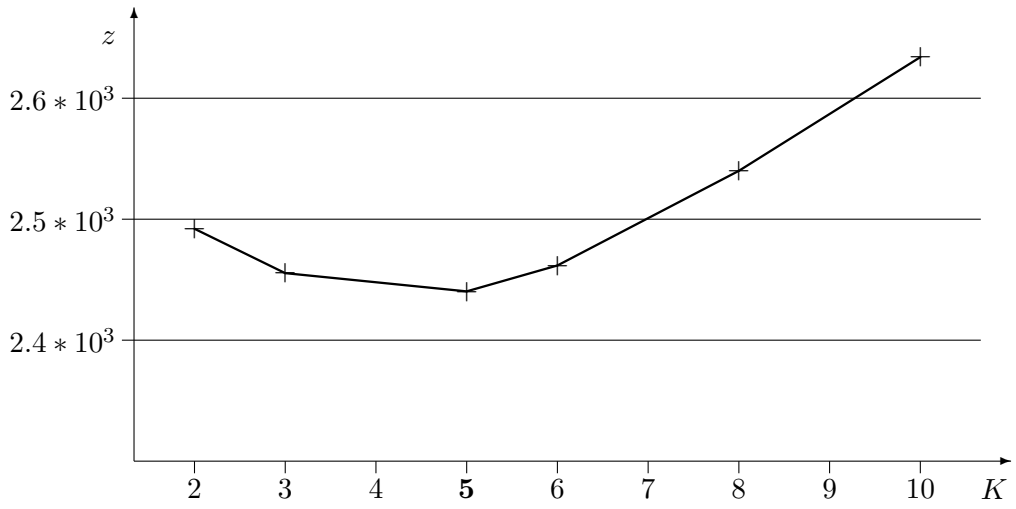


Figure 9.1: Average value of the solutions obtained with algorithm GR (w.r.t. parameter K).

1) randomly select a solution $s' \in \mathcal{N}(s)$
2) **if** $z(s') \leq z(s)$ **then** set $s = s'$
else set $s = s'$ with probability $e^{(z(s)-z(s'))/temp}$

where *temp* is the current *temperature*.

The overall algorithm starts by setting the value of *temp* to an initial temperature T_{start}. Then it repeats the above two steps a number $NITER(n)$ of times, depending on the size of the problem, and decreases the temperature with the *geometric cooling schedule*: $temp = \alpha \cdot temp$ ($\alpha < 1$). When the temperature descends below a minimum value T_{end} it sets again $temp = T_{start}$ and continues as above until a time limit is reached. (See e.g. Aarts, Korst and Laarhoven [1] for a complete description of the SA method.)

In our experiments we defined $NITER(n) = 300n$ and set α to 0.95. The starting and ending values of the temperature were determined through a preprocessing phase which generates 100 random solutions and, for each of them, performs 100 random swaps. During this phase we compute the minimum and maximum difference in cost, say δ and Δ, respectively, between two adjacent solutions. The value T_{start} is set to $-\Delta/\ln(0.90)$ whilst the value T_{end} is set to $-\delta/\ln(0.01)$. This implies that if $temp = T_{start}$ (resp. $temp = T_{end}$), during step 2 a solution s' whose value is equal to $z(s) + \Delta$ (resp. $z(s) + \delta$) is accepted with probability 90% (resp. 1%).

0.3. The eXploring Tabu Search

In this section we describe a general structured method for implementing a Tabu Search algorithm (TS). In particular we introduce a combination of strategies from the Tabu Search framework (see e.g. Laguna and Glover [101] and Glover and Laguna [82]) which leads to a new method that we have called the *eXploring Tabu Search* ($\mathcal{X}$-TS). The reason for the choice of this name will be explained later.

An ideal beginning level Tabu Search method starts with a feasible solution $s \in S$ and, at each iteration, it substitutes s with the best solution $s' \in \mathcal{N}(s)$ that has not been visited in

a previous iteration. The new solution can have an objective function value smaller, equal to or larger than the current solution. In practice it is not possible to store all the information describing all the visited solutions, so we try to recognize a solution using only some *attributes*, i.e. partial information on the structure of the solution. These attributes are stored in a finite length list and a solution $\hat{s} \in \mathcal{N}(s)$ is said to be *tabu*, i.e. it is not considered as a possible candidate for the next iteration, if its attributes are in the list. The search terminates when a given time limit expires.

A common theme of tabu search is to join the beginning level TS with an intermediate term intensification strategy and a longer term diversification strategy, to create an iterated multi-level approach. In spite of this theme, many implementations are limited to the beginning level component of TS. Our $\mathcal{X}$-TS provides a specific and highly effective pattern of a more advanced multi-level design, which we demonstrate to be dramatically superior to the beginning level TS component by itself.

Before going into the details of the $\mathcal{X}$-TS strategies we must describe our implementation of the beginning level TS. Let ξ and ξ' be, respectively, the permutations defining the current solution and the solution selected in $\mathcal{N}(\xi)$. Moreover, let i and j be the row indices of the elements involved in the swap which transforms ξ into ξ'. When we have moved to ξ', the attributes we use to identify solution ξ are the two pairs $(i, \xi(i))$ and $(j, \xi(j))$. In the following iterations we consider tabu a solution which tries to assign again row i to column $\xi(i)$ or row j to column $\xi(j)$. The attributes are stored in a simple FIFO list of length ℓ: when a new solution is selected the two attributes of the new solution are added on the top of the list and the two oldest attributes are removed, if the list length exceeded ℓ. We also apply a simple *aspiration criterion* which removes the tabu status of a solution if the solution value is smaller than the current best solution value.

We are now ready to introduce the specific strategies we have adopted. We use a multi-level approach consisting of three intensification and/or diversification tools which operate on areas of the solution space growing with the level. The first level tool operates in the neighborhood of the current solution. The second level tool operates on a *local area* which is close to the trajectory in the solution space, followed during the search. The third level tool operates on the whole solution space. The first tool consists of a *tabu list management*, the second one is a *proximate good solutions management*, whilst the third is a *global restarting strategy.*

Tabu list management

This tool is an implementation of a strategy known as *dynamic updating of the tabu list length.* The main idea is to emphasize the intrinsic behavior of the TS method. Indeed TS is a so-called "hill climbing" method, i.e. it descends into a "valley" to find a local minimum, then it climbs one of the faces of the same valley trying to reach a different minimum, placed in another valley. The tabu tenure ℓ, i.e. the length of the tabu list (or the number of iterations a solution maintains its tabu status) is initialized to a given value *start_tenure* and is modified according to the evolution of the search. The aim is to intensify the search when we think we are close to a local minimum, and to accelerate the diversification when we are escaping from an already visited minimum.

The intensification is obtained by shortening the tabu tenure, i.e. allowing more solutions to be considered candidate for the next step. On the contrary the diversification is obtained by increasing the value ℓ, which avoids removing the attributes of the recent solutions from the

tabu list. More specifically, let us call *improving phase* a set of Δimp consecutive iterations which lower the objective function value. If we detect an improving phase, then the search is certainly going toward a local minimum, so we reduce the tabu tenure. In particular we use the following updating which guarantees that the tabu tenure remains greater than a reasonable minimum:

$$\ell = \max(\ell - 1, \frac{1}{2} start\_tenure).$$

Now let us consider the case in which the current solution is a local minimum and we start to climb a face of the valley. If the tabu tenure does not change, and the climbing is long enough, then after ℓ iterations the attributes of the local minimum are forgotten and there is the possibility of the search returning toward the already visited minimum. To avoid this, it is important that before ℓ iterations are performed we increase the value ℓ so that we do not forget the attributes of the local minimum. An effective choice is to increase the tabu tenure before ℓ climbing iterations have been performed, so providing that also the attributes of few solutions visited just before descending into the minimum remain in the list. Let us call *worsening phase* a set of $\Delta wor < \ell$ consecutive iterations in which the objective function value does not improve. If a worsening phase is detected, then it is enough to increase the tabu tenure by one, in order to recall the attributes of the minimum and of the last $\ell - \Delta wor$ solutions visited before reaching the minimum. In practice, we adopt the following updating:

$$\ell = \min(\ell + 1, \frac{3}{2} start\_tenure),$$

which limits the tabu tenure, so that the tabu status does not become too binding.

Preliminary computational experiments were used to fix the value of *start_tenure* to 15 and the values of Δimp and Δwor to 3. The same parameters were used for all the remaining experiments, presented in Section 9.6.

Proximate good solutions management

This is an implementation of another tool from the TS framework which is often neglected: a long term memory which enables the algorithm to learn from its evolution.

The basic idea of this tool is to store some good solutions which have been analyzed, but not visited, during the evolution of the algorithm. These solutions are used, under certain conditions, to continue the search. When we have recourse to one of such solutions, then we jump from the current solution s to a new one which is not in $\mathcal{N}(s)$, but is in a promising region close to the trajectory in the solution space followed by the algorithm up to the current iteration. Hence the method implements both an intensification of the search into regions analyzed, but not completely explored, and a diversification from the current solution.

We implemented this tool as follows. We used a fixed length list called *Second* to store l high quality solutions which were analyzed during the search, but whose value was only the second best value in their neighborhood. At each iteration, when we determine, as well as the best solution $s' \in \mathcal{N}(s)$, also the second best solution $s'' \in \mathcal{N}(s)$, then we add s'' to *Second*, either if we have already stored less than l solutions or if there exists a solution $\hat{s} \in Second$ such that $z(s'') < z(\hat{s})$. Note that owing to our implementation not all the solutions in a neighborhood are examined, so it may happen that we compute completely only one solution and none is added to *Second*.

Instead of using a solution of the current neighborhood, we resort to a solution from *Second*, when the behavior of the search indicates that a great effort would be necessary to

find an improving solution, if we continue the search from the current neighborhood. We use three conditions to try to detect the above situation:

1. the tabu status prevents all the solutions in the current neighborhood from being used;
2. the current objective function value has not been improved in the last MC iterations;
3. in the last MB iterations there was no improving of the global best solution.

If one of the three above conditions holds, we remove from *Second* the solution s^* with the best objective function value and we continue the search from it. Note that we obtain a correct working of the algorithm only if we can restore the conditions present when s^* was added to *Second*. To do this we need to store in *Second*, within each solution, also a copy of the tabu list and of the other parameters driving the search.

The meaning of the first condition is obvious, but some explanations are necessary for the other two conditions. The value MC involves using a solution from *Second* when the algorithm is climbing a very deep and high face, or when it is exploring a flat region. In both cases the last local optimum remains the best local solution for many iterations: using a solution from *Second*, we accelerate the search by jumping into a new region. This jump is a diversification from the current solution, but it is also an intensification of the search in the local area, since the solutions stored in *Second* are not too far from the current solution, (they were found along the path leading to the current solution).

From a set of few preliminary experiments we have seen that the value 15 is adequate for MC, for the instances we tested, but a slight growing with n is also useful. Hence we adopted the final formula $MC = \lceil 15 + 7\ln(n/100) \rceil$.

The value MB is used to detect situations in which the exploration of the current area seems not to be fruitful in determining the global optimum. In this case we need to have a complete but fast exploration of the area. This is achieved with a further recourse to a solution from *Second*, so that we increase the intensification, but we also draw the search toward a restart from a completely new solution (see below the subsection on the global restarting). Thus the parameter MB reduces the time needed to explore the local area, by inducing further intensification, but also reduces the time between two strong diversification points. For the above reasons the value of MB must not be too small. With preliminary computational experiments we chose to set $MB = 500$ for the smallest instances tested ($n = 50$) and $MB = 800$ for larger instances.

Global restarting strategy

The aim of the previous two tools is to optimize the ratio accuracy/speed in the exploration of a local area in the solution space.

When we are confident enough that no better solution can be found in the current local area, we must move to a new and not yet explored local area. To do this we generate a new starting solution and we re-initialize the search from this new point. This method determines a jump into a new area, so giving a strong diversification.

To apply this tool we obviously need a procedure which generates a different feasible solution at each run. Moreover, it would be most advisable if the procedure is able to define solutions which are "uniformly" distributed in the solution space. In general it is not too difficult to write a procedure satisfying the first requirement, but it is much more difficult to satisfy the second one. In this implementation we used procedure GR to generate the

solutions. This is a randomized method which gives different solutions for different runs, with a sufficiently large probability. However this algorithm does not guarantee any uniformity in the distribution of the solutions. The study of greedy algorithms satisfying the two above requirements is a challenge for future research in the metaheuristic area.

The $\mathcal{X}$-TS method restarts the search when there is some evidence that the search in the current local area is no longer profitable. In particular, we adopted the three following inexact criteria to detect such situations: (a) the conditions adopted for the second level tool indicate that it is necessary to use a solution from *Second*, but the list is empty; (b) the value of the best solution found after I iterations from the last global restart is p percent larger than the value of the global best solution; (c) the second level tool has been used for SL consecutive times without improving the best solution from the last restart, or from the beginning. Criterion (a) is an obvious extension of the considerations which define the use of the second level tool; criterion (b) is useful to detect situations in which the last randomly generated solution (after a restart) belongs to a local area with no good solutions; criterion (c) is introduced to avoid fury in the search inside a single local area.

The flow chart of Figure 9.2 sketches the $\mathcal{X}$-TS method.

The sets of parameters used for criteria (b) and (c) are as follows. Parameter I is adaptively set to $0.1\lceil n/100 \rceil$ times the number of iterations performed from the starting of the algorithm to the first recourse to a global restart. The percentage value p is initially set to $0.05n$ for instances with 'small' values of the starting solution (less than 10^7), and to 0.1 for the other instances. The value of p is increased of one third of its initial value whenever criterion (b) is applied twice consecutively. The value of the last parameter SL is a little more sensitive to the instance. We set $SL = 5$ for all instances, except for that of Class C with $n \leq 100$, (see below Section 9.6) where we set $SL = 10$.

Why the name ?

Most of the metaheuristic and evolutionary methods owe their name to the resemblance of their behavior with some natural phenomenon. This is also true for $\mathcal{X}$-TS, but the phenomenon it tries to imitate is the way of operating of an expert human, in a particular field. Suppose you are a famous explorer whose main activity is to go to some lost-land to look for ancient treasures. In particular, suppose you are interested in finding a treasure hidden in a very large and intricate jungle. You can use all modern equipment (like helicopters, radios, etc.), but the nature of the environment makes them almost ineffective. For example you can take the helicopter and fly over the jungle, but it is so thick and intricate that nothing can be seen beneath the trees. So you can only choose a point on the map and descend to the ground from the helicopter with a rope. Then you can look around this point and if you find some interesting trace or piece of evidence, you can follow this trace and repeat your observation from the new position. Suppose you continue with this method until you see no interesting trace around your last position. Now you take your radio and call the helicopter that picks you up and delivers you to a new random position. This is exactly the behavior of a *Multi-Start approach.*

This explorer uses the technology, but not his brain. Indeed, he makes no effort to learn from his previous explorations. A more skilled explorer, instead, uses short and long term memory to drive the search. In order to obtain the maximum information from his walk on the ground and to avoid walking round in circles, he tries to remember the places where he has already passed by storing in his memory some peculiar aspects of the various places

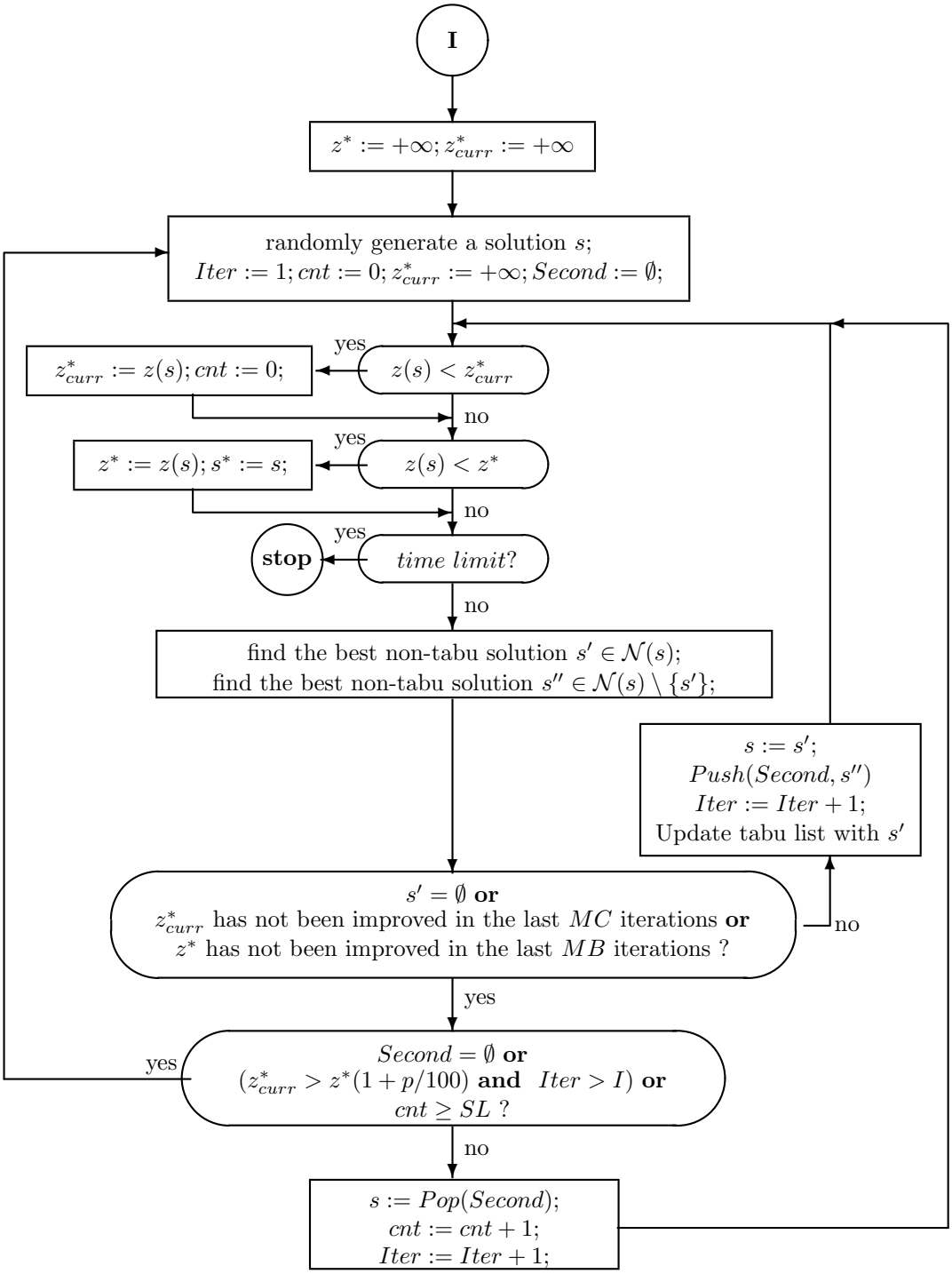


Figure 9.2: The $\mathcal{X}$-TS method.

visited. So he moves from one point to an adjacent one, even if the second is no better than the first. This is the behavior of a beginning level explorer. A more advanced explorer recognizes exceptional interesting situations: for example, a series of consecutive traces. In this case he moves to a new point even if some aspects of the place are similar to those of places already visited. This can be seen as an intensification of the search, into a region close to the current position, obtained by discarding some old tabu status. If, instead, he does not see interesting traces for a certain time interval, then he becomes more cautious and tries to recall all the previously visited places, so that he certainly moves toward new places. This is a local diversification technique. But any explorer with a long experience also knows that if he is in a given point of the jungle and, looking around, he sees more than one interesting trace, then the best local trace does not always lead to the best find. Therefore he tries to recall good traces that he has seen during his walk, but has not followed. When the search fails to give interesting results for a long time, then he returns to the path at the point in which he recalls the first good unexplored trace, and continues the search in this new direction. This is an intensification of the search in the local area, but it is also a diversification from the current situation. Lastly, if the walk does not give good results for a long time he calls the helicopter and moves to a new area. These three ways of approaching the search are the three-level tools of our $\mathcal{X}$-TS method.

9.6 Computational experiments

We have implemented and tested the lower bounds of Section 9.3 and the approximating algorithms of Section 9.5. More precisely we have coded in C language the Multi-Start algorithm (MS), the Simulated Annealing (SA) and four different Tabu Search algorithms. The first algorithm, called TS, is the beginning level tabu search. The second algorithm, denoted with TS1, is an improved version of TS obtained using the first level tool, algorithm TS2 is obtained from TS1 by adding the second level tool, and $\mathcal{X}$-TS is the complete algorithm which uses all the three-level tools. With these four implementations we aim to study the effect of each strategy on the final performances of the tabu search method. The computational experiments were performed on a Sun Sparc Ultra 2 workstation running under Unix System V 4.0.

To test the algorithms, we generated and solved 320 random instances from four different classes. The instances of *Class A* are obtained by randomly generating the costs $[c_{ij}]$ and the penalties a_k, in the interval $[0, 100]$. The instances of *Classes B, C* and *D* are obtained by using, respectively, the intervals $[0, 1000]$, $[50, 100]$ and $[500, 1000]$. The number of rows and columns was set to $50, 100, 150$ and 200. For each pair $(n, Class)$ 20 random instances were generated and solved. We gave all the algorithms the same time limit: $4n$ seconds.

The columns of the tables corresponding to the heuristic algorithms give: (i) the average percentage error $\Delta\%$ between the solution value and the lower bound value (i.e. $\Delta\% =$ 100(upper bound value - LB) / LB); (ii) the number of times the procedure has found the best solution, among those generated by the heuristic algorithms (bst).

In Table 9.1 we report, for each value of n, the averages over the 80 instances generated from the four classes. The total number of best solutions found by the beginning level tabu search TS (see Figure 9.3) is more than double that of MS and SA.

This value strictly increases when we add the first, the second and the third level tool. In particular, $\mathcal{X}$-TS obtains more than twice best solutions than TS. The average percentage

Table 9.1: Grand total for the four classes.

	MS		SA		TS		TS1		TS2		$\mathcal{X}$-TS	
n	Δ%	bst	Δ%	bst	Δ%	bst	Δ%	bst	Δ%	bst	Δ%	bst
50	1.236	7	0.231	28	0.255	29	0.100	41	0.083	53	0.068	52
100	5.478	1	1.251	1	0.649	19	0.360	20	0.369	35	0.315	34
150	15.198	7	3.496	0	1.266	14	1.255	11	1.091	29	1.083	30
200	31.677	15	5.424	0	3.425	9	2.124	16	1.986	27	2.060	34
gr.tot	13.397	30	2.601	29	1.399	71	0.960	88	0.882	144	0.881	150

Sun Sparc Ultra 2 seconds, averages over 80 instances.

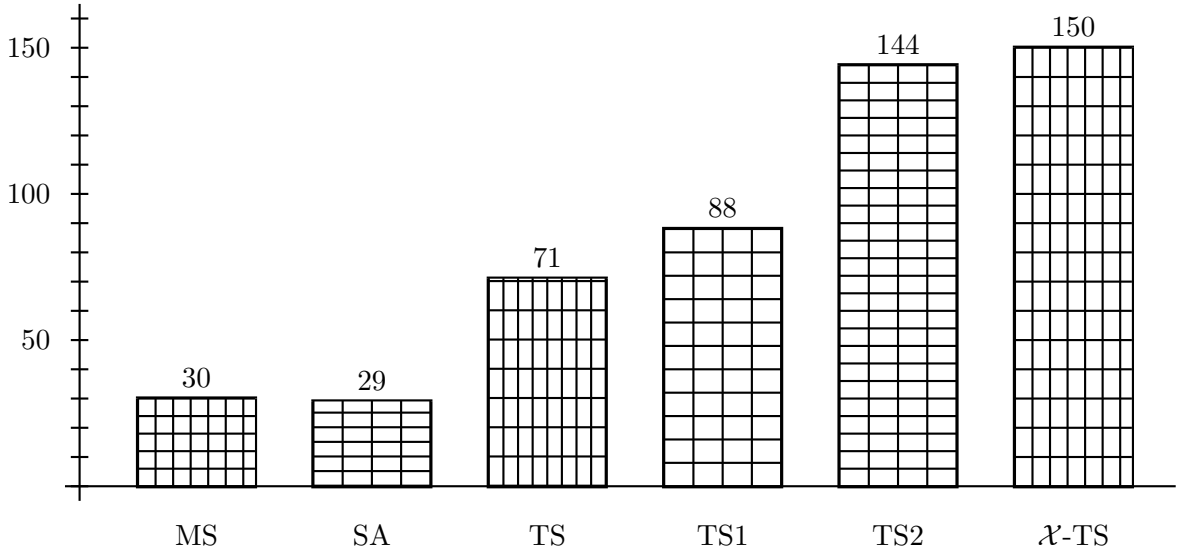


Figure 9.3: Grand total of the best solution found.

error also decreases when we go from TS to $\mathcal{X}$-TS (see Figure 9.4). The error of SA, instead, is two times that of TS, while the error of MS is one order of magnitude larger. However, looking at the disaggregate data of Table reftab:2 we can see that the performances of MS change significantly with the class of instances and are not always bad. For class A it has very poor performances (only one best solution and errors up to two orders of magnitude larger than that of the other algorithms), but for the large instances of class D it performs very well and is slightly better than $\mathcal{X}$-TS (for $n = 200$ the average error of MS is 0.287 versus the average error of $\mathcal{X}$-TSwhich is 0.290).

Lastly it is worth noting that the performances of the tabu search improve at each addition of a tool, but the most significant changes are due to the first two tools. The situation is quite different when we consider other problems. For example in the equicut problem (see Dell'Amico and Maffioli [52]; Dell'Amico and Trubian [58]) the most important tools are the first and the third (tabu list management and global restarting strategy), while in the

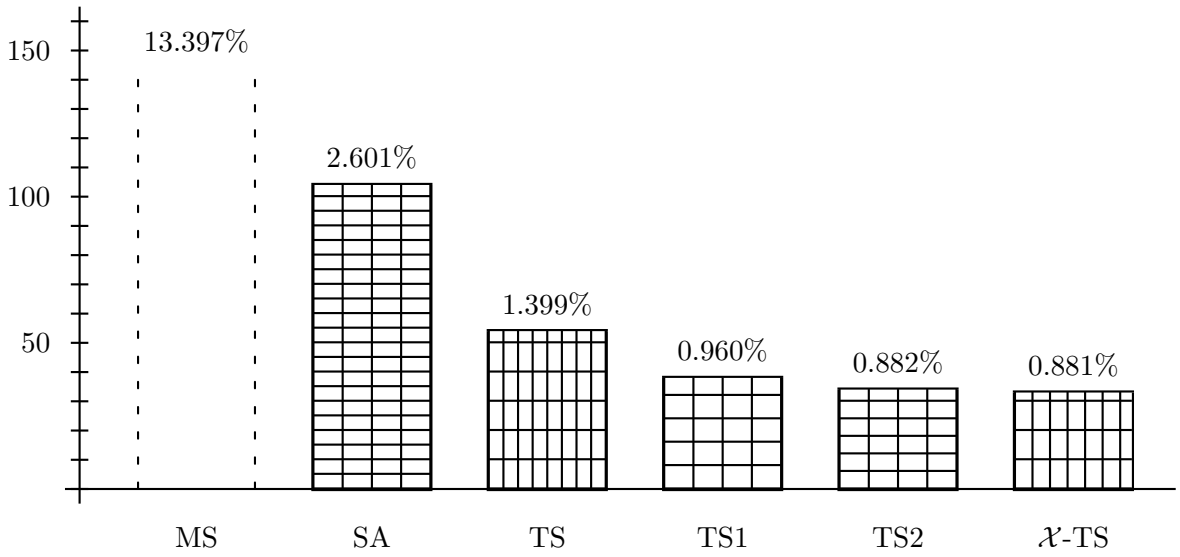


Figure 9.4: Average errors (grand total).

SS/TDMA problem considered in Dell'Amico, Maffioli and Trubian [53], the importance of the second tool (proximate good solutions management) and of the global restarting is comparable.

From these studies it seems that the tabu list management is a "basic" tool which should be used extensively in all tabu search algorithms. Instead the rules to determine when we have to use the second and the third tool are the object of a parameter tuning. Up to now we have no theory which helps to find these rules, therefore preliminary computational experiments are the only method we can apply to define rules and parameters. In the next section we propose some questions and research direction which should be considered to improve the use of long term memory and restarting strategies.

9.7 Conclusions and Future Research

We have considered an NP-complete problem which is obtained by substituting the objective function of the classic linear assignment problem with a cumulative function. We have reviewed the relevant results of the literature and we have proposed a study of metaheuristic algorithms for solving the problem. In particular, we have carefully developed a procedure to explore a neighborhood function and have used this procedure to obtain different metaheuristic algorithms using different strategies with zero memory, local memory or long term memory. Lastly we have described a general meta-strategy using three tools from the tabu search framework. The resulting algorithm, called $\mathcal{X}$-TS, is a general well-structured tabu search approach. The computational results show that the systematic use of strategies based on long time memory helps towards improving dramatically the performances of a beginning level tabu search and that $\mathcal{X}$-TS is very effective as against other metaheuristic approaches.

We also observed that the rules and the parameters which determine the recourse to the second and third tools must be defined with preliminary computational experiments. The

second level tool is a pure long term memory management, whilst the third one is a restarting procedure. Also the third tool can be considered a long term memory-based approach if we are able to drive the generation of new solutions far from the already visited areas. As already pointed out in the previous section, we believe that the study of procedures to generate feasible solutions which uniformly span the space of the solutions is a very important challenge in the theory of metaheuristic algorithms. But other questions arise. In particular, the use of long term memory would be much more effective if we know the mapping $\mathcal{M}_{\mathcal{N}} : S \to S$ which associates with each solution $s \in S$ the local minimum $\hat{s} \in S$ that we would find by applying a pure local search to s, with a certain neighborhood function $\mathcal{N}$. We believe that the study of properties of this mapping is a second very important challenge for future research. An attempt to solve this problem is given in Glover [79] where a structured method to combine previously visited solutions is proposed. A third point that should be considered for improving the theory of metaheuristic algorithms is how one can efficiently store the different local minima corresponding to already visited solutions and how one can efficiently check if the mapping of a given solution s corresponds to an already visited solution.

Table 9.2: Sun Sparc Ultra 2 seconds, averages over 20 instances.

		MS		SA		TS		TS1		TS2		$\mathcal{X}$-TS	
Class	n	$\Delta\%$	bst	$\Delta\%$	bst	$\Delta\%$	bst	$\Delta\%$	bst	$\Delta\%$	bst	$\Delta\%$	bst
	50	3.070	1	0.070	16	0.479	13	0.039	19	0.129	18	0.025	20
A	100	15.651	0	2.534	1	1.598	4	0.486	7	0.703	7	0.486	11
	150	50.055	0	8.688	0	3.277	3	2.981	7	2.706	7	2.693	5
	200	111.452	0	15.601	0	11.115	1	6.052	7	5.461	11	5.719	9
	tot	45.057	1	6.723	17	4.117	21	2.390	40	2.250	43	2.231	45
Class	n	$\Delta\%$	bst	$\Delta\%$	bst	$\Delta\%$	bst	$\Delta\%$	bst	$\Delta\%$	bst	$\Delta\%$	bst
	50	1.511	3	0.285	10	0.285	10	0.040	19	0.032	19	0.060	18
B	100	5.603	0	1.408	0	0.364	8	0.355	9	0.280	9	0.280	9
	150	10.092	0	4.121	0	1.019	8	1.382	2	1.094	6	1.091	7
	200	14.668	0	5.040	0	1.886	6	1.785	6	1.887	8	1.962	8
	tot	7.969	3	2.714	10	0.889	32	0.891	36	0.823	42	0.848	42
Class	n	$\Delta\%$	bst	$\Delta\%$	bst	$\Delta\%$	bst	$\Delta\%$	bst	$\Delta\%$	bst	$\Delta\%$	bst
	50	0.186	2	0.324	0	0.119	5	0.134	3	0.096	7	0.106	6
C	100	0.367	0	0.523	0	0.319	6	0.284	3	0.257	8	0.255	6
	150	0.356	2	0.572	0	0.410	1	0.305	1	0.271	11	0.274	10
	200	0.301	4	0.458	0	0.366	1	0.306	3	0.292	4	0.268	10
	tot	0.303	8	0.469	0	0.303	13	0.257	10	0.229	30	0.226	32
Class	n	$\Delta\%$	bst	$\Delta\%$	bst	$\Delta\%$	bst	$\Delta\%$	bst	$\Delta\%$	bst	$\Delta\%$	bst
	50	0.178	1	0.245	2	0.138	1	0.187	0	0.076	9	0.080	8
D	100	0.292	1	0.539	0	0.315	1	0.314	1	0.237	11	0.239	8
	150	0.289	5	0.604	0	0.358	2	0.353	1	0.294	5	0.275	8
	200	0.287	11	0.596	0	0.334	1	0.354	0	0.302	4	0.290	7
	tot	0.262	18	0.496	2	0.286	5	0.302	2	0.227	29	0.221	31

Chapter 10

AP as Optimization Component for CP Constraints

10.1 Introduction

Finite[1] Domain Constraint Programming (CP) has been recognized as a powerful tool for modelling and solving combinatorial optimization problems. CP tools provide *global constraints* offering concise and declarative modelling capabilities together with efficient and powerful *domain filtering* algorithms. These algorithms remove combinations of values which cannot appear in any consistent solution.

When coping with optimization problems, an objective function f is defined on problem variables. With no loss of generality, we restrict our discussion to minimization problems. CP systems usually implement a Branch and Bound algorithm to find an optimal solution. The idea is to solve a set of satisfiability problems (i.e., a feasible solution is found if it exists), leading to successively better solutions. In particular, each time a feasible solution s^* is found (whose cost is $f(s^*)$), a constraint $f(x) < f(s^*)$ is added to each subproblem in the remaining search tree. The purpose of the added constraint, called *upper bounding constraint*, is to remove portions of the search space which cannot lead to better solutions than the best one found so far. The problem with this approach is twofold: (i) only the upper bounding constraint is used to reduce the domain of the objective function; (ii) in general, the link between the variable representing the objective function and problem decision variables is quite poor and does not produce effective domain filtering.

As concerns the first point, previous works have been proposed that compute also lower bounds on the objective function by (possibly optimally) solving relaxed problems [22], [35], [141], [144].

Concerning the second point, two notable works by Caseau and Laburthe ([36] and [37]) embed in optimization constraints lower bounds from Operations Research and define a regret function used as heuristic information. Here we propose a further step in the integration of OR technology in CP, by using well known OR techniques, i.e., lower bound calculation and reduced cost fixing [124], for cost-based propagation. We embed in global constraints an *optimization* component, representing a proper relaxation of the constraint itself. This

[1] The results of this chapter appear in: F. Focacci, A. Lodi, M. Milano, "Cost-based Domain Filtering", in J. Jaffar, Ed., *Principle and Practice of Constraint Programming - CP'99*, LNCS 1713, Springer-Verlag, Berlin Heidelberg, 1999, 189–203, [71].

component provides three information: (*i*) the optimal solution of the relaxed problem, (*ii*) the optimal value of this solution representing a lower bound on the original problem objective function, and (*iii*) a *gradient function* $grad(V, v)$ which returns, for each possible couple variable-value (V, v), an optimistic evaluation of the additional cost to be paid if v is assigned to V. The *gradient function* extends and refines the notion of regret used in [36] and [37]. We exploit these pieces of information both for propagation purposes and for guiding the search.

We have implemented this approach on two global constraints in ILOG Solver [135]: a constraint of difference and a path constraint. The *optimization* component used in both constraints embeds the Hungarian Algorithm [34] for solving Assignment Problem (AP) which is a relaxation of the problem represented by the path constraint and exactly the same problem as the one modelled by the constraint of difference. The Hungarian Algorithm provides the optimal solution of the AP, its cost and the *gradient function* in terms of reduced costs matrix. Reduced costs provide a significant information allowing to perform cost-based domain filtering, and to guide the search as heuristics. In general however, any relaxation can be used, e.g., a LP relaxation or a spanning tree (spanning forest) for the path constraint, provided that it produces the information needed (i.e., the lower bound and reduced costs).

We have used the resulting constraints to solve Timetabling Problems, Travelling Salesman Problems and Scheduling Problems with setup times (where the path constraint has been interpreted and adapted to be a multi-resource transition time constraint). By using the cost-based domain filtering technique in global constraint, we achieve a significant computational speedup with respect to traditional CP approaches: in fact, we can optimally solve (and prove optimality for) problems which are one order of magnitude greater than those solved by pure CP approaches. Also, comparisons with related literature describing other OR-based hybrid techniques show that integrating cost-based reduction rules in global constraints gets unarguable advantages.

10.2 Motivations and Background

In this section, we present the main motivation of this chapter. We start from the general framework, Branch & Infer, proposed by Bockmayr and Kasper [26], which unifies and subsumes Integer Linear Programming (ILP) and Constraint Programming (CP). In a constraint language, the authors recognize two kind of constraints: *primitive* and *non primitive* ones. Roughly speaking, primitive constraints are those which are easily handled by the constraint solver, while non primitive ones are those for which it does not exist a (complete) method for satisfiability, entailment and optimization running in polynomial time. Thus, the purpose of a computation in a constraint-based system is to infer primitive constraints p from non primitive ones c.

As mentioned, when solving optimization problems, CP systems usually perform the branch and bound method. In particular, each time a feasible solution s^* is found (whose cost is $f(s^*)$), a constraint $f(x) < f(s^*)$ is added to each subproblem in the remaining search tree. The purpose of the added *upper bounding constraint* is to remove portions of the search tree which cannot lead to better solution than the best one found so far. Two are the main limitations of this approach: (*i*) we do not have good information on the problem lower bound, and consequently, on the quality of the solutions found; (*ii*) the relation between the cost of the solution and the problem variables is in general not very tight, in the sense that is usually represented by a non primitive constraints.

Many works have been proposed in order to solve the first problem by computing a lower bound on the problem, thus obtaining in CP a behavior similar to the OR branch and bound technique. In global constraints, for example, a lower bound is computed on the basis of variable bounds involved in the constraint itself, see for instance [144]. Alternatively, Linear Programming (LP) [127] can be used for this purpose as done for example in [22], [35], [141].

The second problem arises from the fact that in classical CP systems primitive constraints are the following:

$$Prim = \{X \leq u, X \geq b, X \neq v, X = Y, integral(X)\}$$

where X and Y are variables, u, v, b are constants. All other constraints are *non primitive*. The branch and bound *a-la* CP would be very effective if the *upper bounding constraint* would be a primitive constraint. Unfortunately, in general, while the term $f(s^*)$ is indeed a constant, the function $f(x)$ is in general not efficiently handled by the underlying solver.

For example, in scheduling problems, the objective function may be the *makespan* which is computed as the $max_{i \in Task}\{St_i + d_i\}$ where St_i is a variable representing the start time of Task i and d_i its duration. In matching, timetabling and travelling salesman problems, each variable assignment is associated with a cost (or a penalty), the objective function is the sum of the assignment costs. In these cases, the function f representing the objective function makes the upper bounding constraint a non primitive one.

The general idea we propose is to infer primitive constraints on the basis of information on costs. We use *optimization* components within global constraints representing a proper relaxation of the problem (or exactly the same problem) represented by the global constraint itself. The optimization component provides the optimal solution of the relaxed problem, its value and a gradient function computing the cost to be added to the optimal solution for each variable-value assignment. In this section, we provide an intuition on how this information is exploited. In section 10.3 we formally explain the proposed technique.

With no loss of generality, we consider here as optimization component a Linear Program (LP) representing a (continuous) linear relaxation of the constraint itself. The optimal solution of the relaxed problem can be used as heuristic information as explained in section 10.4. The optimal value of this solution improves the lower bound of the objective function and prunes portions of the search space whose lower bound is bigger than the best solution found so far. The reduced costs associated to linear variables is proportional to the cost to be added to the optimal solution of the relaxed problem if the corresponding linear variable becomes part of a solution. If this sum is greater than the best solution found so far, the linear variable can be fixed to 0, i.e., it is excluded from the solution. This technique is known in OR as variable fixing [124]. Given a mapping between LP and CP variables, we have the same information for CP variable domain values. Thus, we can infer primitive constraints of the kind $X \neq v$, and we prune the subproblem defined by the branching constraint $p = (X = v)$.

The advantage of this approach is twofold. First, we exploit cost-based information for domain filtering in global constraints. The advantage with respect to traditional OR variable fixing technique is that in our case domain filtering usually triggers propagation of other constraints thanks to shared variables. Second, we do not need to define each time a proper relaxation of the original problem, but we associate a proper relaxation to each global constraint which can be written once for all for optimization purposes. A complementary approach could instead generate a single linear program containing a linearization of the inequalities corresponding to the whole set of constraint representing the problem as done in [141]. This would

allow to have one single global optimization constraint in the form of LP. However, it can be applied only if we consider as a relaxed problem a linear problem, while our approach is more general and we can apply more sophisticated techniques such as additive bounds [69].

10.3 Global optimization constraints

In this chapter we apply our ideas on two global constraints of ILOG solver: a constraint of difference (`IlcAllDiff`) and a path constraint (`IlcPath`) which was extended in order to handle transition costs depending on the selected path.

The constraint `IlcAllDiff` [137] applied to an array of domain variables $Vars = (X_1, ..., X_n)$, ensures that all variables in $Vars$ have a different value.

The constraint `IlcPath` ensures that, given a set of nodes I, a maximum number of paths `NbPath`, a set of starting nodes S and a set of ending nodes E, there exists at most `NbPath` paths starting from nodes in S, visiting all nodes in I and ending at nodes in E. Each node will be visited only once, will have only one predecessor and only one successor. The constraint works on an array of domain variables $Next$, each representing the next node in the path ($Next[i] = j$ if and only if node i precedes j in the solution).

In both cases, as LP relaxation we use the Assignment Problem (AP) solved by the Hungarian algorithm described in [34]. We have chosen the AP solver as a Linear Component for two reasons: (*i*) it is a suitable relaxation for the `IlcPath` constraint and exactly the same problem represented by `IlcAllDiff` constraint; (*ii*) we have a specialized, polynomial and incremental algorithm (the Hungarian method) for solving it and computing the reduced costs[2]. Notice that the proposed approach is independent from the used relaxation. In fact, the algorithm providing lower bound values and reduced costs can be seen as a software component, and it can be easily substituted by other algorithms. For example, an algorithm which incrementally solves the Minimum Spanning Arborescence can be easily used instead of the Hungarian algorithm for computing the lower bound and the reduced costs for the path constraint as shown in [74].

Two important points that should be defined are (*i*) the mapping between variables appearing in the global constraint and variables appearing in the AP formulation; (*ii*) the cost based propagation.

In the next sections, we formally define the Assignment Problem, the mapping and the cost-based propagation.

10.3.1 The Assignment Problem as *optimization* component

The *Linear Assignment Problem* (AP) has been already presented in Chapter 7, and given a square cost matrix c_{ij} of order n, is the problem of assigning to each row a different column, and vice versa in order to minimize the total sum of the row-column assignment costs.

We recall here the ILP model presented in Chapter 7 (and we refer to the notation introduced in that chapter):

$$Z(AP) = \min \sum_{i \in V} \sum_{j \in T} c_{ij}\, x_{ij} \tag{10.1}$$

[2]Note that the AP can be formulated as an Integer Linear Program. However, being the cost matrix totally unimodular, the LP relaxation of the AP always provides an integer (thus optimal) solution.

subject to

$$\sum_{i \in V} x_{ij} = 1, \qquad j \in T \tag{10.2}$$

$$\sum_{j \in T} x_{ij} = 1, \qquad i \in V \tag{10.3}$$

$$x_{ij} \in \{0,1\} \qquad \in V, j \in T \tag{10.4}$$

where $x_{ij} = 1$ if and only if arc (i, j) is in the optimal solution.

As stated in Chapter 7, the AP optimal solution can be obtained through the Hungarian (*primal-dual*) algorithm. We have used a C++ adaptation of the Hungarian algorithm described in [34]. The solution of the AP requires in the worst case $O(n^3)$, whereas each re-computation of the optimal AP solution, needed in the case of modification of one value in the cost matrix, can be efficiently computed in $O(n^2)$ time through a single augmenting path step.

The information provided by the Hungarian algorithm is the AP optimal solution and a reduced cost matrix $\overline{c}$. In particular, for each arc $(i, j) \in A$ the reduced cost value is defined as $\overline{c}_{ij} = c_{ij} - u_i - v_j$, where u_i and v_j are the optimal values of the Linear Programming dual variables associated with the i-th constraint of type (10.2) and the j-th constraint of type (10.3), respectively. The reduced cost values are obtained from the AP algorithm without extra computational effort during AP solution. Each $\overline{c}_{ij}$ is a lower bound on the cost to be added to the optimal AP solution if we force arc (i, j) in solution.

10.3.2 Mapping

In this section, we define the mapping between variables and constraints used in our optimization component and those used in the CP program. The mapping between the ILP formulation and the CP formulation is straightforward and has been previously suggested in [141]. In CP, we have global constraints involving variables $X_1, ..., X_n$ (in the path constraints they are called $Next_i$), ranging on domains $D_1, ..., D_n$, and cost c_{ij} of assigning value $j \in D_i$ to X_i. Obviously, the cost of each value not belonging to a variable domain is infinite. The problem we want to solve is to find an assignment of values to variables consistent with the global constraint, and whose total cost is minimal. If an ILP variable x_{ij} is equal to 1, the CP variable X_i is assigned to the value j, $x_{ij} = 1 \leftrightarrow X_i = j$. Constraints (10.2) and (10.3) correspond to a constraint of difference imposing that all CP variables assume different values. The ILP objective function corresponds to the CP objective function.

It is worth noting that the AP codes work on square matrices, while, in general, in the CP problem considered, it is not always true that the number of variables is equal to the number of values. Thus, the cost matrix of the original problem should be changed. Suppose we have n variables $X_1, \ldots, X_n$, and suppose that the union of their domains contains m different values. A necessary condition for the problem to be solvable is that $m \geq n$. The original cost matrix has n rows (corresponding to variables) and m columns (corresponding to values). Each matrix element c_{ij} represents a cost of assigning j to X_i if value j belongs to the domain of X_i. Otherwise, $c_{ij} = +INF$. In addition, we have to change the matrix so as to have a number of rows equal to the number of columns. Thus, we can add to the matrix $m - n$ rows where each value $c_{ij} = 0$ for all $i = n + 1, \ldots, m$ and for all $j = 1, \ldots, m$, obtaining an $m \times m$ cost matrix. The addition of these $m - n$ rows brings the algorithm to

a time complexity of $O(mn^2)$ (and not $O(m^3)$), whereas each re-computation of the optimal AP solution requires only $O(nm)$ time.

Note that the constraint of difference and the AP component have exactly the same semantics: they compute a solution where all variables are assigned to different values. Thus, each solution of the AP is feasible for the constraint of difference. In general, in a CP program the same variables appear in different constraints. Thus, the constraint of difference alone (and the AP component alone) can be seen as a relaxation of a more general problem. As a consequence, the AP optimal solution Z_{LB} is a lower bound on the optimal solution of the overall problem. On the contrary, when used within a path constraint, the AP component represents a relaxation of the constraint itself (where sub-tours may appear) and it is no longer true that the optimal solution of the AP is feasible for the path constraint. In this case, the AP optimal solution Z_{LB} is a lower bound of the sum of the arcs appearing in the path constraint.

As already mentioned, the AP provides a reduced cost matrix. Given the mapping between LP and CP variables, we know that the LP variable x_{ij} corresponds to the value j in the domain of the CP variable X_i. Thus, the reduced cost matrix $\overline{c}_{ij}$ provides information on CP variable domain values, $grad(X_i, j) = \overline{c}_{ij}$.

10.3.3 The Cost-Based Propagation

In this section we describe filtering techniques based on the information provided by the optimization component. We have a first (trivial) propagation based on the AP optimal value Z_{LB}. At each node of the search tree, we check the constraint $Z_{LB} < Z$ where Z is the variable representing the CP objective function. This kind of propagation generates a yes/no answer on the feasibility of the current node of the search tree; therefore it does not allow any real interaction with the other constraints of the problem.

More interesting is the second propagation from reduced costs $\overline{c}$ towards decision variables $X_1, ..., X_n$, referred to as RC-based propagation. This filtering algorithm directly prunes decision variables $X_1, ..., X_n$ domains on the basis of reduced costs $\overline{c}$. Suppose we have already found a solution whose cost is Z^*. For each domain value j of each variable X_i, $Z_{LB_{X_i=j}} = Z_{LB} + \overline{c}_{ij}$ is a lower bound value of the subproblem generated if value j is assigned to X_i. If $Z_{LB_{X_i=j}}$ is greater or equal to Z^*, j can be deleted from the domain of X_i. This filtering algorithm performs a real back-propagation from Z to X_i. Such domain filtering usually triggers other constraints imposed on shared variables, and it appears therefore particularly suited for CP. Indeed, the technique proposed represents a new way of inferring primitive constraints starting from non primitive ones. In particular, primitive constraints added (of the form $X_i \neq j$) do not derive, as in general happens, from reasoning on feasibility, but they derive from reasoning on optimality. Furthermore, note that the same constraints of the form $X_i \neq j$ are also inferred in standard OR frameworks (variable fixing). However, this fixing is usually not exploited to trigger other constraints, but only in the next lower bound computation, i.e., the next branching node.

When the AP is used as optimization component, an improvement on the use of the reduced costs can be exploited as follows: we want to evaluate if value j could be removed from the domain of variable X_i on the basis of its estimated cost. Let $X_i = k$ and $X_l = j$ in the optimal AP solution. In order to assign $X_i = j$, a minimum augmenting path, say PTH, from l to k has to be determined since l and k must be re-assigned. Thus, the cost of the optimal AP solution where $X_i = j$ is $Z_{LB} + \overline{c}_{ij} + cost(PTH)$, by indicating with

$cost(PTH)$ the cost of the minimum augmenting path PTH. In [74], two bounds on this cost have been proposed, whose calculation does not increase the total time complexity of the filtering algorithm ($O(n^2)$). We will refer to this propagation as *improved reduced cost* propagation (*IRC-based* propagation).

10.3.4 Propagation Events

In this section, we describe the data structures which should be built and maintained by the global constraints, and the events triggering propagation.

When the constraint is stated for the first time, the cost matrix is built and the Hungarian Algorithm is used to compute the AP optimal solution and the reduced cost matrix in $O(n^3)$. Each time the AP optimal solution is computed, the lower bound of the variable representing the objective function is updated and the RC-based propagation is performed (or IRC-based if the corresponding flag is set). The constraint is triggered each time a change in a variable domain happens and each time the upper bound of the objective function is updated. Each time a value j is removed from the domain of variable X_i, the cost matrix is updated by imposing $c_{ij} = +\infty$, i.e., $x_{ij} = 0$. If value j belongs to the solution of the AP (and only in this case), the lower bound Z_{LB} is updated by incrementally re-computing the assignment problem solution in $O(n^2)$. The AP re-computation leads to a new reduced cost matrix. Thus, the RC-based propagation (or IRC-based) is triggered and some other values may be removed.

Note that since the re-computation of the AP solution is needed only if the value removed from the domain of a variable is part of the current AP solution, it is possible to write the optimization constraint in such a way that whenever a value is assigned to a variable only one incremental re-computation is needed. Each time the objective function upper bound is updated, the RC-based propagation (or IRC-based) is triggered.

10.4 Heuristics

The optimal solution of a relaxed problem, the lower bound value, and the set of reduced costs can be used for the heuristics during the search for a solution. Different examples of such use are described in the next section where three combinatorial problems are considered. In general, we can say that the gradient information (reduced costs) can be used to calculate a regret function (see for example [37] for the definition of regret) useful for the variable selection, whereas the optimal assignment in the relaxed problem can be used for the value selection, and finally the lower bound value can be used to select a working subproblem in a local improvement framework, as described in section 10.5.3.

10.5 Computational Results on Different Problems

In this section we present the empirical results on different problems for which the linear assignment problem turns out to be a relaxation. We report computing times (given in seconds on a Pentium II 200 MHz) and number of fails. We refer to different strategies: (*i*) a pure CP approach exploiting the Branch & Bound *a-la* CP; (*ii*) a strategy exploiting the *LB-based* propagation, referred to as **ST1**; (*iii*) a strategy exploiting both the *LB-based* and

RC-based propagation, referred to as **ST2**; (*iv*) a strategy exploiting the *LB-based* and *IRC-based* propagation, referred to as **ST3**. Also comparisons with related approaches on the same applications (if any) are shown. The problems considered are: Travelling Salesman Problems instances taken from the TSP-lib and solved also in [36], Timetabling problems described in [37]. Scheduling Problems with setup times are finally considered and solved using a local improvement technique.

Travelling Salesman Problems have been chosen because standard CP techniques perform very poorly on these problems; we are able to solve problems which are one order of magnitude greater than those solved by a pure CP approach. Caseau and Laburthe in [37] have already shown the advantages of CP techniques in Timetabling problems w.r.t. pure OR approaches. Here we show that the tighter integration proposed outperforms their approach. Indeed, the modelling uses different constraints of difference embedding information on cost. These constraints represent different relaxations of the same problem on shared decision variables. Thus, they smoothly interact with each other and with the entire set of problem constraints allowing to efficiently solve the problem. Finally, preliminary results obtained on Scheduling Problems with setup times show the generality of the approach, and propose a new method for modelling and solving such problems. Implementation details, and more computational results on the TSP and Timetabling problems presented can be found respectively in [74] and [72].

10.5.1 TSPs

TSP concerns the task of finding a tour covering a set of nodes, once and only once, with a minimum cost. The problem is strongly NP-hard, and has been deeply investigated in the literature (see [100] for an annotated bibliography). Although CP is far from obtaining better results than the ones obtained with state of the art OR technology, it is nevertheless very interesting to build an effective TSP constraint; in fact, many problems contain sub-problems that can be described as TSPs, e.g., *Vehicle Routing Problem* (VRP), *Scheduling Problems*, and many variants of TSP are also interesting, e.g., TSP with *Time Windows* (see, Section 10.5.4). In these cases the flexibility and the domain reduction mechanism of Constraint Programming languages can play an important role, and hybrid CP-OR systems could outperform pure OR approaches (as shown in [129] and [130]).

In this section, a set of symmetric TSP instances (up to 30 nodes, from TSP-lib [138]) is analyzed. The pure CP approach has not been reported because it is not able to prove optimality within 30 minutes on none of the instances considered. Our results have been compared with those achieved by Caseau and Laburthe [36] and reported in row CL97, Table 10.1. The computing times of this last row are given in seconds on a Sun Sparc 10.

Table 10.1: Results on small symmetric TSP instances.

Problem	gr17		gr21		gr24		fri26		bayg29		bays29	
	Time	Fails	Time	Fails	Time	Fails	Time	Fails	Time	Fails	Time	Fails
ST1	8.79	13k	0.11	96	1.7	1.5k	19.88	16.6k	89.4	79.8k	135.7	112.8k
ST2	0.71	758	0.05	31	0.28	145	3.68	1.8k	10.6	9.4k	15.4	10.8k
ST3	0.66	646	0.06	31	0.27	120	2.86	1.6k	11.09	7.8k	13.7	8.8k
CL97	3.10	5.8k	7.00	12.5k	6.90	6.6k	930.0	934k	4.4k	4.56M	1.2k	1.1M

The search strategy used exploits the information coming from the optimization component. It implements a sub-tour elimination branching rule often used in OR-based Branch and Bound algorithm for the TSP. In any stage of the search tree, we consider the solution of the AP, we choose a tour belonging to the optimal AP solution, and we branch by removing one arc of the tour in each branch. Note that the tour chosen, infeasible for the TSP, will not appear in any of the generated branches.

Results show that the use of the back propagation from the objective function to the decision variables (strategies ST2 and ST3) turns out to be very important for efficiently solve optimization problems.

10.5.2 Timetabling Problems

The timetabling problems considered have been described in [37]. The problems consist in producing a weekly schedule with a set of lessons whose duration goes from 1 to 4 hours. Each week is divided in 4-hours time slots and each lesson should be assigned to one time slot. The problem involves disjunctive constraints on lessons imposing that two lessons cannot overlap and constraints stating that one lesson cannot spread on two time slots. The objective function to be minimized is the sum of weights taking into account penalties associated to pairs lesson-hour. We have modelled the problem by considering: (*i*) an array of domain variables `Start` representing the course starting times; (*ii*) an array of variables `Slot` representing the slot to which the course is assigned; (*iii*) an array of variables `SingleHours` representing the single hours of each course. Different variables are linked by the following constraints:

```
Start[i] mod 4 = Slot[i]
Start[i] = SingleHours[i][0]
```

Two different matching problems representing two relaxations of the timetabling problem have been modelled by two constraints of difference embedding an optimization component. The first one is the linear assignment relaxation arising when lessons are considered interruptible involving variables `SingleHours`. The cost of assigning each `SingleHours[i]` variable to a value `H` is the cost of assigning the corresponding course to the time slot `H mod 4` divided by the duration of the course. The second relaxation considers variables `Slot` for courses lasting 3 and 4 hours. The corresponding problem is an AP since two 3 or 4 hours courses cannot be assigned to the same slot for limited capacity. The cost of assigning a course to a slot is defined by the problem. The interesting point here is that different problem relaxations coexist and easily interact through shared variables.

In Table 10.2 we report, in addition to the results of the four described approaches, the results obtained by the constraint *MinWeightAllDifferent* described by Caseau and Laburthe [37]. (In the last row of Table 10.2, we refer to row 4 of Table 6 of [37], and the corresponding computing times are given in seconds on a Pentium Pro 200 MHz.)

Table 10.2 shows that for these instances ST2 outperforms in terms of computing times other approaches, although ST3 has more powerful propagation (less number of fails). In this case, in fact, the reduction of the search space does not pay off in terms of computing time.

We have used the information provided by the AP solution also for guiding the search. Defining the regret of a variable as the difference between the cost of the best assignment and the cost of the second best, a good heuristic consists in selecting first variables with high regret. In [37] the regret has been heuristically evaluated directly on the cost matrix as the difference between the minimum cost and the second minimum of each row (despite of the fact that these two minimum could not be part of the first best and the second best solutions).

Table 10.2: Results on timetabling instances.

Problem	Problem 1		Problem 2		Problem 3	
	Time	Fails	Time	Fails	Time	Fails
Pure-CP	3.77	5.4k	5.50	8.5k	11.20	14.5k
ST1	0.70	213	0.15	58	7.60	2.5k
ST2	0.70	199	0.10	30	4.00	1.3k
ST3	0.90	182	0.16	28	6.10	1.2k
CL [37]	29.00	3.5k	2.60	234	120.00	17k

Reduced cost provide a more accurate computation of the regret: for each variable, a lower bound on the regret is the minimum reduced cost excluding the reduced cost of the value in the AP solution. This regret is then combined in a weighted sum with the size of the domain (following the First-Fail principle), and such a weighted sum is used in the variable selection strategy. Concerning the value selection strategy for variable X_i, we have used the solution of the AP.

10.5.3 Scheduling with Set up times

We are given a set of n activities $A_1, ..., A_n$ and a set of m unary resources (resources with maximal capacity equal to one) $R_1, ..., R_m$. Each activity A_i has to be processed on a resource R_j for p_i time units. Resource R_j can be chosen within a given subset of the M resources. Activities may be linked together by precedence relations. Sequence dependent setup times exist among activities. Given a setup time matrix S^k (square matrix of dimension equal to n), s^k_{ij} represents the setup time between activities A_i and A_j if A_i and A_j are scheduled sequentially on the same resource R_k. In such a case, $\text{start}(A_j) \geq \text{end}(A_i) + s^k_{ij}$. Also a setup time su^k_j before the first activity A_j can start on resource R_k may exist. A teardown time td^k_i after the last activity A_i ends on resource R_k may exist.

Constraints of the problem are defined by the resource capacity, the temporal constraints, and the time bounds of the activities (release date, and due date). The goal is to minimize the sum of setup time, given a maximal makespan.

A multiple-TSP *M-TSP* can model a relaxation of the scheduling problem where each resource, and each activity are represented by nodes and arc costs are the setup times. The solution of the M-TSP provides both an assignment of activities to resources and their minimum cost sequencing.

In the following, we will give some preliminary results. The scheduling problem analyzed were solved in two phases: we first looked for a feasible solution, and then we iteratively select a small time window TW_i, we freeze the solution outside TW_i, and perform a Branch and Bound search within the selected window. The scheduling problem considered consists in 25 job of 6 activities each. The activities of each job are linked by temporal constraints and the last activity of each job is subject to a deadline. Each activity requires a set of alternative unary resources and a discrete resource with a given capacity profile.

The first solution (first row of Table 10.3) produces a makespan equal to 2728 and a total setup time equal to 930. This first solution is used as starting point for the local improvement phase. The second row of Table 10.3 reports the improvement on the first

Table 10.3: Results on a Scheduling Problem with setup times.

	Makespan	Total Setup	CPU Time
First Sol.	2728	930	8
Pure-CP	2705	750	386
ST2	2695	600	249

solution obtained using a pure CP approach, while the third row reported the results obtained using the optimization constraint (LB-based and RC-based propagation). Both approaches used the same search strategy. The use of the optimization constraint played an important role in the local improvement phase. In fact for a given time window TW_i, the lower bound gives very good information on the local optimal solution because the scheduling constraints (relaxed on the *M-TSP*) are locally not tight. Indeed, in some cases the gap between the value of the lower bound calculated at the root node and the value of the local optimal solution found is zero.

In this application the optimization constraint is also very important for the selection of the time window TW_i. For each time window TW_i we calculate the gap between the current cost and the lower bound. Such a value is used to select the time window in which running the Branch and Bound optimization. In fact, the higher the gap is, the more chances we have to obtain a good improvement on the solution.

It is important to stress that in this case the optimization constraint interacts with all the scheduling constraints (time bounds, precedence relationship, capacity constraints) thought shared variables. The Edge Finder [11] constraint may, for example, deduce that a given activity A_i must precede a set of other activities, and this information is made available to the optimization constraint.

10.5.4 TSPTW

The Travelling Salesman Problem with Time Windows (TSPTW) is one of the most famous variants of TSP, a time-constrained variant. More formally, TSPTW consists of finding the minimum cost path to be travelled by a vehicle, starting and returning at the same depot, which must visit a set of n cities exactly once. Each pair of cities has an associated travel time. The service at a node i should begin within a time window $[a_i, b_i]$ associated to the node. Early arrivals are allowed, in the sense that the vehicle can reach node i befor a_i, but, in this case, the vehicle has to wait until the node is ready for the beginning of service. This problem can be found in a variety of real-life applications such as routing, scheduling, manufacturing and delivery problems.

One can consider the TSPTW from two different viewpoints: a routing part (TSP) and a scheduling part (related to the time windows). The TSP part is mainly an optimization problem (it is, in general, simple to find a solution, but very difficult to find the best one), whereas often scheduling problems contain very difficult feasibility issues. As already mentioned, CP methods are very effective for scheduling problems, in particular, where the feasibility component is relevant.

We have modeled and solved TSPTW in [73] by using again AP as a bound within a CP framework, and by developing ad hoc propagation techniques and branching strategies. (We

refer to [73] for a complete discussion of these techniques.) In this section, we present some preliminary computational results obtained by this method on a set of real-world asymmetric instances deriving from stacker crane routing applications[3], and considered by Ascheuer, Fischetti and Grötschel [5].

In Table 10.4 we report the results on a subset of these instances, and we compare them with the results obtained with the branch-and-cut approach of Ascheuer, Fischetti and Grötschel. For the branch-and-cut we report in the first column either the value of the optimal solution (when known) or the lower and upper bounds, whereas in the second column we have the computing times (in seconds on a Sun Sparc Station 10). A time limit of 5 hours (5h in Table 10.4) is given. Our algorithm runs with two different branching strategies particularly suitable for problems with a relevant scheduling component (see, [73] for a detailed discussion), and, as in the previous tables, we indicate the computing times, and the number of fails. The computing times are expressed in seconds on Pentium II 200 MHz, and if the time limit of 2 hours (2h in Table 10.4), is reported the corresponding solution is not proven to be optimal.

Table 10.4: Results on rbg instances from Ascheuer, Fischetti and Grötschel [5].

Instance	Ascheuer et al.		Sequence			Sequence LDS		
	best	*time*	*best*	*time*	*fails*	*best*	*time*	*fails*
`rbg010a`	149	0.12	149	0.03	11	149	0.02	11
`rbg016a`	179	0.2	179	0.04	12	179	0.03	12
`rbg020a`	210	0.2	210	0.03	10	210	0.04	10
`rbg027a`	268	2.25	268	0.19	33	268	0.18	33
`rbg031a`	328	1.7	328	279	99.5k	328	318	89.2k
`rbg048a`	[456-527]	5h	503	2h	-	487	2h	-
`rbg049a`	[420-501]	5h	505	2h	-	497	2h	-
`rbg050b`	[453-542]	5h	546	2h	-	548	2h	-
`rbg050c`	[508-536]	5h	573	2h	-	542	2h	-

10.6 Conclusions

In this chapter, we have proposed the use of an optimization component such as a Linear Program in global constraints. For feasibility purposes, global constraints represent a suitable abstraction of general problems. For optimization purposes embedding OR methods in global constraints is a necessary condition for efficiently handle objective functions.

The advantages of the proposed integration are that we are able to infer primitive constraints starting from non primitive ones on the basis of lower bound and reduced costs information. This enhances operational behavior of CP for optimization problems by maintaining its flexibility and its modelling capabilities.

Although most of the OR techniques used are fairly standard in the OR community we believe that their introduction in CP global constraints leads to significant new contributions.

[3]The complete set can be downloaded, along with the best known solutions, from the web page: `http://www.zib.de/ascheuer/ATSPTWinstances.html`.

We greatly powered the CP constraints for optimization problems. We also powered the back-propagation from the objective function to the decision variables; such propagation is limited in a pure OR framework since pure OR branch and bound does not have a constraint store active on shared variables. This last point, in particular, allowed us to easily model and solve problems whose pure OR modelling would lead to very complex algorithms. Finally, the different prospective in which reduced cost fixing is used brought (and may bring) to new contributions such as the *improved reduced cost* propagation.

Future work concern further generalization of the method by integrating in global constraint a general LP solver providing information on lower bound and on reduced costs. Also, we are currently investigating the use of additive bounds [69] and other specialized cost-based methods in global constraints.

Chapter 11

The Multiple Depot Vehicle Scheduling Problem

11.1 Introduction

The[1] *Multiple-Depot Vehicle Scheduling Problem* (MD-VSP) is an important combinatorial optimization problem arising in the management of transportation companies. In this problem we are given a set of n *trips*, $T_1, T_2, \ldots, T_n$, each trip T_j $(j = 1, \ldots, n)$ being characterized by a starting time s_j and an ending time e_j, along with a set of m *depots*, $D_1, D_2, \ldots, D_m$, in the k-th of which $r_k \leq n$ *vehicles* are available. All the vehicles are supposed to be identical. In the following we assume $m \leq n$.

Let τ_{ij} be the time needed for a vehicle to travel from the end location of trip T_i to the starting location of trip T_j. A pair of consecutive trips (T_i, T_j) is said to be *feasible* if the same vehicle can cover T_j right after T_i, a condition implying $e_i + \tau_{ij} \leq s_j$. For each feasible pair of trips (T_i, T_j), let $\gamma_{ij} \geq 0$ be the *cost* associated with the execution, in the duty of a vehicle, of trip T_j right after trip T_i, where $\gamma_{ij} = +\infty$ if (T_i, T_j) is not feasible, or if $i = j$. For each trip T_j and each depot D_k, let $\overline{\gamma}_{kj}$ (respectively, $\tilde{\gamma}_{jk}$) be the non-negative cost incurred when a vehicle of depot D_k starts (resp., ends) its duty with T_j. The overall cost of a duty $(T_{i_1}, T_{i_2}, \ldots, T_{i_h})$ associated with a vehicle of depot D_k is then computed as $\overline{\gamma}_{ki_1} + \gamma_{i_1 i_2} + \ldots + \gamma_{i_{h-1} i_h} + \tilde{\gamma}_{i_h k}$.

MD-VSP consists of finding an assignment of trips to vehicles in such a way that:

i) each trip is assigned to exactly one vehicle;

ii) each vehicle in the solution covers a sequence of trips (duty) in which consecutive trip pairs are feasible;

iii) each vehicle starts and ends its duty at the same depot;

iv) the number of vehicles used in each depot D_k does not exceed depot capacity r_k;

v) the sum of the costs associated with the duty of the used vehicles is a minimum (unused vehicles do not contribute to the overall cost).

[1] The results of this chapter appear in: M. Fischetti, A. Lodi, P. Toth, "A Branch-and-Cut Algorithm for the Multiple Depot Vehicle Scheduling Problem", *Technical Report* OR/99/1, DEIS - Università di Bologna, [68].

Depending on the possible definition of the above costs, the objective of the optimization is to minimize:

a) the number of vehicles used in the optimal solution, if $\overline{\gamma}_{kj} = 1$ and $\tilde{\gamma}_{jk} = 0$ for each trip T_j and each depot D_k, and $\gamma_{ij} = 0$ for each feasible pair (T_i, T_j);

b) the overall cost, if the values (γ_{ij}), $(\overline{\gamma}_{kj})$ e $(\tilde{\gamma}_{jk})$ are the operational costs associated with the vehicles, including penalities for dead-heading trips, idle times, etc.;

c) any combination of a) and b).

MD-VSP is NP-hard in the general case, whereas it is polynomially solvable if $m = 1$. It was observed in Carpaneto, Dell'Amico, Fischetti and Toth [33] that the problem is also polynomially solvable if the costs $\overline{\gamma}_{kj}$ and $\tilde{\gamma}_{jk}$ are independent of the depots.

Several exact algorithms for the solution of MD-VSP have been presented in the literature, which are based on different approaches. Carpaneto, Dell'Amico, Fischetti and Toth [33] proposed a Branch-and-Bound algorithm based on additive lower bounds. Ribeiro and Soumis [140] studied a column generation approach, whereas Forbes, Holt and Watts [75] analyzed a three-index integer linear programming formulation. Bianco, Mingozzi and Ricciardelli [25] introduced a more effective set-partitioning solution scheme based on the explicit generation of a suitable subset of duties; although heuristic in nature, this approach can provide a provably-optimal output in several cases. Heuristic algorithms have been proposed, among others, by Dell'Amico, Fischetti and Toth [48]. Both exact and heuristic approaches were recently proposed by Löbel [112, 113] for constrained versions of the problem.

In this chapter we consider a branch-and-cut [126] approach to solve MD-VSP to proven optimality, in view of the fact that branch-and-cut methodology proved very successful for a wide range of combinatorial problems; see e.g. the recent annotated bibliography of Caprara and Fischetti [31].

The chapter is organized as follows. In Section 11.2 we discuss a graph theory and an integer linear programming model for MD-VSP. In Section 11.3 we propose a basic class of valid inequalities for the problem, and in Section 11.3.1 we address the associated separation problem. A second class of inequalities is introduced in Section 11.4 along with a separation heuristic. Our branch-and-cut algorithm is outlined in Section 11.5. In particular, we describe an effective branching scheme in which the branching variable is chosen according to the concept of "fractionality persistency", a completely general criterion that can be extended to other combinatorial problems. In Section 11.6 we report extensive computational experiments on a test-bed made by 135 randomly generated and real-world test instances, all of which are available on the web page `http://www.or.deis.unibo.it/ORinstances/`. Some conclusions are finally drawn in Section 11.7.

11.2 Models

We consider a directed graph $G = (V, A)$ defined as follows. The set of vertices $V = \{1, \ldots, m + n\}$ is partitioned into two subsets: the subset $W = \{1, \ldots, m\}$ containing a vertex k for each depot D_k, and the subset $N = \{m + 1, \ldots, m + n\}$ in which each vertex $m + j$ is associated with a different trip T_j. We assume that graph G is complete, i.e., $A = \{(i, j) : i, j \in V\}$. Each arc (i, j) with $i, j \in N$ corresponds to a transition between trips

T_{i-m} and T_{j-m}, whereas arcs (i,j) with $i \in W$ (respectively, $j \in W$) correspond to the start (resp., to the end) of a vehicle duty. Accordingly, the cost associated with each arc (i,j) is defined as:

$$c_{ij} = \begin{cases} \gamma_{i-m,j-m} & \text{if } i,j \in N; \\ \overline{\gamma}_{i,j-m} & \text{if } i \in W,\ j \in N; \\ \tilde{\gamma}_{i-m,j} & \text{if } i \in N,\ j \in W; \\ 0 & \text{if } i,j \in W,\ i = j; \\ +\infty & \text{if } i,j \in W,\ i \neq j. \end{cases}$$

Note that arcs with infinite cost correspond to infeasible transitions, hence they could be removed from the graph (we keep them in the arc set only to simplify the notation). Moreover, the subgraph obtained from G by deleting the arcs with infinite costs along with the vertices in W is acyclic.

By construction, each finite-cost subtour visiting (say) vertices $k, v_1, v_2, \ldots, v_h, k$, where $k \in W$ and $v_1, \ldots, v_h \in N$, corresponds to a feasible duty for a vehicle located in depot D_k that covers consecutively trips $T_{v_1-m}, \ldots, T_{v_h-m}$, the subtour cost coinciding with the cost of the associated duty. Finite-cost subtours visiting more than one vertex in W, instead, correspond to infeasible duties starting and ending in different depots.

MD-VSP can then be formulated as the problem of finding a min-cost set of subtours, each containing exactly one vertex in W, such that all the trip-vertices in N are visited exactly once, whereas each depot-vertex $k \in W$ is visited at most r_k times.

The above graph theory model can be reformulated as an integer linear programming model as in Carpaneto, Dell'Amico, Fischetti and Toth [33]. Let decision variable x_{ij} assume value 1 if arc $(i,j) \in A$ is used in the optimal solution of MD-VSP, and value 0 otherwise.

$$v(MD-VSP) = \min \sum_{i \in V} \sum_{j \in V} c_{ij}\, x_{ij} \tag{11.1}$$

$$\sum_{i \in V} x_{ij} = r_j, \qquad j \in V \tag{11.2}$$

$$\sum_{j \in V} x_{ij} = r_i, \qquad i \in V \tag{11.3}$$

$$\sum_{(i,j) \in P} x_{ij} \leq |P| - 1, \qquad P \in \Pi \tag{11.4}$$

$$x_{ij} \geq 0 \quad \text{integer}, \qquad i,j \in V \tag{11.5}$$

where we have defined $r_i := 1$ for each $i \in N$, and Π denotes the set of the inclusion-minimal *infeasible paths*, i.e., the simple and finite-cost paths connecting two different depot-vertices in W.

The degree equations (11.2) and (11.3) impose that each vertex $k \in V$ must be visited exactly r_k times. Notice that variables x_{kk} ($k \in W$) act as slack variables for the constraints (11.2)-(11.3) associated with k, i.e., x_{kk} gives the number of unused vehicles in depot D_k.

Constraints (11.4) forbid infeasible subtours, i.e., subtours visiting more than one vertex in W. Finally, constraints (11.5) state the nonnegativity and integrality conditions on the variables; because of (11.2)-(11.3), they also imply $x_{ij} \in \{0,1\}$ for each arc (i,j) incident with at least one trip-vertex in N.

In the single-depot case ($m = 1$), set Π is empty and model (11.1)-(11.5) reduces to the well-known Transportation Problem (TP), hence it is solvable in $O(n^3)$ time.

11.3 Path Elimination Constraints (PECs)

The exact solution of MD-VSP can be obtained through enumerative techniques whose effectiveness strongly depends on the possibility of computing, in an efficient way, tight lower bounds on the optimal solution value. Unfortunately, the continuous relaxation of model (11.1)-(11.5) typically yields poor lower bounds. In this section we introduce a new class of constraints for MD-VSP, called Path Elimination Constraints, which are meant to replace the weak constraints (11.4) forbidding infeasible subtours.

Let us consider any nonempty $Q \subset W$, and define $\overline{Q} := W \setminus Q$. Given any finite-cost integer solution x^* of model (11.1)-(11.5), let

$$A^* := \{(i,j) \in A : x^*_{ij} \neq 0\}$$

denote the multiset of the arcs associated with the solution, in which each arc (k,k) with $k \in W$ appears x^*_{kk} times. As already observed, A^* defines a collection of $\sum_{k=1}^{m} r_k$ subtours of G, $\sum_{k=1}^{m} x^*_{kk}$ of which are loops and correspond to unused vehicles.

Now suppose removing from G (and then from A^*) all the vertices in $\overline{Q}$, thus breaking a certain number of subtours in A^*. The removal, however, cannot affect any subtour visiting the vertices $k \in Q$, hence A^* still contains $\sum_{k \in Q} r_k$ subtours through the vertices $k \in Q$. This property leads to the following *Path Elimination Constraints* (PECs):

$$\sum_{i \in S \cup Q} \sum_{j \in (N \setminus S) \cup Q} x_{ij} \geq \sum_{k \in Q} r_k, \quad \text{for each} \quad S \subseteq N,\ S \neq \emptyset. \tag{11.6}$$

Note that the variables associated with the arcs incident in $\overline{Q}$ do not appear in the constraint.

By subtracting constraint (11.6) from the sum of the equations (11.3) for each $i \in Q \cup S$, we obtain the following equivalent formulation of the path elimination constraints:

$$\sum_{i \in Q} \sum_{j \in S} x_{ij} + \sum_{i \in S} \sum_{j \in S} x_{ij} + \sum_{i \in S} \sum_{j \in \overline{Q}} x_{ij} \leq |S|, \quad \text{for each} \quad S \subseteq N,\ S \neq \emptyset, \tag{11.7}$$

where we have omitted the left-hand-side term $\sum_{i \in Q} \sum_{j \in \overline{Q}} x_{ij}$ as it involves only infinite-cost arcs. This latter formulation generally contains less nonzero entries than the original one in the coefficient matrix, hence it is preferable for computation.

Constraints (11.7) state that a feasible solution cannot contain any path starting from a vertex $a \in Q$ and ending in a vertex $b \in \overline{Q}$. This condition is then related to the one expressed by constraints (11.4). However, PEC constraints (11.7) dominate the weak constraints (11.4). Indeed, consider any infeasible path $P = \{(a, v_1), (v_1, v_2), \ldots, (v_{t-1}, v_t), (v_t, b)\}$, where $a, b \in W$, $a \neq b$, and $S := \{v_1, \ldots, v_t\} \subseteq N$. Let Q be any subset of W such that $a \in Q$ and $b \notin Q$. The constraint (11.4) corresponding to path P has the same right-hand side value as in the PEC associated with sets S and Q (as $|P| = t+1$ and $|S| = t$), but each left-hand side coefficient in (11.4) is less or equal to the corresponding coefficient in the PEC.

We finally observe that, for any given pair of sets S and Q, the corresponding PEC does not change by replacing S with $\overline{S} := N \setminus S$ and Q with $\overline{Q} := W \setminus Q$. Indeed, the PEC for pair $(\overline{S}, \overline{Q})$ can be obtained from the PEC associated with (S, Q) by subtracting equations (11.2) for each $j \in S \cup \overline{Q}$, and by adding to the result equations (11.3) for $i \in \overline{S} \cup \overline{Q}$. As a result, it is always possible to halve the number of relevant PECs by imposing, e.g., $1 \in Q$.

11.3.1 PEC Separation Algorithms

Given a family $\mathcal{F}$ of valid MD-VSP constraints and a (usually fractional) solution $x^* \geq 0$, the separation problem for $\mathcal{F}$ aims at determining a member of $\mathcal{F}$ which is violated by x^*. The exact or heuristic solution of this problem is of crucial importance for the use of the constraints of family $\mathcal{F}$ within a branch-and-cut scheme. In practice, the separation algorithm tries to determine a large number of violated constraints, chosen from among those with large degree of violation. This usually accelerates the convergence of the overall scheme.

In the following we denote by $G^* = (V, A^*)$ the support graph of x^*, where $A^* := \{(i,j) \in A : x^*_{ij} \neq 0\}$.

Next we deal with the separation problem for the PEC family. Suppose, first, that the subset $Q \subseteq W$ in the PEC has been fixed. The separation problem then amounts to finding a subset $S \subseteq N$ maximizing the violation of PEC (11.6) associated with the pair (S, Q). We construct a flow network obtained from G^* as follows:

1. for each $w \in W$, we add to G^* a new vertex w', and we let $W' := \{w' : w \in W\}$;
2. we replace each arc $(i, w) \in A^*$ entering a vertex $w \in W$ with the arc (i, w'), and define $x^*_{iw'} := x^*_{iw}$ and $x^*_{iw} := 0$;
3. we define the capacity of each arc $(i, j) \in A^*$ as x^*_{ij};
4. we add two new vertices, s (source) and d (sink);
5. for each $w \in Q$, we add two arcs with very large capacity, namely (s, w) and (w', d).

By construction:

- the flow network is acyclic;
- no arc enters vertices $w \in \overline{Q} := W \setminus Q$, and no arc leaves vertices $w' \in W'$;
- for each $w \in W$, the network contains an arc (w, w') with capacity x^*_{ww}.
- the network depends on the chosen Q only for the arcs incident with s and d (steps 1–4 being independent of Q).

One can easily verify that a minimum-capacity cut in the network with shore (say) $\{s\} \cup Q \cup S$ corresponds to the most violated PEC (11.6) (among those for the given Q). Therefore, such a highly violated PEC cut can be determined, in $O(n^3)$ time, through an algorithm that computes a maximum flow from the source s to the sink d in the network. In practice, the computing time needed to solve such a problem is much less than in the worst case, as A^* is typically very sparse and contains only $O(n)$ arcs.

As to the choice of the set Q, one possibility is to enumerate all the $2^{m-1}-1$ proper subsets of W that contain vertex 1. In that way, the separation algorithm requires, in the worst case, $O(2^{m-1}n^3)$ time, hence it is still polynomial for any fixed m. In practice, the computing time is acceptable for values of m not larger than 5. For a greater number of depots, a possible heuristic choice consists of enumerating only the subsets of W with $|W| \leq \mu$, where parameter μ is set e.g. to 5.

Once a PEC is detected, we refine it by fixing its trip-node set S and by re-computing (through a simple greedy scheme) the depot-vertex set Q so as to maximize the degree of violation.

Preliminary computational experiments showed that the lower bounds obtained through the separation algorithm described are very tight, but often require a large computing time because the number of PECs generated at each iteration is too small. It is then very important to be able to identify a relevant number of violated PECs at each round of separation.

PEC decomposition

A careful analysis of the PECs generated through the max-flow algorithm showed that they often "merge" several violated PECs defined on certain subsets of S. A natural idea is therefore to decompose a violated PEC into a series of PECs with smaller support.

To this end, let S and Q be the two subsets corresponding to a most-violated PEC (e.g., the one obtained through the max-flow algorithm). Consider first the easiest case in which x^* is integer, and contains a collection of $q \geq 2$ paths $P_1, \ldots P_q$ starting from a vertex in Q, visiting some vertices in S, and ending in a vertex in $\overline{Q}$. Now, consider the subsets $S_1, \ldots, S_q \subseteq S$ containing the vertices in S visited by the paths $P_1, \ldots, P_q$, respectively. It is easy to see that all the q PECs associated to the subsets $S_1, \ldots, S_q$ are violated (assuming that S is inclusion minimal, and letting Q be unchanged). Even if it is not possible to establish a general dominance relation between the new PECs and the original PEC, our computational results showed that this refining procedure guarantees a faster convergence of the branch-and-cut algorithm.

When x^* is fractional the refining of the original PEC is obtained in a similar way, by defining $S_1, \ldots, S_q$ as the connected components of the undirected counterpart of the subgraph of G^* induced by the vertex set S.

Infeasible path enumeration

A second method to increase the number of violated PECs found by the separation scheme consists in enumerating the paths contained in G^* so as to identify infeasible paths of the form $P = \{(a, v_1), (v_1, v_2), \ldots, (v_{t-1}, v_t), (v_t, b)\}$ with $a, b \in W$, $a \neq b$, and such that the corresponding constraint (11.7) is violated for $S := \{v_1, \ldots, v_t\} \subseteq N$ and $Q := \{a\}$. Since the graph G^* is typically very sparse, this enumeration usually needs acceptable computing times. According to our computational experience, enumeration is indeed very fast, although it is unlike to identify violated PECs for highly fractional solutions.

PECs with nested support

The above separation procedures are intended to identify a number of violated PEC chosen on the basis of their individual degree of violation, rather than on an estimate of their combined effect. However, it is a common observation in cutting plane methods that the effectiveness of a set of cuts belonging to a certain family depends heavily on their overall action, an improved performance being gained if the separation generates certain highly-effective patterns of cuts.

A known example of this behavior is the travelling salesman problem (TSP), for which commonly-applied separation schemes based on vertex shrinking, besides reducing the com-

putational effort spent in each separation, have the important advantage of producing at each call a noncrossing family of violated subtour elimination constraints.

A careful analysis of the PECs having a nonzero dual variable in the optimal solution of the LP relaxation of our model showed that highly-effective patterns of PECs are typically associated with sets S defining an almost nested family, i.e., only a few pairs S cross each other. We therefore implemented the following heuristic "shrinking" mechanism to force the separation to produce violated PECs with nested sets S.

For each given depot subset Q, we first find a minimum-capacity cut in which the shore of the cut containing the source node, say $\{s\} \cup Q \cup S_1$, is minimal with respect to set inclusion. If violated, we store the PEC associated with S_1, and continue in the attempt at determining, for the same depot subset Q, additional violated PECs associated with sets S strictly containing S_1. This is achieved by increasing to a very large value the capacity of all network arcs having both terminal vertices in $Q \cup S_1$, and by re-applying the separation procedure in the resulting network (for the same Q) so as to hopefully produce a sequence of violated PECs associated with nested sets $S_1 \subset S_2 \cdots \subset S_t$.

In order to avoid stalling on the same cut, at each iteration we increase slightly (in a random way) the capacity of the arcs leaving the shore $\{s\} \cup Q \cup S_i$ of the current cut. In some (rare) cases, this random perturbation step needs to be iterated in order to force the max-flow computation to find a new cut.

As shown in the computational section, the simple scheme above proved very successful in speeding up the convergence of the cutting-plane phase.

11.4 Lifted Path Inequalities (LPIs)

The final solution x^* that we obtain after separating all the PECs can often be expressed as a linear combination of the characteristic vectors of feasible subtours of G. As an illustration, suppose that x^* can be obtained as the linear combination, with 1/2 coefficients, of the characteristic vectors of the following three feasible subtours (among others):

$$C_1 = \{(a, i_1), (i_1, i_2), (i_2, a)\},$$

$$C_2 = \{(a, i_1), (i_1, i_3), (i_3, a)\},$$

$$C_3 = \{(b, i_2), (i_2, i_3), (i_3, b)\},$$

where $a, b \in W$, $a \neq b$, and i_1, i_2 and i_3 are three distinct vertices in N (see Figure 11.1).

Notice that, because of the degree equations on the trip-nodes, only one of the above three subtours can actually be selected in a feasible solution.

The solution x^* of our arc-variable formulation then has: $x^*_{ai_1} \geq 1/2+1/2 = 1$, $x^*_{i_1i_2} \geq 1/2$, $x^*_{i_2i_3} \geq 1/2$, $x^*_{i_1i_3} \geq 1/2$, $x^*_{i_3b} \geq 1/2$, hence it violates the following valid inequality, obtained as a reinforcement of the obvious constraint forbidding the path (a, i_1), (i_1, i_2), (i_2, i_3), (i_3, b):

$$x_{ai_1} + x_{i_1i_2} + x_{i_2i_3} + x_{i_3b} + 2x_{i_1i_3} \leq 3. \tag{11.8}$$

The example shows that constraints of type (11.8) can indeed improve the linear model that includes all degree equations and PECs. As a result, the lower bound achievable by

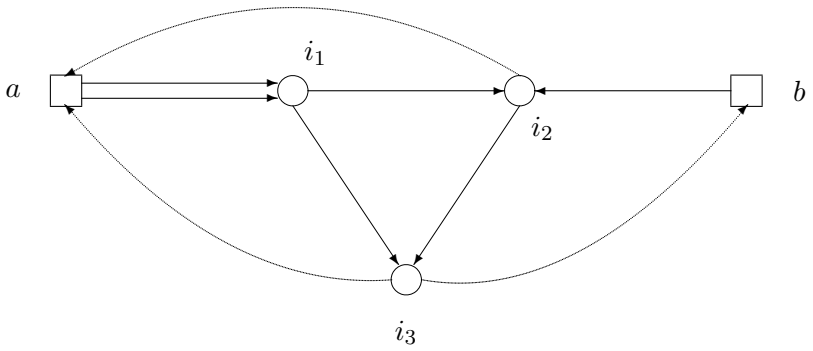


Figure 11.1: A possible fractional point x^* with $x^*_{ij} = 1/2$ for each drawn arc.

means of (11.8) can be strictly better than those obtainable through the set-partitioning or the 3-index formulations from the literature [25, 75, 140].

Constraints (11.8) can be improved and generalized, thus obtaining a more general family of constraints that we call *Lifted Path Inequalities* (LPIs):

$$\sum_{i\in Q_a}\sum_{j\in I_1\cup I_3} x_{ij} + \sum_{i\in I_1}\sum_{j\in I_2\cup Q_b} x_{ij} + \sum_{i\in I_2}\sum_{j\in I_2\cup I_3} x_{ij} + \sum_{i\in I_3}\sum_{j\in Q_b} x_{ij} + 2\sum_{i\in I_1}\sum_{j\in I_1\cup I_3} x_{ij} +$$

$$2\sum_{i\in I_3}\sum_{j\in I_3} x_{ij} \le 3 + 2(|I_1|-1) + (|I_2|-1) + 2(|I_3|-1), \tag{11.9}$$

where (Q_a, Q_b) is any proper partition of W, whereas I_1, I_2 and I_3 are three pairwise disjoint and nonempty subsets of N.

Validity of LPIs follows from the fact that they are rank-1 Chvátal-Gomory cuts obtained by combining the following valid MD-VSP inequalities:

1/3 times PEC($I_1 \cup I_2 \cup I_3, Q_a$),

2/3 times PEC($I_1 \cup I_3, Q_a$),

1/3 times SEC(I_1),

2/3 times SEC(I_2),

1/3 times SEC(I_3),

2/3 times CUT-OUT(I_1),

2/3 times CUT-IN(I_3),

where PEC(S, Q) is the inequality (11.7) associated to the sets $S \subseteq N$ and $Q \subset W$, whereas for each $S \subseteq N$ we denote by SEC(S), CUT-OUT(S) and CUT-IN(S) the following obviously valid (though dominated) constraints:

$$\begin{aligned} \text{SEC}(S) &: \sum_{i\in S}\sum_{j\in S} x_{ij} \le |S|-1 \\ \text{CUT-OUT}(S) &: \sum_{i\in S}\sum_{j\in V\setminus S} x_{ij} \le |S| \\ \text{CUT-IN}(S) &: \sum_{i\in V\setminus S}\sum_{j\in S} x_{ij} \le |S| \end{aligned}$$

A separation algorithm for the "basic" LPIs (11.9) having $|I_1| = |I_2| = |I_3| = 1$ is obtained by enumerating all the possible triples of trip-vertices, and by choosing the partition (Q_a, Q_b) that maximizes the degree of violation of the corresponding LPI. In practice, the computing time needed for this enumeration is rather short, provided that simple tests are implemented to avoid generating triples that obviously cannot correspond to violated constraints.

For the more general family, we have implemented a shrinking procedure that contracts into a single vertex all paths made by arcs (i, j) with $i, j \notin W$ and $x^*_{ij} = 1$, and then applies to the shrunk graph the above enumeration scheme for basic LPIs.

11.5 A Branch-and-Cut Algorithm

In this section we present an exact branch-and-cut algorithm for MD-VSP, which follows the general framework proposed by Padberg and Rinaldi [126]; see Caprara and Fischetti [31] for a recent annotated bibliography.

The algorithm is a lowest-first enumerative procedure in which lower bounds are computed by means of an LP relaxation that is tightened, at run time, by the addition of cuts belonging to the classes discussed in the previous sections.

11.5.1 Lower Bound Computation

At each node of the branching tree, we initialize the LP relaxation by taking all the constraints present in the last LP solved at the father node. For the root node, instead, only the degree equations (11.2)-(11.3) are taken, and an optimal LP basis is obtained through an efficient code for the min-sum Transportation Problem.

After each LP solution we call, in sequence, the separation procedures described in the previous section that try to generate violated cuts. At each round of separation, we check both LPIs and PECs for violation. The constraint pool is instead checked only when no new violated cut is found. In any case, we never add more than $NEWCUTS = 15$ new cuts to the current LP.

Each detected PEC is first refined, and then added to the current LP (if violated) in its $\le$ form (11.7), with pair (S, Q) complemented if this produces a smaller support. In order to avoid adding the same cut twice we use a hashing-table mechanism.

A number of tailing-off and node-pausing criteria are used. In particular we abort the current node and branch if the current LP solution is fractional, and one (at least) of the following conditions hold:

1. we have applied the pricing procedure more than 50 times at the root node, or more than 10 times at the other nodes.

2. the (rounded) lower bound did not improve in the last 10 iterations;

3. the current lower bound exceeds by more than 10 units (a hard-wired parameter) the best lower bound associated with an active branch-decision node; in this situation, the current node is suspended and re-inserted (with its current lower bound) in the branching queue.

According to our computational experience, a significant speed-up in the convergence of the cutting plane phase is achieved at each branching node by using an "aggressive" cutting policy consisting in replacing the extreme fractional solution x^* to be separated by a new point y^* obtained by moving x^* towards the interior of the polytope associated to the current LP relaxation; see Figure 11.2 for an illustration. A similar idea was proposed by Reinelt [139].

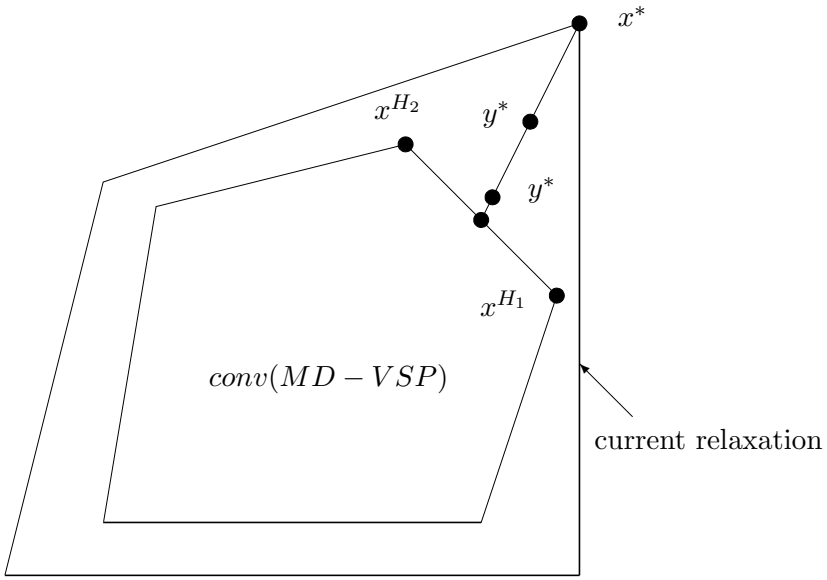


Figure 11.2: Moving the fractional point x^* towards the integer hull $conv(MD-VSP)$.

In our implementation, the point y^* is obtained as follows. Let x^{H_1} and x^{H_2} denote the incidence vector of the best and second-best feasible MD-VSP found, respectively. We first define the point

$$y^* = 0.1\, x^* + 0.9\ (x^{H_1} + x^{H_2})/2$$

and give it on input to the separation procedures in order to find cuts which are violated by y^*. If the number of generated cuts is less than $NEWCUTS$, we re-define y^* as

$$y^* = 0.5\, x^* + 0.5\ (x^{H_1} + x^{H_2})/2$$

and re-apply the separation procedures. If again we did not obtain a total of $NEWCUTS$ valid cuts, the classical separation with respect to x^* is applied.

11.5.2 Pricing

We use a pricing/fixing scheme akin to the one proposed in Fischetti and Toth [70] to deal with highly degenerated primal problems. A related method, called Lagrangian pricing, was proposed independently by Löbel [112, 113].

The scheme computes the reduced costs associated with the current LP optimal dual solution. In the case of negative reduced costs, the classical pricing approach consists of adding to the LP some of the negative entries, chosen according to their individual values. For highly-degenerated LP's, this strategy may either result in a long series of useless pricings, or in the addition of a very large number of new variables to the LP; see Fischetti and Toth [70] for a discussion of this behavior.

The new pricing scheme, instead, uses a more clever and "global" selection policy, consisting of solving on the reduced-cost matrix the Transportation Problem (TP) relaxation of MD-VSP. Only the variables belonging to the optimal TP solution are then added to the LP, along with the variables associated with an optimal TP basis and some of the variables having zero reduced-cost after the TP solution; see [70] for details.

Important by-products of the new separation scheme are the availability of a valid lower bound even in the case of negative reduced costs, and an improved criterion for variable fixing.

In order to save computing time, the Transportation Problem is not solved if the number of negative reduced-cost arcs does not exceed $\max\{50, n\}$, in which case all the negative reduced-cost arcs are added to the current LP.

As to the pricing frequency, we start by applying our pricing procedure after each LP solution. Whenever no variables are added to the current LP, we skip the next 9 pricing calls. In this way we alternate dynamically between a pricing frequency of 1 and 10. Of course, pricing is always applied before leaving the current branching node.

11.5.3 Branching

Branching strategies play an important role in enumerative methods. After extensive computational testing, we decided to use a classical "branch-on-variables" scheme, and adopted the following branching criteria to select the arc (a, b) corresponding to the fractional variable x^*_{ab} of the LP solution to branch with. The criteria are listed in decreasing priority order, i.e., the criteria are applied in sequence so as to filter the list of the arcs that are candidates for branching.

1. *Degree of fractionality*: Select, if possibile, an arc (a, b) such that $0.4 \leq x^*_{ab} \leq 0.6$.

2. *Fractionality persistency*: Select an arc (a, b) whose associated x^*_{ab} was persistently fractional in the last optimal LP solutions. The implementation of this criterion requires initializing $f_{ij} = 0$ for all arcs (i, j), where f_{ij} counts the number of consecutive optimal LP solutions for which x^*_{ij} is fractional. After each LP solution, we set $f_{ij} = f_{ij} + 1$ for all fractional x^*_{ij}'s, and set $f_{ij} = 0$ for all integer variables. When branching has to take place, we compute $f_{max} := \max f_{ij}$, and select a branching variable (a, b) such that $f_{ab} \geq 0.9 f_{max}$.

3. *1-paths from a depot*: Select, if possible, an arc (a, b) such that vertex a can be reached from a depot by means of a 1-path, i.e, of a path only made by arcs (i, j) with $x^*_{ij} = 1$.

4. *1-paths to a depot*: Select, if possible, an arc (a, b) such that vertex b can reach a depot by means of a 1-path.

5. *Heuristic recurrence*: Select an arc (a, b) that is often chosen in the heuristic solutions found during the search. The implementation of this mechanism is similar to that used for fractionality persistency. We initialize $h_{ij} = 0$ for all arcs (i, j), where h_{ij} counts the number of times arc (i, j) belongs to an improving heuristic solution. Each time a new heuristic solution improving the current upper bound is found, we set $h_{ij} = h_{ij} + 1$ for each arc (i, j) belonging to the new incumbent solution. When branching has to take place, we select a branching variable (a, b) such that h_{ab} is a maximum.

11.5.4 Upper Bound Computation

An important ingredient of our branch-and-cut algorithm is an effective heuristic to detect almost-optimal solutions very early during the computation. This is very important for large instances, since in practical applications the user may need to stop code execution before a provably-optimal solution is found. In addition, the success of our "aggressive" cutting plane policy depends heavily on the early availability of good heuristic solutions x^{H_1} and x^{H_2}.

We used the MD-VSP heuristic framework proposed by Dell'Amico, Fischetti and Toth [48], which consists of a constructive heuristic based on shortest-path computations on suitably-defined arc costs, followed by a number of refining procedures.

The heuristic is applied after each call of the pricing procedure, even if new variables have been added to the current LP. In order to exploit the primal and the dual information available after each LP/pricing call, we drive the heuristic by giving on input to it certain modified arc costs c'_{ij} obtained from the original costs as follows:

$$c'_{ij} = \overline{c}_{ij} - 100\, x^*_{ij}$$

where $\overline{c}_{ij}$ are the (LP or TP) reduced costs defined within the pricing procedure, and x^* is the optimal LP solution of the current LP. Variables fixed to zero during the branching correspond to very large arc costs c'_{ij}. Of course, the modified costs c'_{ij} are used only during the constructive part of the heuristic, whereas the refining procedures always deal with the original costs c_{ij}.

11.6 Computational Experiments

The overall algorithm has been coded in FORTRAN 77 and run on a Digital Alpha 533 MHz. We used the CPLEX 6.0 package to solve the LP relaxations of the problem.

The algorithm has been tested on both randomly generated problems from the literature and real-world instances.

In particular, we have considered test problems randomly generated so as to simulate real-world public transport instances, as proposed in [33] and considered in [25, 48, 140]. All the times are expressed in minutes. Let $\rho_1, \cdots, \rho_\nu$ be the ν *relief points* (i.e., the points where trips can start or finish) of the transport network. We have generated them as uniformly random points in a (60×60) square and computed the corresponding travel times θ_{ab} as the Euclidean distance (rounded to the nearest integer) between relief points a and b. As for the trip generation, we have generated for each trip T_j $(j = 1, \cdots, n)$ the starting and ending relief points, ρ'_j and ρ''_j respectively, as uniformly random integers in $(1, \nu)$. Hence we

have $\tau_{ij} = \theta_{\rho''_i \rho'_j}$ for each pair of trips T_i and T_j. The starting and ending times, s_j and e_j respectively, of trip T_j have been generated by considering two classes of trips: *short trips* (with probability 40%) and *long trips* (with probability 60%).

(i) Short trips: s_j as uniformly random integer in (420, 480) with probability 15%, in (480, 1020) with probability 70%, and in (1020, 1080) with probability 15%, e_j as uniformly random integer in $(s_j + \theta_{\rho'_j \rho''_j} + 5, s_j + \theta_{\rho'_j \rho''_j} + 40)$;

(ii) Long trips: s_j as uniformly random integer in (300, 1200) and e_j as uniformly random integer in $(s_j + 180, s_j + 300)$. In addition, for each long trip T_j we impose $\rho''_j = \rho'_j$.

As for the depots, we have considered three values of m, $m \in \{2, 3, 5\}$. With $m = 2$, depots D_1 and D_2 are located at the opposite corners of the (60×60) square. With $m = 3$, D_1 and D_2 are in the opposite corners while D_3 is randomly located in the (60×60) square. Finally, with $m = 5$, D_1, D_2, D_3 and D_4 are in the four corners whereas D_5 is located randomly in the (60×60) square. The number r_k of vehicles stationed at each depot D_k has been generated as a uniformly random integer in $(3 + n/(3m), 3 + n/(2m))$.

The costs have been obtained as follows:

(i) $\gamma_{ij} = \lfloor 10\,\tau_{ij} + 2(s_j - e_i - \tau_{ij}) \rfloor$, for all compatible pairs (T_i, T_j);

(ii) $\overline{\gamma}_{k,j} = \lfloor 10\,(\text{Euclidean distance between } D_k \text{ and } \rho'_j) \rfloor + 5000$, for all D_k and T_j;

(iii) $\tilde{\gamma}_{j,k} = \lfloor 10\,(\text{Euclidean distance between } \rho''_j \text{ and } D_k) \rfloor + 5000$, for all T_j and D_k.

The addition of a big value of 5000 to the cost of both the arcs starting and ending at a depot (cases (ii) and (iii) above) copes with the aim of considering as an objective of the optimization the minimization of both the number of used vehicles and the sum of the operational costs (see Section 11.1).

Five values of n, $n \in \{100, 200, 300, 400, 500\}$, have been considered, and the corresponding value of ν is a uniformly random integer in $(n/3, n/2)$.

In Table 11.1, we consider the case of 2 depots ($m = 2$). 50 instances have been solved, 10 for each value of $n \in \{100, 200, 300, 400, 500\}$. For each instance, we report the instance identifier (`ID`, built as `m-n-NumberOfTheInstance`, see Appendix A), the percentage gap of both the Transportation Problem (*LB0*) and the improved (*Root*) lower bounds, computed at the root node with respect to the optimal solution value, the number of nodes (*nd*) and the number of PEC (*PEC*) and LPI (*LPI*) inequalities generated along the whole branch-decision tree. The next four columns in Table 11.1 concern the heuristic part of the algorithm: the first and the third give the percentage gaps of the *initial* upper bound (*UB0*) with respect to the initial lower bound (*LB0*) and the optimal solution value (*OPT*), respectively; the second and the fourth columns, instead, give the computing times needed to close to 1% the gaps between the current upper bound (*UB*) with respect to the current lower bound (*LB*) and *OPT*, respectively. In other words, from each pair of columns in this part of the table we obtain an indication of the behavior of the branch-and-cut if it is used as a heuristic: for the first pair the gap is computed on line by comparing the decreasing upper bound (*UB*) with the increasing lower bound (*LB*), whereas for the second pair the computation is off line with respect to the optimal solution value. Finally, the last three columns in Table 11.1 refer to the optimal solution value (*OPT*), to the number of vehicles used in the optimal solution (*nv*), and to the overall computing time (*time*), respectively. Moreover, for each pair (m, n) the

results of the 10 reported instances are summarized in the table by an additional row with the average values of the above-mentioned entries.

Note that the percentage gaps reported in this table and in the following ones are obtained by purging the solution values of the additional costs of the vehicles (2 times 5000, for each used vehicle) in order to have more significant values.

Tables 11.2 and 11.3 report the same information for the cases of 3 and 5 depots, respectively. In particular, 40 instances are shown in Table 11.2, which correspond to four values of $n \in \{100, 200, 300, 400\}$, whereas in Table 11.3 we consider 30 instances corresponding to three values of $n \in \{100, 200, 300\}$.

As expected, the larger the number of depots the harder the instance for our branch-and-cut approach, both in terms of computing times and the number of cuts that need to be generated. The number of branching nodes, instead, increases only slightly with m.

The behavior of the algorithm as a heuristic is quite satisfactory, in that short computing time is needed to reduce to 1% the gap between the heuristic value UB and the optimal solution value. For the case of 2 depots, this is not surprising as the initial heuristic of Dell'Amico, Fischetti and Toth [48] is already very tight; for the other cases ($m \in \{3, 5\}$), the information available during the cutting-plane phase proved very important to drive the heuristic. The overall scheme also exhibits a good behavior as far as the speed of improvement of the lower bound is concerned, which is important to provide an on-line performance guaranteed of the heuristic quality.

In Table 11.4, the instances of the previous tables are aggregated in classes given by the pair (m, n), and the average computing times in Tables 11.1-11.3 are decomposed by considering the main parts of the branch-and-cut algorithm. In particular, we consider the time spent for solving the linear programming relaxations (LP), the pricing time (PRI), the separation time (SEP), the time spent at the root node ($ROOT$) and the time spent for the heuristic part of the algorithm ($HEUR$). Finally, the last two columns compare the computing times obtained by our branch-and-cut algorithm with those reported in Bianco, Mingozzi and Ricciardelli [25] for their set-partitioning approach (algorithms B&C and BMR, respectively). As a rough estimate, our Digital Alpha 533 MHz is about 50 times faster than the PC 80486/33 Mhz used in [25].

A direct comparison between algorithms B&C and BMR in Table 11.4 is not immediate, since the instances considered in the two studies are not the same. Moreover, for $n \geq 200$ the set-partitioning approach was tested by its authors on only 4 (as opposed to 10) instances, and no instance with $m = 2$ and $n \geq 400$ nor with $m \geq 3$ and $n \geq 400$ was considered by BMR. More importantly, the set-partitioning solution scheme adopted by BMR is heuristic in nature, in that it generates explicitly only a subset of the possible feasible duties, chosen as those having a reduced cost below a given threshold. Therefore its capability of proving the optimality of the set-partitioning solution with respect to the overall (exponential) set of columns depends heavily on the number of columns fitting the threshold criterion, a figure that can be impractically large in some cases.

Table 11.5 presents the results obtained on a subset of the instances (chosen as the largest ones), by disabling in turn one of the following branch-and-cut mechanisms: the fractionality-persistency criterion within the branching rule ("without FP"), the convex combination of heuristic solutions when cutting the fractional point ("without CC"), the generation of nested cuts within PEC separation ("without NC"). Finally, the last two columns in Table 11.5 give the results obtained by using a basic branch-and-cut algorithm ("basic B&C") that incorporates none of the above tools. In the table, the first two columns identify, as usual,

Table 11.1: Randomly generated instances with $m = 2$; computing times are in Digital Alpha 533 MHz seconds.

ID	% Gap LB					% Gap	time to 1%	% Gap	time to 1%			
	$LB0$	$Root$	nd	PEC	LPI	$\frac{UB0-LB0}{LB0}$	$\frac{UB-LB}{LB}$	$\frac{UB0-OPT}{OPT}$	$\frac{UB-OPT}{OPT}$	OPT	nv	$time$
2-100-01	0.2444	0.0000	1	102	9	1.69	0.05	1.44	0.03	279463	25	0.35
2-100-02	0.9400	0.0000	1	120	11	1.30	0.29	0.35	0.02	301808	27	0.38
2-100-03	0.9198	0.0000	1	126	9	2.91	0.32	1.96	0.22	341528	31	0.45
2-100-04	1.7379	0.0000	1	170	19	3.23	0.34	1.44	0.32	289864	26	0.67
2-100-05	2.5750	0.0000	1	277	26	4.04	1.14	1.36	0.09	328815	30	1.45
2-100-06	0.8830	0.0000	1	68	8	0.96	0.01	0.07	0.00	360466	33	0.28
2-100-07	0.5411	0.0000	1	102	7	1.51	0.01	0.97	0.00	290865	26	0.42
2-100-08	1.0636	0.0000	1	118	13	3.61	0.40	2.51	0.22	337923	31	0.50
2-100-09	0.6601	0.0000	1	223	20	2.26	0.59	1.59	0.07	270452	24	0.90
2-100-10	0.4427	0.0000	1	170	15	0.89	0.01	0.45	0.00	291400	26	0.63
Average	1.0008	0.0000	1.0	147.6	13.7	2.24	0.32	1.21	0.10	–	27.9	0.60
2-200-01	0.5599	0.0000	1	392	30	0.94	0.09	0.38	0.08	545188	49	5.23
2-200-02	0.8168	0.0000	3	668	22	2.03	3.80	1.20	1.03	617417	56	13.58
2-200-03	1.4321	0.0123	14	839	41	2.22	4.23	0.75	0.10	666698	61	26.73
2-200-04	0.2559	0.0000	1	319	38	1.29	0.34	1.03	0.14	599404	54	4.17
2-200-05	0.6807	0.0045	3	1354	49	1.46	4.20	0.77	0.07	626991	56	27.73
2-200-06	0.6262	0.0000	1	312	30	1.60	3.08	0.96	0.10	592535	54	5.15
2-200-07	0.8525	0.0441	7	3467	79	1.28	3.34	0.42	0.07	611231	55	77.43
2-200-08	0.7922	0.0231	4	2595	61	1.87	4.68	1.06	2.00	586297	53	61.02
2-200-09	0.4396	0.0000	1	634	32	1.53	2.03	1.09	1.40	596192	54	9.10
2-200-10	0.4629	0.0000	1	261	29	0.84	0.06	0.37	0.05	618328	56	2.88
Average	0.6919	0.0084	3.6	1084.1	41.1	1.51	2.59	0.80	0.50	–	54.8	23.30
2-300-01	1.0487	0.0169	23	4778	71	2.75	22.19	1.67	8.62	907049	83	349.38
2-300-02	0.6277	0.0025	3	1306	74	1.44	9.13	0.81	0.48	789658	71	46.30
2-300-03	0.2890	0.0123	19	1234	67	1.31	3.61	1.02	0.66	813357	74	61.12
2-300-04	0.6514	0.0000	1	1312	51	1.37	12.91	0.71	0.33	777526	70	51.37
2-300-05	0.4559	0.0000	1	557	46	1.61	9.75	1.15	7.47	840724	76	19.25
2-300-06	0.5946	0.0205	5	1499	50	1.24	6.18	0.64	0.23	828200	75	66.55
2-300-07	0.4223	0.0090	3	1200	49	1.14	4.30	0.72	0.12	817914	74	30.67
2-300-08	0.5443	0.0000	1	880	60	1.53	1.08	0.97	0.15	858820	78	33.02
2-300-09	0.6855	0.0073	3	1902	68	1.57	11.54	0.88	0.27	902568	82	77.20
2-300-10	0.8440	0.0142	3	2580	55	1.70	14.00	0.84	0.55	797371	72	106.72
Average	0.6163	0.0083	6.2	1724.8	59.1	1.57	9.47	0.94	1.89	–	75.5	84.16
2-400-01	0.4177	0.0058	7	5559	95	1.26	12.34	0.84	1.02	1084141	98	431.27
2-400-02	0.6690	0.0000	1	3153	81	1.76	24.42	1.08	6.75	1028509	93	171.45
2-400-03	0.8149	0.0000	1	2530	127	1.85	31.27	1.02	1.47	1152954	105	137.85
2-400-04	0.7740	0.0107	5	5593	86	1.89	20.16	1.10	5.71	1112589	101	412.78
2-400-05	0.7163	0.0306	9	7743	89	1.46	19.61	0.73	0.78	1141217	104	670.77
2-400-06	0.3347	0.0000	1	1270	79	1.19	5.12	0.85	0.77	1100988	100	61.57
2-400-07	1.3563	0.0000	1	4175	111	2.67	76.70	1.28	6.13	1237205	113	398.30
2-400-08	0.5709	0.0000	1	2569	74	1.46	25.05	0.88	0.43	1111077	101	158.92
2-400-09	0.8082	0.0000	1	4286	90	2.34	79.45	1.51	13.95	1104559	100	410.67
2-400-10	0.6185	0.0021	3	2444	72	1.84	27.33	1.21	4.70	1086040	99	125.85
Average	0.7081	0.0049	3.0	3932.2	90.4	1.77	32.15	1.05	4.17	–	101.4	297.94
2-500-01	0.5132	0.0051	5	10994	112	2.26	58.38	1.74	26.06	1296920	118	1222.15
2-500-02	0.5425	0.0115	22	19595	126	0.84	0.99	0.29	0.98	1490681	136	2667.48
2-500-03	0.6780	0.0059	5	7540	151	2.10	77.52	1.41	35.77	1328290	121	854.77
2-500-04	0.4815	0.0032	3	12196	185	1.47	67.62	0.98	0.70	1373993	125	1351.38
2-500-05	0.4315	0.0008	5	7928	143	1.29	26.53	0.85	1.38	1315829	119	807.68
2-500-06	0.6797	0.0017	14	11265	113	1.61	57.39	0.92	0.92	1358140	124	1155.47
2-500-07	0.8368	0.0063	3	5175	103	2.53	141.60	1.67	66.40	1436202	131	1025.73
2-500-08	0.5110	0.0000	1	2941	64	1.59	52.30	1.07	2.09	1279768	116	356.93
2-500-09	0.6671	0.0000	1	5331	163	1.47	74.96	0.79	2.98	1462176	134	588.92
2-500-10	0.7041	0.0008	3	8085	95	1.96	86.75	1.24	13.28	1390435	127	1576.82
Average	0.6045	0.0035	6.2	9105.0	125.5	1.71	64.40	1.10	15.06	–	125.1	1160.73

Table 11.2: Randomly generated instances with $m = 3$; computing times are in Digital Alpha 533 MHz seconds.

	% Gap LB					% Gap	time to 1%	% Gap	time to 1%			
ID	$LB0$	$Root$	nd	PEC	LPI	$\frac{UB0-LB0}{LB0}$	$\frac{UB-LB}{LB}$	$\frac{UB0-OPT}{OPT}$	$\frac{UB-OPT}{OPT}$	OPT	nv	$time$
3-100-01	1.4330	0.0938	11	867	12	3.60	2.14	2.12	1.35	307705	28	9.22
3-100-02	1.1900	0.0000	1	222	12	1.50	0.97	0.30	0.02	300505	27	1.05
3-100-03	1.2729	0.0000	1	441	14	1.78	0.80	0.48	0.02	316867	29	2.22
3-100-04	2.3361	0.0000	1	468	13	2.72	1.05	0.32	0.02	336026	31	2.37
3-100-05	0.5087	0.0000	1	223	12	2.32	0.10	1.80	0.10	278896	25	1.25
3-100-06	2.4235	0.0035	3	419	19	2.91	1.35	0.42	0.02	368925	34	2.35
3-100-07	1.5778	0.0000	1	368	14	2.80	2.48	1.18	0.08	287190	26	2.78
3-100-08	2.4476	0.0000	1	436	10	4.61	1.53	2.05	0.66	338436	31	3.55
3-100-09	1.3260	0.0000	1	270	9	1.34	0.42	0.00	0.02	275943	25	1.13
3-100-10	2.8307	0.0000	1	306	12	4.98	1.95	2.01	1.02	285930	26	2.03
Average	1.7346	0.0097	2.2	402.0	12.7	2.86	1.28	1.07	0.33	–	28.2	2.80
3-200-01	0.9718	0.0832	14	4108	49	2.69	10.86	1.69	3.60	551657	50	151.05
3-200-02	1.1254	0.0502	25	3943	59	2.39	3.68	1.24	0.35	543805	50	124.93
3-200-03	1.2151	0.0000	1	290	15	2.63	3.68	1.38	3.41	615675	57	7.18
3-200-04	2.2455	0.0169	3	2752	34	4.62	16.40	2.27	5.12	557339	51	112.22
3-200-05	1.1319	0.0000	1	1692	33	2.03	6.49	0.88	0.22	626364	57	55.12
3-200-06	0.9749	0.0000	1	405	12	2.40	3.60	1.40	1.32	558414	51	6.65
3-200-07	1.5283	0.0044	3	1053	24	3.75	7.80	2.16	2.45	595605	55	33.48
3-200-08	1.2196	0.0000	1	779	24	1.99	6.65	0.74	0.08	562311	51	15.22
3-200-09	1.7184	0.0549	11	4553	19	2.91	13.31	1.14	5.51	671037	62	196.08
3-200-10	1.1409	0.0000	1	1308	43	3.30	6.73	2.12	2.23	565053	52	25.50
Average	1.3272	0.0210	6.1	2088.3	31.2	2.87	7.92	1.50	2.43	–	53.6	72.74
3-300-01	0.9527	0.0047	7	1778	32	2.21	23.63	1.23	1.35	834240	77	87.43
3-300-02	1.0743	0.0185	20	10943	77	2.94	30.31	1.84	9.60	830089	76	706.75
3-300-03	1.9330	0.0117	3	3358	44	4.55	34.40	2.53	8.95	799803	74	286.57
3-300-04	1.2872	0.0042	3	2260	44	2.90	46.55	1.58	14.59	850929	78	166.17
3-300-05	1.0288	0.0222	5	5264	26	2.97	55.72	1.92	10.77	837460	77	576.20
3-300-06	0.9292	0.0000	1	2758	33	2.63	21.13	1.67	10.37	795110	73	142.05
3-300-07	0.5823	0.0013	3	2276	43	2.03	21.72	1.43	1.34	774873	70	138.10
3-300-08	1.2559	0.0045	3	2739	26	3.51	76.69	2.21	20.62	916484	85	261.42
3-300-09	1.3253	0.0282	9	6254	36	3.25	32.35	1.88	10.48	830364	77	560.77
3-300-10	1.0055	0.0199	21	8900	96	2.21	19.15	1.19	5.37	850515	78	472.95
Average	1.1374	0.0115	7.5	4653.0	45.7	2.92	36.17	1.75	9.34	–	76.5	339.84
3-400-01	1.5358	0.0074	5	10679	65	3.83	211.20	2.24	102.55	1141067	106	3188.92
3-400-02	0.4626	0.0167	13	21240	97	1.18	7.83	0.71	0.60	1059717	97	1617.23
3-400-03	0.6149	0.0053	8	14811	79	1.30	50.25	0.68	1.37	1124169	103	2205.48
3-400-04	1.1152	0.0246	35	24730	74	2.68	66.53	1.53	25.66	1091238	101	5142.95
3-400-05	0.7706	0.0000	1	4548	65	2.11	61.78	1.33	3.47	1159027	107	429.15
3-400-06	1.2421	0.0195	21	26217	139	2.97	117.21	1.69	17.11	1042121	96	4476.55
3-400-07	1.0737	0.0255	21	25868	111	2.16	83.86	1.06	14.51	1104156	101	4144.12
3-400-08	0.9852	0.0398	43	32159	102	2.21	91.88	1.21	11.98	1050490	97	5480.95
3-400-09	1.1130	0.0000	1	5732	57	2.69	58.85	1.54	38.39	1007810	93	775.32
3-400-10	0.5863	0.0203	32	34646	130	1.48	34.92	0.89	1.10	1063571	98	4315.67
Average	0.9499	0.0159	18.0	20063.0	91.9	2.26	78.43	1.29	21.67	–	99.9	3177.63

Table 11.3: Randomly generated instances with $m = 5$; computing times are in Digital Alpha 533 MHz seconds.

ID	% Gap LB					% Gap	time to 1%	% Gap	time to 1%			
	$LB0$	$Root$	nd	PEC	LPI	$\frac{UB0-LB0}{LB0}$	$\frac{UB-LB}{LB}$	$\frac{UB0-OPT}{OPT}$	$\frac{UB-OPT}{OPT}$	OPT	nv	$time$
5-100-01	3.3840	0.0000	1	738	5	6.28	3.25	2.68	0.68	365591	34	6.87
5-100-02	2.1433	0.0000	1	454	3	4.00	1.64	1.77	0.74	295568	27	2.95
5-100-03	4.6979	0.1824	21	3240	10	7.53	9.60	2.48	9.16	314117	29	58.02
5-100-04	1.5884	0.0552	13	1516	5	2.63	3.17	1.00	0.07	340785	31	25.18
5-100-05	0.9784	0.0000	1	245	1	2.27	0.55	1.27	0.44	306369	28	1.25
5-100-06	4.0112	0.1091	2	1012	5	6.38	3.42	2.11	1.30	333833	31	11.32
5-100-07	3.2928	0.0895	11	1871	8	6.66	6.30	3.15	1.50	296816	27	30.07
5-100-08	2.6971	0.1091	14	1862	14	5.99	6.85	3.13	6.85	355657	33	34.18
5-100-09	1.5456	0.0075	3	581	2	3.20	1.46	1.61	0.88	306721	28	4.58
5-100-10	4.3193	0.2840	24	3499	8	6.19	3.90	1.60	0.60	291832	27	50.48
Average	2.8658	0.0837	9.1	1501.8	6.1	5.11	4.01	2.08	2.22	–	29.5	22.49
5-200-01	2.4575	0.1190	13	8467	8	6.10	83.42	3.49	74.33	619511	58	603.50
5-200-02	2.4653	0.0463	6	2971	3	5.53	19.05	2.93	14.73	601049	56	123.45
5-200-03	1.9709	0.0435	5	4452	4	6.40	44.23	4.30	32.41	623685	58	247.73
5-200-04	5.5508	0.1391	31	12267	14	10.05	82.40	3.94	35.05	622408	58	883.22
5-200-05	2.1769	0.0000	1	4681	2	4.87	45.24	2.59	6.42	597086	55	221.12
5-200-06	1.9155	0.0253	4	3938	1	2.90	28.09	0.93	0.37	479571	44	160.57
5-200-07	2.4430	0.0000	1	2624	2	4.83	27.24	2.27	26.72	553880	51	128.22
5-200-08	1.6582	0.0574	11	6393	0	3.33	70.28	1.62	43.35	595291	55	594.38
5-200-09	1.4916	0.0000	1	3991	0	4.33	45.66	2.77	34.16	588537	54	220.32
5-200-10	1.1019	0.0207	9	4869	16	2.10	14.03	0.97	0.43	593183	54	231.77
Average	2.3232	0.0451	8.2	5465.3	5.0	5.04	45.96	2.58	26.80	–	54.3	341.43
5-300-01	1.3620	0.0139	7	10383	7	3.09	153.03	1.68	9.77	784685	72	2006.65
5-300-02	2.1725	0.0426	13	11445	16	4.55	188.02	2.28	129.37	856341	80	1899.32
5-300-03	2.6642	0.0233	6	14724	5	5.42	319.76	2.61	148.86	900205	84	3040.72
5-300-04	2.1696	0.0072	3	6277	1	4.04	111.28	1.78	109.95	815586	76	847.63
5-300-05	1.9572	0.0393	21	20860	14	4.23	153.60	2.19	153.60	868503	81	4506.17
5-300-06	1.9015	0.0561	9	21257	20	5.17	278.69	3.17	166.24	787059	73	4863.87
5-300-07	1.5106	0.0131	13	13876	5	4.25	129.78	2.67	11.70	811301	75	2799.87
5-300-08	1.8754	0.0576	12	25377	9	4.73	307.65	2.77	67.12	780788	72	5796.38
5-300-09	1.9037	0.0098	6	13507	0	3.79	168.88	1.81	24.00	850934	79	3148.93
5-300-10	2.2229	0.0220	15	15177	8	5.53	179.31	3.19	65.81	819068	76	2395.40
Average	1.9740	0.0285	10.5	15288.3	8.5	4.48	199.00	2.42	88.64	–	76.8	3130.49

the class of instances considered. For each version of the algorithm two columns are given, which report averages (over the 10 instances) on the number of nodes and the computing time, respectively. A time limit of 10,000 CPU seconds has been imposed for each instance. The number of possibly *unsolved* instances within the time limit is given in brackets. The number of nodes and the computing time considered for unsolved instances are those reached at the time limit, i.e., averages are always computed over 10 instances so as to give a lower bound on the worsening of the branch-and-cut algorithm without the considered tools (a larger time limit would have produced even greater worsenings).

The results of Table 11.5 prove the effectiveness of the improvements we proposed in speeding up the branch-and-cut convergence. This is particularly interesting in view of the fact that these rules are quite general and can be applied/extended easily to other problems.

In Figure 11.3 we give an example of how, at the root node, the speed of convergence of the lower bound depends on the different versions of the branch-and-cut algorithm considered in the previous table (except for the version without fractionality-persistency in the branching rule, that of course does not differ from B&C at the root node). The instance `2-500-01` is

Table 11.4: Randomly generated instances: average computing times over 10 instances; CPU seconds on a Digital Alpha 533 MHz.

		Partial Computing Times					Overall	
m	n	LP	PRI	SEP	$ROOT$	$HEUR$	B&C	BMR
	100	0.25	0.10	0.10	0.60	0.07	0.60	81
	200	14.04	2.36	2.97	13.34	0.90	23.30	647*
2	300	52.42	10.91	7.39	46.56	3.83	84.16	755*
	400	195.86	29.08	30.65	225.28	9.42	297.94	–
	500	787.99	100.59	106.97	712.47	35.96	1160.73	–
	100	1.69	0.22	0.33	2.30	0.20	2.80	109
	200	53.03	5.10	5.33	39.80	1.96	72.74	835*
3	300	254.41	21.65	22.07	213.68	9.30	339.84	1472*
	400	2549.43	151.46	160.82	1187.70	44.94	3177.63	–
	100	16.00	1.06	2.29	9.31	0.63	22.49	186
5	200	271.52	10.75	24.45	189.98	8.00	341.43	1287*
	300	2645.93	68.97	133.37	1562.15	45.23	3130.49	1028*

* Average values over 4 instances ([25]; BMR computing times are CPU seconds on a PC 80486/33).

Table 11.5: Randomly generated instances: different versions of the branch-and-cut algorithm. Average computing times over 10 instances; CPU seconds on a Digital Alpha 533 MHz.

		B&C		without FP			without CC			without NC			basic B&C		
m	n	nd	$time$	nd	$time$		nd	$time$		nd	$time$		nd	$time$	
	300	6.2	84.16	7.8	96.01		7.1	94.07		5.3	105.55		9.2	141.69	
2	400	3.0	297.94	9.1	466.32		4.5	325.90		8.4	519.42		31.3	1239.70	
	500	6.2	1160.73	17.5	1691.56		8.1	1238.22		10.3	1389.99		60.3	6171.15	(3)
3	300	7.5	339.84	10.5	422.39		8.7	380.73		8.6	562.11		17.6	509.06	
	400	18.0	3177.63	196.3	5281.56	(4)	188.5	5010.50	(2)	125.1	4395.07	(2)	326.1	6385.34	(6)
5	200	8.2	341.43	16.0	457.92		11.5	615.87		15.8	562.11		20.4	810.90	
	300	10.5	3130.49	24.8	4011.97	(1)	640.9	7716.95	(6)	10.8	3611.57		415.3	7745.74	(5)

considered: the final lower bound at the root node is obtained in 800 CPU seconds by B&C, in 1100 CPU seconds when the nested cuts are not generated, in 1400 CPU seconds when the convex combination is disabled, and in 2300 CPU seconds by the basic B&C.

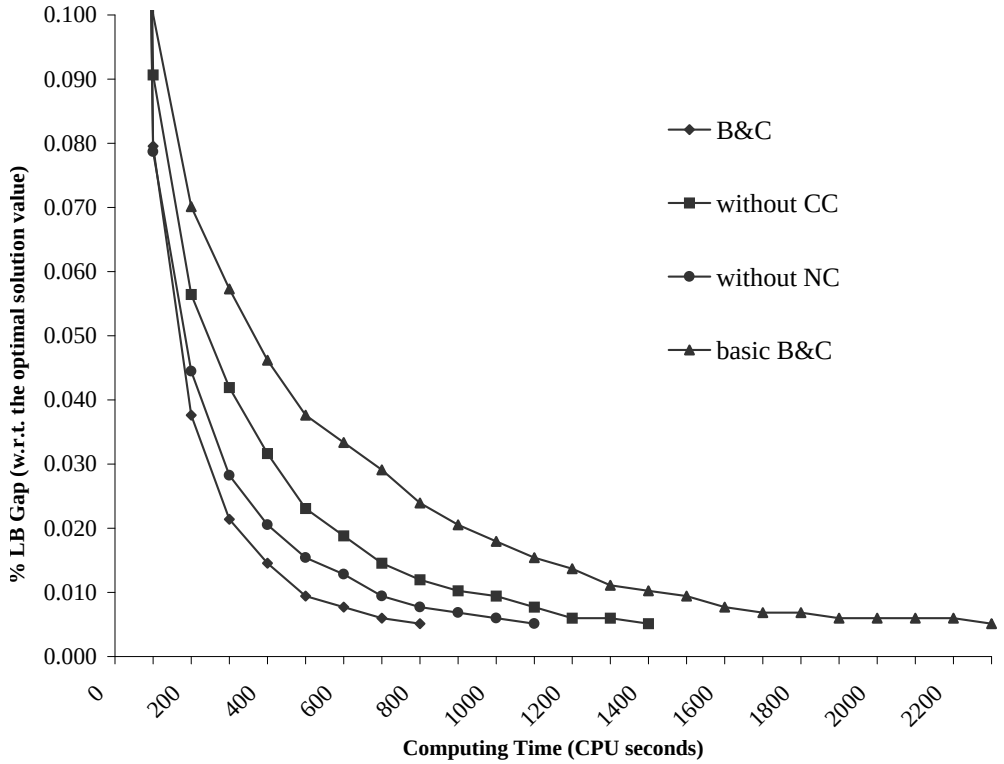


Figure 11.3: Instance `2-500-01`: lower bound convergence at the root node for different versions of the cutting plane generation.

Finally, the table in Figure 11.4 reports the results obtained by the branch-and-cut algorithm on a set of 5 real-world instances (with $n \in \{184, 285, 352, 463, 580\}$) that we obtained from an Italian bus company. The bus company currently has $m = 3$ bus depots to cover the area under consideration, and was interested in simulating the consequences of adding/removing some depots. This "what-if" analysis resulted in 3 instances for each set of trips, each associated with a different pattern of depots (i.e., $m \in \{2, 3, 5\}$). As for the randomly generated instances, a big value of 5000 is added to the cost of each arc visiting a depot.

The entries in the table are the same as in Tables 11.1-11.3. In addition, as in Table 11.4, we report the computing times of the main components of the algorithm.

The real-world instances appear considerably easier to solve for our branch-and-cut algorithm than those considered in the randomly-generated test bed. Indeed, the computing times reported in the table of Figure 11.4 are significantly smaller than those corresponding to random instances, and the number of branching nodes is always very small. In our view, this is mainly due to the increased average number of trips covered by the duty of each vehicle: in the real-world instances of the table in Figure 11.4, each duty covers on average 7-9 trips, whereas for random instances this figure drops to the (somehow unrealistic) value of 3-4 trips per duty. This improved performance is an important feature of our approach,

Figure 11.4: Real-world instances: computing times in Digital Alpha 533 MHz seconds.

ID	% Gap LB		nd	PEC	LPI	% Gap $\frac{UB0-LB0}{LB0}$	time to 1% $\frac{UB-LB}{LB}$	% Gap $\frac{UB0-OPT}{OPT}$	time to 1% $\frac{UB-OPT}{OPT}$	OPT	nv	Computing Times					B&C
	LB0	Root										LP	PRI	SEP	ROOT	HEUR	
2-184-00	0.2471	0.0000	1	175	18	1.26	2.23	1.01	0.16	320304	26	0.4	4.5	0.3	6.0	0.4	6.00
3-184-00	1.1120	0.0000	1	1272	17	2.88	10.62	1.73	6.69	318904	26	9.9	8.7	3.0	28.2	2.4	28.20
5-184-00	1.7884	0.0428	4	2972	4	5.01	85.78	3.13	85.78	316083	26	48.3	122.1	12.9	135.6	8.8	209.30
2-285-00	0.1859	0.0000	1	1243	120	0.86	0.38	0.67	0.38	488767	40	14.8	7.2	6.4	38.4	3.7	38.43
3-285-00	0.5990	0.0127	7	4948	97	2.66	37.45	2.04	34.89	486315	40	109.2	60.2	23.6	153.1	19.9	258.48
5-285-00	0.9542	0.0160	9	15089	23	4.53	146.29	3.53	114.05	481113	40	556.1	187.0	108.4	536.3	79.5	1135.70
2-352-00	0.1080	0.0000	1	1865	76	0.95	0.57	0.84	0.57	541814	44	30.5	13.8	12.2	77.9	8.6	77.88
3-352-00	0.3195	0.0000	1	3483	86	1.82	50.05	1.50	26.47	539221	44	107.8	35.4	25.4	243.3	38.8	243.35
5-352-00	0.6884	0.0118	21	13705	20	3.27	250.27	2.56	199.75	533404	44	1066.2	427.1	137.3	1189.8	194.4	2138.83
2-463-00	0.0420	0.0000	8	11353	498	0.73	0.98	0.69	0.98	660839	53	394.1	79.0	189.0	574.3	41.8	920.93
3-463-00	0.1262	0.0024	7	16614	148	1.49	87.09	1.36	58.05	657555	53	1233.5	165.2	205.6	1179.4	120.5	1974.92
5-463-00	0.2691	0.0033	15	35272	35	2.18	1041.57	1.91	812.77	650382	53	5300.8	650.2	788.7	5719.9	1049.4	8874.92
2-580-00	0.0195	0.0000	1	8508	556	0.30	1.42	0.28	1.42	838643	68	393.4	86.2	223.6	924.8	51.9	924.80
3-580-00	0.0273	0.0000	1	8256	114	0.76	3.52	0.73	3.52	834031	68	618.5	115.5	164.4	1139.9	137.4	1139.95
5-580-00	0.1386	0.0209	1	30578	2	1.38	578.79	1.24	510.60	823549[o]	68	4200.2	519.7	1554.9	10000.0	3125.2	10000.00

[o] Best solution found within the time limit of 10,000 seconds, lower bound value 823519.

in that set-partitioning solution approaches are known to exhibit the opposite behavior, and run into trouble when the number of nonzero entries of each set-partitioning "column" (duty) increases.

11.7 Conclusions

Vehicle scheduling is a fundamental issue in the management of transportation companies. In this chapter we have considered the multiple-depot version of the problem, which belongs to the class of the NP-hard problems.

We argued that a "natural" ILP formulation based on arc variables has some advantages over the classical "set partitionig" or "multi-commodity flow" formulations, commonly used in the literature, mainly for the cases in which only few depots are present.

We addressed a basic ILP formulation based on variables associated with trip transitions, whose LP relaxation is known to produce rather weak lower bounds. We then enhanced substantially the basic model by introducing new families of valid inequalities, for which exact and heuristic separation procedures have been proposed. These results are imbedded into an exact branch-and-cut algorithm, which also incorporates efficient heuristic procedures and new branching and cutting criteria.

The performance of the method was evaluated through extensive computational testing on a test-bed containing 135 random and real-life instances, all of which are made publicly available for future benchmarking.

The outcome of the computational study is that our branch-and-cut method is competitive with the best published algorithms in the literature when 2-3 depots are specified, a situation of practical relevance for medium-size bus companies. As expected, when several depots are present the performance of the method deteriorates due to the very large number of cuts that need to be generated.

The performance of our branch-and-cut method turned out to be greatly improved for real-world instances in which each vehicle duty covers, on average, 7-9 trips (as opposed to the 3-4 trips per duty in the random problems). Evidently, the increased number of trip combinations leading to a feasible vehicle duty has a positive effect on the quality of our model and on the number of cuts that need to be generated explicitly. This behavior is particularly important in practice, in that the performance of set-partitioning methods is known to deteriorate in those cases where each set-partitioning "column" (duty) tends to contain more than 3-5 nonzero entries. Hence our methods can profitably be used to address the cases which are "hard" for set-partitioning approaches.

We have also shown experimentally the benefits deriving from the use of simple cut selection policies (nested cuts and deeper fractional points) and branching criteria (fractionality persistency) on the overall branch-and-cut algorithm performance.

Finally, significant quality improvements of the heuristic solutions provided by the method of Dell'Amico, Fischetti and Toth [48] have been obtained by exploiting the primal and dual information available at early stages of our branch-and-cut code.

Future directions of work include the incorporation in the model of some of the additional constraints arising in practical contexts, including "trip-objections" that make it impossible for some trips to be covered by vehicles of certain pre-specified types or depots.

Appendix 11.A: Format of the instances in the test-bed

The instances on which we tested our algorithm are made publicly available for benchmarking.

The data set is composed by 120 random instances generated as in [33] (see Section 11.6), and by 15 real-world instances. Each instance is associated with a unique identifier, which is a string of the form `m-n-NumberOfTheInstance`. E. g., `ID = 3-200-05` corresponds to the 5-th instance with 3 depots and 200 trips. For real-world instances, the identifier has instead the form `ID = m-n-00`.

For each instance `ID` we distribute two files, namely `ID.cst` and `ID.tim`, containing the *cost matrix* and the *starting* and *ending time vectors* of instance `ID`, respectively.

The first line of each `ID.cst` file contains the $m+2$ entries m, n, and $nv(i)$ for $i = 1, \ldots, m$ (where $nv(i)$ is the number of vehicles available at depot D_i), whereas the next lines give the complete $(n + m) \times (n + m)$ cost matrix, whose entries are listed row-wise. Each file of type `ID.tim` contains the n trip starting-times followed by the n trip ending-times, all expressed in minutes from midnight.

Chapter 12

Other Combinatorial Optimization Problems

12.1 Introduction

In this chapter we consider the *unconstrained Quadratic 0-1 Programming Problem* (QP) and the *Data sets Reconstruction Problem* (DRP). These two combinatorial optimization problems are not directly related to AP, but the approaches used to address them are similar to others presented in the previous chapters, and for this reason have been included in the thesis.

In particular, the evolutionary heuristic developed for QP shares with the tabu search for 2BP presented in Part I the same metahueristic idea: iteratively (heuristically) fixing a fragment of the current solution (a set of bins for 2BP) and trying to improve it by performing Local Search on the remaining part. In the following section, this simple guideline is successfully applied to QP.

Concerning DRP, the exact algorithm developed for it extensively uses the heuristic approach separately proposed, as already done by the branch-and-cut for MD-VSP. This is a very general feature because exact approaches usually take advantages from heuristics, but the DRP example shows once again how a good exploitation of the heuristic rules could be crucial for the exact solution.

12.2 The unconstrained Quadratic 0–1 Programming

The[1] general formulation of the unconstrained Quadratic 0-1 Programming Problem is the following:

$$\min f(x) = c^t x + \frac{1}{2} x^t Q x, \qquad x \in \{0,1\}^n \tag{12.1}$$

where n is the size of the problem, c is a rational n-vector and Q is an $n \times n$ symmetric rational matrix. This problem is NP-hard, indeed it contains as a special case the *Maximum Stable Set Problem*.

QP is a well known and interesting combinatorial optimization problem first because it has many practical applications for example in capital budgeting and financial analysis [120], traffic message management [77] and machine scheduling [4] and second because it is closely

[1] The results in this section appear in: A. Lodi, K. Allemand, T. M. Liebling, "An Evolutionary Euristic for Quadratic 0–1 Programming", *European Journal of Operations Research* 119, 662–670, 1999, [106].

related to other problems in the literature, as for example the *Max Cut Problem* (see below) and the *Quadratic Assignment Problem*.

The first results in the literature regarding QP are due to Hammer [90], who pointed out the equivalence between QP and the *Max Cut Problem*.

Two cases where QP is polynomially solvable are known. The first one is the case in which all the elements of the matrix Q are nonpositive (see Picard and Ratliff [131]) while the second one, shown by Barahona [12], is the case where the graph defined by Q is *series-parallel*. Lower bounds for the problem have been proposed by Hammer, Hansen and Simeone [91] and polyhedral studies have been given by Barahona and Mahjoub [14], Padberg [125] and De Simone [46]. Exact approaches to QP are due to Barahona, Jünger and Reinelt [13] (Branch and Cut) and to Pardalos and Rodgers [128] (Branch and Bound).

In the last years two effective heuristic approaches have been presented: the first proposed by Chardaire and Sutter [39] is based on a decomposition method while the second, based on the Tabu Search technique, has been studied in two different papers by Glover, Kochenberger and Alidaee [80] and Glover, Kochenberger, Alidaee and Amini [81].

This section presents a heuristic approach for QP based on *combining solutions* inside a genetic paradigm. In the following Section we discuss some classic methods from the literature for combining solutions and we briefly present our own combination strategy. In Section 12.2.2 we describe two effective algorithms used to locally optimize the solutions before and after such combinations while in Section 12.2.3 the evolutionary algorithm is presented. Finally, in Section 12.2.4 we discuss a large set of computational experiments comparing our results with the ones of powerful exact and heuristic algorithms from the literature and establishing the effectiveness of our approach in terms of quality of solutions and computing time.

12.2.1 Combining solutions methods

Two of the most known and effective methods for combining solutions in an evolutionary way are Genetic Algorithms (GAs) and Scatter Search (SS) (see Mühlenbein [123] and Glover [79], respectively). Both methods can be outlined, very briefly, as follows:

1. *define* a set P of feasible solutions;
2. *combine* a subset of P in some way;
3. *update* P and repeat until a stopping criterion is reached.

The way to implement the above steps is very different in the two approaches. In this section we refer to an updated version of the classic genetic framework in which the main operators of *selection*, *mutation* and *recombination-crossover* (see Section 12.2.3) are designed to be constructive instead of simply evolutionary (see again Mühlenbein [123] for a complete overview of the evolution of the genetic approach). In particular, the treatment of the solutions in an ordered way and the use of heuristics are features of the new genetic paradigm which were not part of the original one. The addition of these tools brings the new genetic framework close to the Scatter Search approach with the important difference that GAs are still part of mathematical random search methods, while in Scatter Search no random choice is performed during the exploration of the solution space.

The heuristic algorithm we developed, and that will be presented in the following sections, is designed on the genetic framework, but is also indebted to Scatter Search because it makes

an extensive use of intensification algorithms, as is standard in SS and not in GAs. The original idea followed here is to incorporate effective and fast heuristic algorithms inside a well structured method of combining solutions. Such a method is designed to iteratively select a subset of assignments common to high quality solutions, then, letting the variables in this subset fixed (they identify a promising region), we can define a subproblem of reduced size and perform intensification on it. This simple and clean approach is used for the first time to address QP and the results are proved to be satisfactory (see Section 12.2.4).

12.2.2 Intensification algorithms

As mentioned in the previous section, our evolutionary heuristic (EH) for QP makes extensive use of heuristic algorithms to improve solutions and speed up the method. In this section we present two simple algorithms which are very well suited to this aim.

MinRange Algorithm

Let us consider the following property pointed out by Pardalos and Rodgers [128]:

Lemma 12.1. *If x^* is the optimal solution of the quadratic problem*

$$\min f(x) = c^t x + \frac{1}{2} x^t Q x, \qquad x \in S \tag{12.2}$$

where S is a convex set, then x^* is also optimal for the following linear problem:

$$\min g(x) = \nabla^t f(x^*) x, \qquad x \in S \tag{12.3}$$

In the special case where $S = [0,1]^n$, Lemma 12.1. implies that variables whose partial derivatives have fixed sign in $[0,1]^n$ can be forced to 0 or 1. For each component of the gradient of $f(x)$ it is easy to compute a lower and an upper bound:

$$lb_i = c_i + \sum_j q_{ij}^- \leq \frac{\partial f}{\partial x_i} \leq c_i + \sum_j q_{ij}^+ = ub_i \tag{12.4}$$

where q_{ij}^- (resp. q_{ij}^+) are the negative (resp. positive) elements of the i-th row of Q. If $\frac{\partial f}{\partial x_i}$ has fixed positive (resp. negative) sign, then x_i can be forced to value 0 (resp. 1).

It is easy to see that the property shown in Lemma 12.1. remains true in the discrete case where $S = \{0,1\}^n$, i.e. in the QP case.

Using this property we developed a simple algorithm called MinRange (MINimum RANGE of the gradient) utilized as a procedure in the genetic algorithm in order to fix variables. A variable is considered fixed if its value cannot be changed in the following steps of the search. It is *completely* fixed (see below) if its value can never be changed or *temporarily* fixed if its value must remain unchanged only during a given number of steps. The framework of the MinRange algorithm can be outlined as follows.

Algorithm MinRange
1. compute ub_i and lb_i of each unfixed variable i;
2. **repeat**
   2.a try to fix the variables not already fixed to 0 or 1;
   2.b update the ub_i and lb_i of the still unfixed variables i

until ((all the variables are fixed) **or**
(no variable has been fixed in this cycle)).

Updating the bounds of the unfixed variables (step 2.b) is performed as follows: if variable i^* has been forced to 0 (resp. to 1) and variable i is still unfixed then

if $q_{ii^*} > 0$
then $ub_i := ub_i - q_{ii^*}$ (resp. $lb_i := lb_i + q_{ii^*}$)
else $lb_i := lb_i - q_{ii^*}$ (resp. $ub_i := ub_i + q_{ii^*}$).

In a preprocessing phase, i.e. when no variable has been fixed before calling MinRange, it is clear that the variables forced by the MinRange algorithm (if any) will be considered *completely* fixed. Indeed these variables have been forced to their exact value and so the size of the problem becomes smaller. In particular, if MinRange performed in the preprocessing forces μ variables then only $n - \mu$ components of x remain to be fixed in the rest of the global algorithm.

Local Search Algorithm (LS)

Each iteration of the local search algorithm is composed of two phases, a *constructive* phase and a *destructive* one. This terminology is due to Glover et al. [80] who perform very similar phases to identify solutions called *critical events* that represent the base of their tabu strategy.

A *constructive* phase consists in successively setting to 1 the variables (previously set to 0) that lead to an improvement of the objective function. The phase is initialized by calculating the improvement vector imp (imp_i indicates the improvement achievable by setting $x_i := 1$), then at each step the best improvement is selected, the corresponding variable is set to 1, and finally the vector and the objective function are updated. The phase ends if either a prefixed number of variables has been set to 1 or no improvement is possible.

Letting z be the current solution, inf a large positive value and C a prefixed parameter, the scheme of a constructive phase can be described as follows.

for $i := 1$ **to** n **do**
if ($x_i = 0$ **and** i not fixed)
then $imp_i := c_i + \sum_{j=1}^{n} q_{ij} x_j$
else $imp_i := inf$;
$count = 0$;
repeat
if $\min_{i=1,\dots,n} imp_i < 0$
then $i^* := \arg\min_{i=1,\dots,n} imp_i$
else $i^* := 0$;
if ($i^* \neq 0$) **then**
begin
$x_{i^*} := 1$;
$z := z + imp_{i^*}$;
$imp_{i^*} := inf$;
for $i := 1$ **to** n **do**
if ($imp_i \neq inf$) **then** $imp_i := imp_i + q_{ii^*}$;
$count := count + 1$
end
until (($count > C$) **or** ($i^* = 0$)).

It should be noted that $O(n^2)$ time is needed in the initialization to calculate vector *imp* while updating the same vector is performed efficiently in a mere $O(n)$.

A *destructive* phase is clearly analogous; the variables considered are those set to 1 and we try to improve z by setting them to 0.

The constructive and destructive phases are performed consecutively and together represent a single iteration of LS. Since this single iteration is heuristic the result depends from the order in which the two phases are performed. We decided to perform first the constructive phase according to preliminary computational experiments. The algorithm terminates after performing a prefixed number φ of iterations.

12.2.3 An Evolutionary Heuristic (EH)

According to the genetic paradigm discussed in Section 12.2.1, the framework of our evolutionary algorithm can be outlined as follows:

1. generate an *initial population* of different solutions;
2. **repeat**
   3. *select* two solutions in the population, the parents;
   4. generate a new solution by applying a *cross-over* operator to the parents;
   5. the new solution is modified by a *mutation* operator;
   6. *insert* the new solution in the population and *update* the population

   until a *stopping* or *restart criterion* is satisfied
7. **if** stopping criteria are not satisfied **go to** 1.

For each step of this structure we will present below our implementation based on the strategy of efficiently combining strong intensification and timely diversification.

Initial population

The initial population is composed of $SIZE_{min}$ elements. $SIZE_{min} - g$ of these elements are randomly generated and improved through LS, while the remaining g elements are the best solutions found in the g previous restarts performed by the algorithm (clearly $g = 0$ at the beginning).

Stopping and restart criteria

The stopping and restart criteria used are quite standard. The algorithm terminates both if (i) a prefixed number of iterations is performed or (ii) a prefixed time limit is reached. We restart the search by generating a new initial population (see *Initial population* and also *Mutation operator*) after R consecutive iterations without an improvement of the *current best solution*.

Selection operator

The selection operator is implemented according to the choice of Taillard [147]. We consider the population sorted in non-decreasing order of the objective function value. With $SIZE$ denoting the current size of the population and y_1, y_2 two random variables uniformly distributed in $[0, 1]$, the two parents P_1 and P_2 are selected as follows:

- $P_1 = \lfloor y_1^2 \times SIZE + 1 \rfloor$

- $P_2 = \begin{cases} \lfloor y_2^2 \times (SIZE-1)+1 \rfloor & if\ \lfloor y_2^2 \times (SIZE-1)+1 \rfloor < P_1 \\ \lfloor y_2^2 \times (SIZE-1)+2 \rfloor & otherwise \end{cases}$

Note that by squaring y_1 and y_2 the probability of choosing the best solution in the population at each step becomes higher.

Crossing-Over operator

The cross-over phase is intended to perform a strong intensification action. The motto followed is: good solutions are similar. We temporarily fix the variables with the same value in P_1 and P_2 obtaining a *child* (SON) that has a reduced number of unfixed variables and we apply to it the intensification actions (see Section 12.2.2). We illustrate the strategy with the following example.

Example

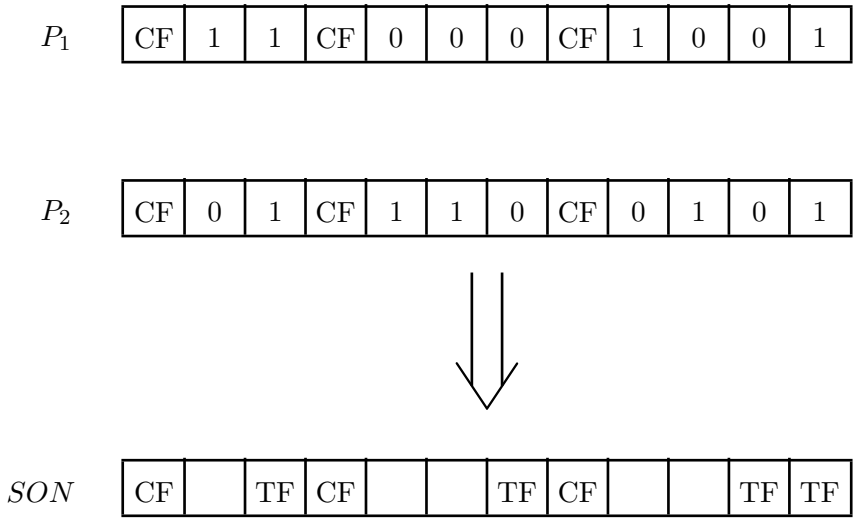


Figure 12.1: Example of crossing-over phase, $n = 12$.

Therein **CF** and **TF** indicate respectively the variables completely fixed (in an exact way in the preprocessing phase) and the temporarily fixed ones (in a heuristic way through comparison of P_1 and P_2).□

The variables **CF** are common to all the elements of the population and so also to SON, while the variables **TF** remain fixed only during the next operations performed on SON. Marking these variables **TF** fixed enables to fix other variables applying the MinRange algorithm to SON. In other words, starting from a partial heuristic solution, we perform MinRange in order to complete it in the best possible way (exactly) and then converge quickly to the local optimum.

Mutation operator

The mutation operator in genetic algorithms is intended to perform an action of diversification. In our algorithm we have two types of diversification:

a. **if** at the end of the cross-over phase not all the variables have been fixed **then** the unfixed variables of SON are randomly set;

b. after T restarts without improvement of the *globally best solution*, we restart with a population containing no element of the previous restarts.

Post-Optimization

Between the phases of mutation and insertion, a post-optimization of the current solution SON is performed by applying algorithm LS.

Insert and Update

After having inserted SON in the current population, we sort the population and eliminate its worst solution if $SIZE > SIZE_{max}$, where $SIZE_{max}$ indicates the limit in the size of the population allowed.

12.2.4 Computational experiments

To test our approach we used a set of problems generated with the generator proposed by Pardalos and Rodgers [128], which has been designed to incorporate the characteristics of many computational experiences presented in the literature.

Test problems

We solved all the instances of the problem set studied by Glover at al. [80]. This set is composed of six different classes of problems up to $n = 500$. Each class is characterized by the range of the elements in the main diagonal and in the off diagonal of the matrix Q (see Table 12.1).

Table 12.1: Classes of QP problems.

name	class	main diagonal	off diagonal
WBRJ	a	+/− 100	+/−100
Gulati	b	−63 – 0	0 – 100
Barahona	c	+/− 100	+/− 50
–	d	+/− 75	+/− 50
–	e	+/− 100	+/− 50
–	f	+/− 75	+/− 50

Each instance is characterized by the number of variables (n), the density of the matrix Q (*dens*) and the seed needed to initialize the random generator of Pardalos and Rodgers (*seed*).

Classes a, b and c are designed according to the profile given by Pardalos and Rodgers for problems called "WBRJ", "Gulati" and "Barahona" respectively and represent the maximal sizes of problems solved in the literature prior to recent works of Glover et al. [80, 81]. For the instances in these classes the optimal solutions can be computed through the Branch and Bound approach of Pardalos and Rodgers (B&B). This algorithm performs very well for the problems of classes a and b (for most of them it requires less than 1 CPU second to find the optimal solution and at most 200 seconds to terminate on a Pentium 90 PC, see [80]), while the instances of class c are already too large for B&B, that needs thousands of seconds to validate the optimal solutions when possible (see again [80]).

The other classes we have tested were first introduced in [80] and the instances have much larger size than the ones solved before the above cited papers.

Globally we use as benchmark the values of B&B for problems of the first three classes (optimal solutions), while for each problem of the remaining classes we take the best value obtained by the different versions (TS-0, TS-1, TS-2 and TS-3) of the Tabu Search approach by Glover et al. described in [81]. Note that for the problems in classes a, b and c each Tabu Search algorithm finds the same solutions of B&B, i.e., the optimal one, while for the other problems none of the Tabu Search algorithms described in [81] strictly dominates the others.

In particular, Table 12.2 presents the results of our approach (EH) on instances of the first three classes, comparing with algorithm B&B . For each problem, in addition to n, *dens* and *seed*, we indicate the number of variables that we are able to fix (*completely*) through MinRange in the preprocessing phase, the best value found by EH, the number of iterations and the CPU time (expressed in seconds) to find this solution (Itb and Ttb indicate *Iter to best* and *Time to best* respectively) and finally the value provided by B&B.

Table 12.3 presents the results for the remaining classes of problems and contains the same information of Table 12.2. We also include a column (Rtb) indicating in which restart the best solution has been found, a column that is not indicated in Table 12.2 because all the best solutions were found in the beginning iterations. In this table, as mentioned above, the benchmark value for each problem is the best known solution (BKS in Table 12.3) provided by the different Tabu Search algorithms.

Implementation details

The evolutionary heuristic (EH) was coded in ANSI C and run on a Silicon Graphics INDY R10000sc 195Mhz. The stopping criterion used for these experiments was a prefixed time limit that was adapted according to the size of the instances. In particular, a limit of 1 CPU second was assigned to the instances of classes a, b, c and d while for classes e and f the time limit becomes respectively of 5 and 60 CPU seconds.

The default values for the parameters introduced are the following:

- $C = 50$ and $\varphi = 1$ (LS);
- $SIZE_{min} = 50$, $SIZE_{max} = 80$, $R = 50$ and $T = 3$ (EH).

In order to obtain a more effective intensification action for the large instances of classes e and f the number of iterations of algorithm LS is increased by 1 ($\varphi = 2$) and, only for the instances of the class f, the size of the initial population ($SIZE_{min}$) and the number of non-improving iterations allowed before a restart (R) grow up to 60 and 80 respectively.

Table 12.2: Computational results on problems of classes a, b and c. Time limit of 1 CPU second on a Silicon Graphics INDY R10000sc.

ID	n	$dens$	$seed$	$fixed$	EH	Itb	Ttb	$B\&B$
1a	50	.1	10	16	−3414	2	0.02	−3414
2a	60	.1	10	18	−6063	0	0.03	−6063
3a	70	.1	10	8	−6037	32	0.08	−6037
4a	80	.1	10	5	−8598	0	0.06	−8598
5a	50	.2	10	3	−5737	2	0.03	−5737
6a	30	.4	10	0	−3980	0	0.01	−3980
7a	30	.5	10	0	−4541	0	0.01	−4541
8a	100	.0625	10	52	−11109	2	0.06	−11109
1b	20	1.0	10	0	−133	2	0.00	−133
2b	30	1.0	10	0	−121	0	0.01	−121
3b	40	1.0	10	1	−118	1	0.02	−118
4b	50	1.0	10	1	−129	0	0.02	−129
5b	60	1.0	10	0	−150	2	0.03	−150
6b	70	1.0	10	0	−146	2	0.03	−146
7b	80	1.0	10	0	−160	1	0.04	−160
8b	90	1.0	10	1	−145	16	0.06	−145
9b	100	1.0	10	1	−137	18	0.09	−137
10b	125	1.0	10	3	−154	3	0.12	−154
1c	40	.8	10	0	−5058	0	0.01	−5058
2c	50	.6	70	0	−6213	0	0.02	−6213
3c	60	.4	31	0	−6665	2	0.04	−6665
4c	70	.3	34	0	−7398	2	0.05	−7398
5c	80	.2	8	0	−7362	0	0.07	−7362
6c	90	.1	80	8	−5824	2	0.07	−5824
7c	100	.1	142	20	−7225	5	0.10	−7225

Table 12.3: Computational results on problems of classes d, e and f, time limit of 1, 5 and 60 CPU seconds respectively on a Silicon Graphics INDY R10000sc.

ID	n	$dens$	$seed$	$fixed$	EH	Rtb	Itb	Ttb	BKS
1d	100	.1	31	4	−6333	0	0	0.09	−6333
2d	100	.2	37	0	−6579	0	10	0.12	−6579
3d	100	.3	143	0	−9261	0	9	0.12	−9261
4d	100	.4	47	0	−10727	0	21	0.14	−10727
5d	100	.5	31	0	−11626	1	142	0.50	−11626
6d	100	.6	47	0	−14207	0	4	0.11	−14207
7d	100	.7	97	0	−14476	0	16	0.14	−14476
8d	100	.8	133	0	−16352	0	10	0.12	−16352
9d	100	.9	307	0	−15656	0	54	0.21	−15656
10d	100	1.0	1311	0	−19102	0	2	0.11	−19102
1e	200	.1	51	0	−16464	1	141	2.47	−16464
2e	200	.2	43	0	−23395	0	9	0.50	−23395
3e	200	.3	34	0	−25243	0	5	0.45	−25243
4e	200	.4	73	0	−35594	0	18	0.63	−35594
5e	200	.5	89	0	−35154	0	19	0.66	−35154
1f	500	.10	137	0	−61194	2	449	43.06	−61194
2f	500	.25	137	0	−100161	2	465	43.47	−100161
3f	500	.50	137	0	−138035	4	588	59.33	−138035
4f	500	.75	137	0	−172771	0	71	8.80	−172771
5f	500	1.0	137	0	−190507	4	566	59.07	−190507

Computational results

The behavior of EH has proved to be quite satisfactory.

For all instances in classes a, b and c we find the *optimal solution* in a very short computing time and in many cases we are able to find this solution already by simply improving the solutions of the initial population by LS ($Itb = 0$ in Table 12.2).

For problems in classes d, e and f EH finds the best known solution. In particular, for problem 4f EH has been the first algorithm to find the solution of value -172771 which is known as the best one (instead of the previous one of value -172734 referred in the early versions of [80, 81]). The computing time needed clearly increases with the size of the problems (up to a 500×500 dense matrix) but it remains very small even for instances of class f. For these large problems the restart (i.e., search diversification) actions become more and more effective.

The comparison between EH and the Tabu Search algorithms is difficult. As mentioned in Section 12.2.4, none of these algorithms can be considered as dominant also because the time limits given to each of them are quite different since the stopping criterion for each algorithm is not the computing time but the number of cycles. In particular we reported the average times of each algorithm for 40 cycles (stopping condition) for classes d, e and f in Table 12.4, where the computing times are referred to a Pentium 200 PC. In the same table we indicate in the last column the time limit for EH on a Silicon Graphics INDY R10000sc 195Mhz. (To our knowledge the computing times on the above machines are quite comparable.)

Table 12.4: Average time limits of the Tabu Search algorithms (Pentium 200 PC seconds) and time limit of EH (Silicon Graphics INDY R10000sc seconds).

Classes	TS-0	TS-1	TS-2	TS-3	EH
d	4	4	15	10	1
e	8	8	43	24	5
f	21	21	320	133	60

It can be noted that for some instances of classes a, b and c the MinRange algorithm performed as preprocessing phase is quite effective (up to 52 variables completely fixed) while for the remaining classes only in one case (problem 1d) it is possible to reduce the size of the instance.

12.3 The Data sets Reconstruction Problem

In [2] this section we consider the problem of selecting a subset of samples from a given data set so as to minimize the overall reconstruction error of the complete set. This problem has several practical applications, and in particular we focus on applications arising in the biomedical field. The first one is the X-Ray Computed Tomography (CT) examination of long bones. Given a *safe* amount of radiation dose absorbed by a patient, corresponding to a prefixed number of CT scans, the problem is to conveniently locate such scans so as to maximize the

[2]The results of this section appear in: A. Lodi, D. Vigo, C. Zannoni, "Exact and Heuristic Algorithms for Data sets Reconstruction", *European Journal of Operational Research* 124, 139–150, 2000, [111].

information obtained. The second application arises in the Electrocardiography (ECG) data acquisition, where a relevant problem is to reduce as much as possible the memory space required to store the ECG signal without loosing significant diagnostic information. In both cases, the same optimization problem occurs: given a finite major set of information (i.e., the possible positions of the CT scanner and the ECG signal samples) it is necessary to select a subset of it with prefixed cardinality, in order to reconstruct, with a minimum error, the overall information.

More formally, the Data set Reconstruction Problem may be described as follows. We are given a data set S defined by n samples. Each sample $i = 1, \ldots, n$, is associated with a reference *coordinate*, x_i (e.g., representing the time instant or the position of the sample), and the corresponding *value*, $\mathbf{F}(x_i)$. In the one-dimensional (1D) case $\mathbf{F}(x_i)$ is a scalar value, indicated as $f(x_i)$, whereas in the two-dimensional (2D) case $\mathbf{F}(x_i)$ is a vector of p values, indicated as $f_j(x_i)$ $(j = 1, \ldots, p)$. Analogously, in the three-dimensional (3D) case $\mathbf{F}(x_i)$ is a two-dimensional array of $p \cdot q$ values, indicated as $f_{jh}(x_i)$ $(j = 1, \ldots, p; h = 1, \ldots, q)$. Examples of the one- and two-dimensional cases are the previously mentioned ECG and CT applications, respectively. We assume that samples are numbered according to increasing values of the coordinate x_i.

The problem consists of determining a subset $T = \{t_1, \ldots, t_k\} \subset S$ of k samples, such that $t_1 = 1$, $t_k = n$, and $t_1 < t_2 \ldots < t_{k-1} < t_k$. The objective is to minimize the total *reconstruction error*, E^T, of data set S.

Given the subset T, the *reconstructed value* $\mathbf{F}^T(x_i)$ is computed for each sample in S. The total error E^T is a function of the reconstruction errors, e_i^T, for each sample in S, where e_i^T is in turn a function of the original and reconstructed values at coordinate x_i, such that $e_h^T = 0$ for each $h \in T$.

In this section, as normally happens both in the literature and in practical applications, we consider the case in which the reconstructed values are computed by *linear interpolation* (see, Figure 12.2). Each reconstructed value $\mathbf{F}^T(x_i)$ may be easily computed by using the two consecutive original samples t_h and t_{h+1} such that $i \in [t_h, t_{h+1})$. For example, in the 1D case, we have

$$f^T(x_i) = f(x_{t_h}) + \frac{f(x_{t_{h+1}}) - f(x_{t_h})}{x_{t_{h+1}} - x_{t_h}}(x_i - x_{t_h}). \tag{12.5}$$

Moreover, the total reconstruction error is represented by the *root mean square error* (RMSE)

$$E^T = \sqrt{\frac{\sum_{i=1}^{n}(e_i^T)^2}{n}} \tag{12.6}$$

where

$$e_i^T = f^T(x_i) - f(x_i). \tag{12.7}$$

Analogously, in the 2D case, for each $i \in [t_h, t_{h+1})$, the reconstructed value $f_j(x_i)$ $(j = 1, \ldots, p)$ may be obtained as in (12.5), by replacing $f^T(\cdot)$ with $f_j^T(\cdot)$, and $f(\cdot)$ with $f_j(\cdot)$. The resulting total reconstruction error is given by (12.6), where

$$e_i^T = \sqrt{\frac{\sum_{j=1}^{p}(f_j^T(x_i) - f_j(x_i))^2}{p}}. \tag{12.8}$$

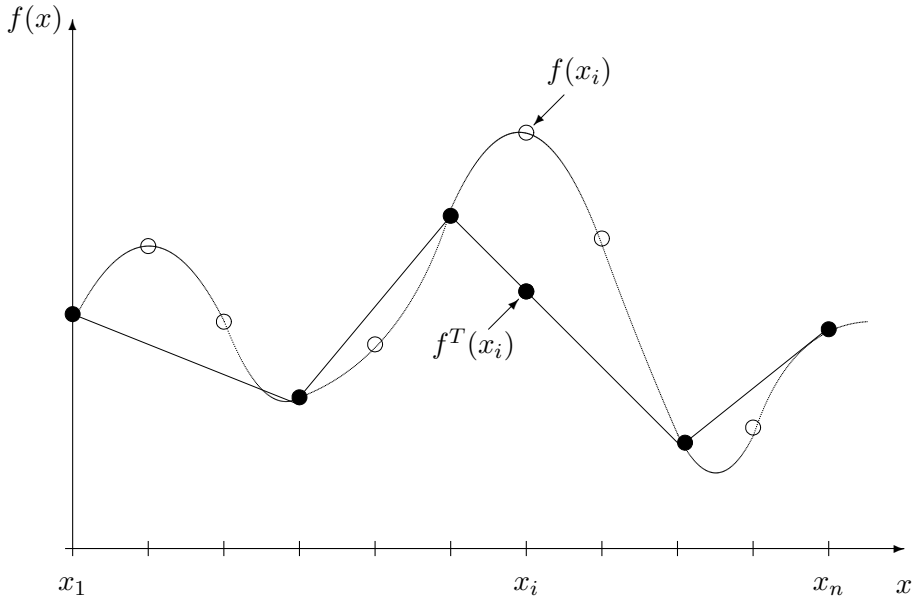


Figure 12.2: Example of reconstruction by linear interpolation of a 1D data set with $n = 11$ and $k = 5$.

In Section 12.3.1 the biomedical applications which motivate the present study are discussed, whereas in Section 12.3.2 possible mathematical models are presented. The exact and heuristic approaches based on dynamic programming are described in Section 12.3.3 and 12.3.4, respectively. Finally, computational experiments on a large set of both real-world and randomly generated instances are presented in Section 12.3.5, and conclusions are drown in Section 12.3.6.

12.3.1 The biomedical applications

The biomedical application which mainly motivates the present work regards the 3D reconstruction of long bones from Computed Tomography (CT) images.

Various orthopaedic research and clinical applications require the analysis of the 3D stress field in bone structures. This information is useful to investigate structural and functional behavior of human bones and can be accomplished through 3D Finite Element Modeling (3D FE). The advent of X-ray computed tomography provided transaxial images containing accurate description of bone geometry and tissue density allowing for the generation of 3D models of human bones *in-vivo*. Due to the risks related to X-ray absorption, however, the number of CT images acquired in-vivo should be kept as low as possible and their position should be selected in order to maximize the geometry and density accuracy of 3D FE models. One of the clinical applications which require the accurate reconstruction of bone morphology is the design of custom made prostheses. These implants are needed in patients affected by severe hip joint malformations altering the normal anatomy. In this case, during CT data acquisition, radiologists simply apply standard protocols provided by the implant producers, suggesting an uniform scanning plan. Zannoni, Cappello and Viceconti [150], analyzed the possibility to improve 3D modeling through the automatic positioning of the CT scans according to geometry and density gradients. In clinical practice the position of the CT scans is

decided by the radiologists on the basis of an X-ray anterior-posterior projection of the bone segment under examination (the so-called *scout image*, see Figure 12.3). In [150] a heuristic algorithm, called DIEOM, has been applied to the scout image to improve the selection of the CT scans for 3D reconstruction. In order to validate the approach based on the optimization on scout image, two different scanning plans have been computed. The DIEOM algorithm has been applied on the 2D scout image and on the 3D data set, on average 430 CT images acquired *in-vitro* with the minimum scanning step available, 1 mm, of three femurs. 3D reconstruction was performed starting from the 2D optimized, 3D optimized scanning plan and the standard protocol. The reconstruction error, RMSE, was evaluated by comparing the reconstructed and the original data sets. The major result was that the 2D optimized scanning plan reduces the 3D reconstruction error of about 20% with respect to the standard protocol. Direct 3D optimization accounts for a slight further reduction of about 4%, at the expense of a dramatic increase of the computational effort. This shows that the approach based on the 2D optimization on the scout image is a good way to improve 3D reconstruction quality of long bones.

The second biomedical application we consider is related to the problem of compressing Electrocardiography (ECG) signal in order to efficiently reduce the storage space. Such a problem is relevant due to the high redundancy of the ECG signals. Haugland, Heber and Husøy [93] formulated this problem as a one-dimensional DRP (1D-DRP). In [93] a mathematical programming model of the problem is introduced and a classic dynamic programming approach is adapted to exactly solve it. The computational tests performed in [93] showed that this exact approach is not directly usable in on-line compression of the ECG signals. Indeed, in practical applications the compression time must be smaller than one second, whereas this exact approach requires computing times about one order of magnitude larger. Nevertheless, the exact approach can be fruitfully used to measure the quality of on-line heuristic methods.

In Section 12.3.5 we consider the application of our exact and heuristic methods to real-world CT scanning optimization and ECG signal compression problems.

The present work was carried out within the European Commission funded project named PROMISE (Pre and Post Processing of Medical Images on High performance Architectures). The results of PROMISE project, including the algorithms described in this section, are available through WWW (http://www.cineca.it/visit/promise/index.html) as decision support tools for radiology units.

12.3.2 Problem definition and modeling

The problem of determining the set of k samples which minimizes the reconstruction error of a given data set, made up by $n >> k$ samples, can be also viewed as the following graph theoretic problem (see, e.g., Ahuja, Magnanti and Orlin [3], and Haugland, Heber and Husøy [93]). As previously mentioned we consider the case of reconstructed values computed by linear interpolation and we use as objective function the RMSE associated with the selected samples.

We are given a directed graph $G = (V, A)$, where V is the set of vertices, each associated with a different sample $i = 1, \ldots, n$. The arc set A contains a directed arc (i, j) for each pair of vertices i and j whose corresponding samples may be consecutive in subset T. By definition of T, an arc $(i, j) \in A$ only if $i < j$, therefore graph G is acyclic.

With each arc $(i, j) \in A$ is associated a *cost* c_{ij}, defined as the sum of the square errors of the reconstructed values of all the samples from i to j when one selected sample is i and the

following one is j. In the 1D case, the arc costs are:

$$c_{ij} = \sum_{h=i}^{j} \left[\left(f(x_i) + \frac{f(x_j) - f(x_i)}{x_j - x_i}(x_h - x_i) \right) - f(x_h) \right]^2 . \tag{12.9}$$

In the 2D and 3D cases, the arc costs are defined in an analogous way by considering linear interpolation between the corresponding values of the samples, as described in Section 12.3. Note that the arc costs defined in this way turn out to be non-negative.

With this representation the actual type of DRP (i.e., with one- or multi-dimensional samples) is only considered in the definition of the cost matrix c. This allows for a common framework for considering 1D-, 2D- and 3D-DRP.

A path in G from vertex 1 to n is a sequence of samples, ordered by increasing coordinate values, and if the path includes $k-1$ arcs it corresponds to a feasible DRP solution. Hence, DRP calls for the determination of the min-cost path from vertex 1 to n with $k-1$ arcs.

For a general graph the problem of determining the min-cost *simple* path (i.e., such that no vertex is visited more than once) with prefixed cardinality is NP-hard in the strong sense, since it generalizes the Hamiltonian path problem. However, when the path is only required to be *elementary* (i.e., such that no arc is used more than once) the problem turns out to be polynomially solvable, see Saigal [142], and Beasley and Christofides [19]. When, as in our case, the graph is acyclic, every path is guaranteed to be elementary and simple, hence DRP is polynomially solvable as well.

An integer linear programming (ILP) formulation of DRP can be defined as follows. For each arc $(i,j) \in A$ we introduce a binary variable x_{ij}, which takes value 1 if the arc is used in the optimal solution, and 0 otherwise. The model then reads:

$$\text{(DRP)} \quad \min \sum_{(i,j)\in A} c_{ij} x_{ij} \tag{12.10}$$

subject to

$$\sum_{i:(i,j)\in A} x_{ij} - \sum_{i:(j,i)\in A} x_{ji} = 0 \quad \text{for each } j \in V, j \neq 1, n \tag{12.11}$$

$$\sum_{j:(1,j)\in A} x_{1j} = 1 \tag{12.12}$$

$$\sum_{i:(i,n)\in A} x_{in} = 1 \tag{12.13}$$

$$\sum_{(i,j)\in A} x_{ij} = k-1 \tag{12.14}$$

$$x_{ij} \in \{0,1\} \qquad \text{for each } (i,j) \in A \tag{12.15}$$

where constraints (12.11) impose that the same number of arcs enters and leaves a given vertex, whereas constraints (12.12) and (12.13) impose the degree requirements of the first and last vertices. Note that when, as in our case, graph G is acyclic each vertex has at most one entering and leaving arc. Finally, constraint (12.14) stipulates the resource requirements by imposing that exactly $k-1$ arcs are used in the optimal solution. Note that the objective function is the sum of the errors, but the RMSE may be easily obtained by using (12.6).

Closely related formulations have been introduced in the literature. Saigal [142] gave an ILP model for the *Cardinality-constrained Shortest Elementary Path* (CSEP) problem,

calling for the determination, on a general graph, of the shortest elementary path which uses a prefixed number of arcs. Beasley and Christofides [19] considered the *Resource Constrained Shortest Path* (RCSP) problem, which consists of finding, on a general graph, a shortest elementary path such that the total amount of (multi-dimensional) resource requirements associated with the traversed arcs and nodes must be within a prescribed range. Note that both CSEP and DRP are special cases of RCSP where: (i) only one resource is considered; (ii) the resource requirement associated with each arc is equal to one; (iii) the resource requirement associated with the nodes is zero; (iv) the total amount of used resource must be equal to $k-1$. The resulting transformation is valid for DRP due to the already mentioned acyclicity of the associated graph. Beasley and Christofides [19], proposed to relax the above model by dualizing in a Lagrangian fashion constraint (12.14). The resulting Lagrangian problem can be solved in polynomial time, since it can be viewed as a min-cost flow problem requiring to send a unit of flow from vertex 1 to vertex n.

The maximum sizes of the DRPs associated with 2D and 3D CT scan positioning are those arising in long bone reconstructions which, as mentioned in Section 12.3.1, involve between 400 and 500 samples. Analogous sizes are encountered in ECG data compression problems. This results in graphs with at most 500 nodes.

12.3.3 An exact algorithm for DRP

In this section we describe an exact algorithm for DRP, based on dynamic programming (see, Bellman and Dreyfus [20], and Dreyfus and Law [61]). We first review dynamic programming approaches proposed in the literature for related problems. These algorithms can be adapted to solve one- and multi-dimensional DRP when they are formulated as graph theoretic problems (see Section 12.3.2). We next propose an effective variant of the basic recursion used by these algorithms and discuss the efficient implementation of the resulting algorithm.

Dynamic programming for related problems

Saigal [142] proposed a dynamic programming approach to solve CSEP, i.e., the problem of finding the shortest elementary path with a prefixed number of arcs. Let $S(j,h)$ be the *state* of the dynamic program, denoting the cost of the shortest path from vertex 1 to vertex j involving exactly h arcs, and $\pi(j,h)$ the index of the vertex preceding j in such a path. The algorithm is based on the following standard recursion, in which each state of the current stage, h, is generated by considering the states of the previous stage, $h-1$.

$$\text{(12.16)} \quad \begin{cases} S(j,h) &= \min_{i=1,\dots,n}\{S(i,h-1)+c_{ij}\} \\ \pi(j,h) &= \arg\min_{i=1,\dots,n}\{S(i,h-1)+c_{ij}\} \end{cases} \qquad j=2,\dots,n; h=2,\dots,k$$

where the initial state is

$$\text{(12.17)} \quad \begin{cases} S(j,1) &= c_{1j} \\ \pi(j,1) &= 1 \end{cases} \qquad j=2,\dots,n.$$

The optimal solution value is $S(n,k)$ and can be computed in $O(kn^2)$ time.

Haugland, Heber and Husøy [93] adapted the above recursion to exactly solve 1D-DRP. The adaptation is required since in DRP the path must contain k vertices. Therefore, $S(j,h)$ now represents the cost of the shortest path from vertex 1 to vertex j, visiting exactly h vertices. In addition, the path must be simple, and its first and last vertices must be 1 and

n, respectively. Finally, due to the special structure of the underlying graph, the path is a sequence of vertices ordered by increasing numbers. As a consequence the number of states which must be considered is drastically reduced. For example, when defining the state $S(j,h)$ in (12.16) only states $S(i,h-1)$ with $i = h-1,\ldots,j-1$ need be considered, since $h-1$ vertices have been already visited and the last of these vertices can be, at most, vertex $j-1$. The above general recursion has been redefined in [93] in a natural *backward* fashion as follows:

procedure B-DP;
0. **for** $j := 2$ **to** n **do** $S(j,2) := c_{1j}$; $\pi(j,2) := 1$;
1. **for** $h := 3$ **to** k **do**
2. **for** $j := h$ **to** n **do**
3. $S(j,h) := S(h-1,h-1) + c_{h-1j}$; $\pi(j,h) := h-1$;
4. **for** $i := h$ **to** $j-1$ **do**
5. **if** $S(i,h-1) + c_{ij} < S(j,h)$ **then**
$S(j,h) := S(i,h-1) + c_{ij}$; $\pi(j,h) := i$

end.

Also in this case the optimal solution value is $S(n,k)$ and can be computed in $O(kn^2)$ time. However, as previously mentioned, the actual computational effort required by this recursion is considerably smaller. In fact, the number of states needed to define the new state $S(j,h)$ is, on average, half of that needed by (12.16). Note that, since the last vertex of the path must be n, in the last iteration of loop 1. (i.e., that with $h = k$), only state $S(k,n)$ should be defined. Hence, in this last iteration, Steps 2 to 5 can be performed just with $j = n$.

From a computational point of view, in the multi-dimensional DRP context, the actual bottleneck is the definition of the cost matrix c. In fact, by using (12.9), the overall computation of matrix c requires $O(skn^2)$ time, where s is the cardinality of each sample in the data set (i.e., $s = p$ in the 2D-DRP, and $s = p \cdot q$ in the 3D-DRP). On the other hand, the determination of the optimal solution, once c is given, takes $O(kn^2)$ time. Haugland, Heber and Husøy [93] proposed, for the 1D-DRP case, a method to compute matrix c in $O(n^2)$ time.

A new forward recursion

In the following we propose an alternative dynamic programming recursion which normally generates a smaller number of states than the above described backward one. In addition, this recursion does not necessarily require the overall computation of the cost matrix, thus considerably reducing the average computational effort for multi-dimensional DRP.

In our new *forward* recursion the non-dominated states of the last considered stage, say $h-1$, are used to generate the states of the current stage h. For example, the non-dominated states are those with value smaller than that of a known feasible solution. After the definition of each stage h ($h = 1,\ldots,k-1$), let us define the set I_h of the *active* vertices, i.e., those corresponding to the non-dominated states of the stage. The recursion is as follows:

procedure F-DP;
0. $I_1 := \{1\}$;
1. **for** $h := 2$ **to** $k-1$ **do**
2. $I_h := \emptyset$;
for $i := h$ **to** $n-k+h$ **do** $S(i,h) := UB$;

3. **for each** $i \in I_{h-1}$ **do**
4. **for** $j := i + 1$ **to** $n - k + h$ **do**
5. **if** $S(i, h-1) + c_{ij} < S(j, h)$ **then**
$S(j, h) := S(i, h-1) + c_{ij}$; $\pi(j, h) := i$;
6. $I_h := I_h \cup \{j\}$;
7. $S(n, k) := UB$;
8. **for each** $i \in I_{k-1}$ **do**
if $S(i, k-1) + c_{in} < S(n, k)$ **then**
$S(n, k) := S(i, k-1) + c_{in}$; $\pi(n, k) := i$
end.

where the *upper bound*, UB, is the value of a feasible solution provided by a heuristic algorithm.

It can be seen that the overall computational complexity of F-DP is the same as that of B-DP. However, if at each stage h the cardinality of the set I_h is kept small, then the average computational effort is considerably reduced. In particular, at Step 6, only the indices of the states $S(j, h)$ whose value is strictly smaller than UB are included in I_h. In fact, recalling that $c_{ij} \geq 0$ for all $(i, j) \in A$, a state with value not smaller than UB may not lead to a feasible solution improving UB. To this end it is crucial to obtain tight upper bound values as those produced by the heuristic algorithm described in the next section.

Set I_h can be further reduced by applying the following *dominance criterion*. A state $S(j, h)$ is *dominated* (hence $\{j\}$ is not included into I_h) if it exists a state $S(j, \ell)$ such that $S(j, h) \geq S(j, \ell)$ and $\ell < h$. In other words, the state is dominated if there exists a different path to vertex j using a smaller number of arcs and with not greater cost.

Our computational experiments (see Section 12.3.5) showed that the combination of these two simple strategies is considerably effective in the reduction of the dynamic program state space.

Another important consequence of the obtained state space reduction is that several entries of the cost matrix c are not used since the corresponding state transition is not performed. We exploit this fact by avoiding the off-line computation of matrix c and by computing its entries only if and when they are used during the dynamic program. Moreover, at each stage h the computation of a still undefined entry c_{ij} can be stopped (and c_{ij} is set to $+\infty$) as soon as it reaches an upper bound value u_h which guarantees that the value associated with the corresponding state transition $S(i, h-1) + c_{ij}$ is not smaller than UB. More formally, by denoting with $\mu_h := \min_{i \in I_{h-1}}\{S(i, h-1)\}$ the cost of the shortest partial path involving $h-1$ vertices, we define $u_h := UB - \mu_h$ as the upper bound on the cost of completion of any path with $h-1$ vertices (where $u_2 := UB$). For each $i \in I_{h-1}$, the state transition $S(i, h-1) + c_{ij}$, which possibly updates state $S(j, h)$, may require the computation of the still undefined entry c_{ij}. During such computation we set $c_{ij} = +\infty$ as soon as we obtain $c_{ij} \geq u_h$, since this implies $S(i, h-1) + c_{ij} \geq \mu_h + UB - \mu_h = UB$. Note that the value c_{ij} is the sum of $s \cdot (j - i - 1)$ interpolation errors, where $s = 1$, $s = p$ and $s = p \cdot q$ in the 1D, 2D and 3D case, respectively. Hence, early stops in the computation of the entries of c can be very effective in reducing the overall computational effort, particularly, in the multi-dimensional cases.

12.3.4 A DP-based heuristic algorithm

In this section we present a new heuristic algorithm for DRP, called H-DP, based on the forward dynamic programming approach of Section 12.3.3. This algorithm proved to be effective both in terms of solution quality and computing time required.

Algorithm H-DP uses the forward dynamic programming recursion but, at each stage h, heuristically purges the set of non-dominated states used to generate the states of the next stage, by removing the less "promising" ones. In particular, we adopted two different reduction criteria which are used at Step 5 of Procedure F-DP to further purge the number of states whose corresponding vertex is included into set I_h.

The first criterion removes the states which introduce a "too large" additional interpolation error, i.e., an error which is larger than $\alpha\, UBE/(k-1)$, where $UBE/(k-1)$ is the average error per selected vertex of the solution obtained with the trivial heuristic which selects a set of k regularly spaced vertices, and α is a real parameter. Our computational experience has shown that good results are obtained with α values between 2 and 5. In the computational testing of the next section we used $\alpha = 3$.

The second reduction rule removes the states which are associated with vertices "too far" from the current vertex i. (This is connected with the concept of planning/forecast horizons used in forward DP algorithms for dynamic lot size problems, see, e.g., Lundin and Morton [114] and Chand, Sethi and Proth [38].) In particular, we remove the states associated with vertices having index larger than $i+d_i$, where d_i is the maximum distance of the next selected vertex from the currently selected vertex i. Parameter d_i is determined by taking into account the degree of fluctuation of the values in the data set following coordinate x_i, associated with vertex i. In the preprocessing step, we compute for each vertex i the *fluctuation index* v_i. In the 1D case the index is defined as

$$v_i = \sum_{j=i}^{\min\{n-1,i+d\}} \frac{f_{x_{j+1}} - f_{x_j}}{x_{j+1} - x_j}.$$

In other words v_i is the sum of the slopes of the segments connecting subsequent samples in the data set not farther from i than a predefined parameter $d = n/(k-1)$. (In the 2D and 3D cases the fluctuation index is defined analogously, by considering the slopes of the segment between corresponding values in subsequent samples.) Then, the *maximum jump* parameter d_i of each vertex i is defined as

$$d_i = \max\{d, 5d(1 - \frac{v_i}{v_{\max}})\}$$

where $v_{\max} = \max_{i=1,\ldots,n-1}\{v_i\}$ is the maximum fluctuation index.

The use of the above criteria drastically reduces the computing time of algorithm H-DP with respect to the exact approach. However, since we have retained the most promising states the quality of the obtained solutions is generally very good and in many cases the optimal solution is found.

12.3.5 Computational results

The effectiveness of both exact and heuristic approaches (F-DP and H-DP, respectively) described in the previous sections has been extensively tested by considering both real-world and randomly generated instances and by comparing their results with those obtained by

previous approaches from the literature. The algorithms have been coded in ANSI-C language and the computational experiments have been performed by using a Silicon Graphics INDY R10000sc.

We first examine the real-world instances of scanning plan selection for the Computed Tomography analysis of long bones. As discussed in Section 12.3.1, this application is modeled as a 2D-DRP by using the scout image of the bone. We considered five different scout images of human femurs characterized by slightly different values of n and p (i.e., different sizes of the original data set to be reconstructed). The images are matrices of values ranging between 0 and 255, representing the gray level of each pixel. Instances F1, F2 and F3 are those described in Zannoni, Cappello and Viceconti [150], whereas F4 and F5 are new test problems. The number of samples to be selected is $k = 50$ which is considered, in clinical practice, a safe amount of radiation absorbed by the patient. The results obtained with these instances are presented in Table 12.5, where for each test problem we report the corresponding values of n and p. For the exact approach F-DP the table reports the optimal value, RMSE*, of the reconstruction error, expressed in gray level values, and the overall computing time, expressed in Silicon Graphics INDY R10000sc CPU seconds, which also includes the initial run of heuristic H-DP. As to our algorithm H-DP and heuristic DIEOM, described in [150], the table gives the relative deviation of the reconstruction error of the heuristic solution with respect to the optimal one and the corresponding computing time. The relative deviation of the solutions of a heuristic algorithm A, is computed as $(\text{RMSE}_A - \text{RMSE}^*)/\text{RMSE}^*$, where RMSE_A and RMSE* are the values of the heuristic and optimal solutions, respectively. Solutions marked with an asterisk are optimal. The code of algorithm DIEOM has been kindly provided us by the authors.

Table 12.5: Test problems from 2D real-world scanning plan optimization instances. Computing times in Silicon Graphics INDY R10000sc CPU seconds.

			F-DP		DIEOM		H-DP		
Problem	n	p	RMSE*	time	dev.	time	dev.		time
F1	419	410	1.574	81.25	0.134	40.47	0.000	*	8.01
F2	442	389	2.399	133.84	0.060	54.92	0.000	*	6.37
F3	453	320	0.650	65.28	0.184	38.15	0.045		6.15
F4	445	104	7.631	38.86	0.071	4.57	0.015		1.71
F5	409	107	12.438	21.27	0.077	3.40	0.000	*	1.69
		Average	4.938	68.10	0.105	28.30	0.012		4.79

The results of Table 12.5 point out that the computing time required by the exact approach is on average only 2.5 times greater than that required by DIEOM, and that it allows for an average improvement of the solution quality larger than 10 %. On the other hand, algorithm H-DP produces optimal or near-optimal solutions in times which are up to one order of magnitude smaller than those needed by DIEOM.

In addition to the real-world problems, we performed a set of experiments on 2D randomly generated instances. These instances were obtained by the superposition of $N \times P$ cosine waves ($N = P = 50$) with uniformly random amplitude and phase. Formally, each value

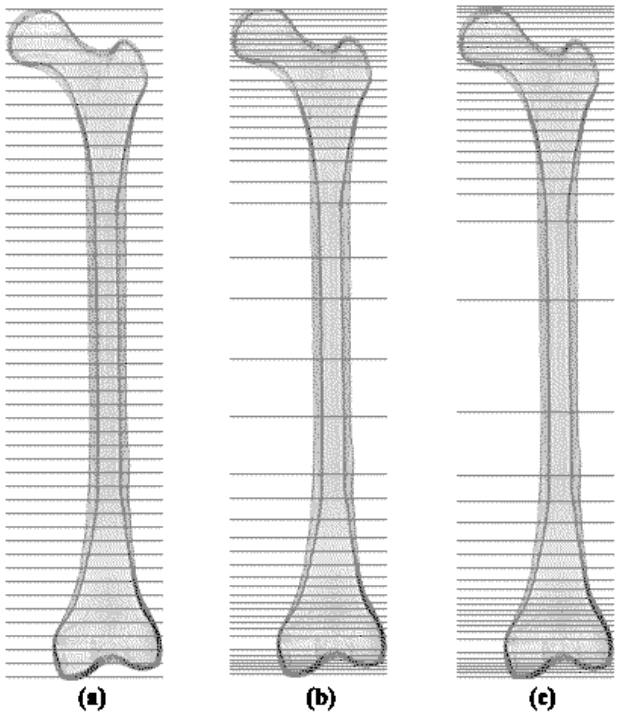


Figure 12.3: Different scanning plans computed for scout image of femur F4: (a) radiological (uniform) plan, (b) plan determined by heuristic H-DP, (c) optimal plan.

$f_j(x_i)$ $(i = 1, \ldots, n; j = 1, \ldots, p)$ is computed as

$$f_j(x_i) = \sum_{r=1}^{N} \sum_{s=1}^{P} A_{rs} \cos(ir\omega_x + js\omega_y - \varphi_{rs}) \tag{12.18}$$

where A_{rs} and φ_{rs} are the random components of amplitude and phase spectra, uniformly distributed in the intervals $[0, 1]$ and $[0, 2\pi]$, respectively, whereas $\omega_x = 2\pi/n$ and $\omega_y = 2\pi/p$.

A large set of 450 instances is considered, corresponding to $n = 100, 200, 300, 400$ and 500, $p = \frac{n}{2}, \frac{3}{4}n$ and n, and $k = \lfloor \frac{n}{15} \rfloor, \lfloor \frac{n}{10} \rfloor$ and $\lfloor \frac{n}{5} \rfloor$. The computational results are presented in Table 12.6, where each entry is the average value computed over ten generated instances. These instances appear easier than the real-world ones for the DIEOM heuristic approach. In fact, the deviation of the DIEOM solutions and the computing times are smaller than in the previous class. On the other hand, our heuristic H-DP confirms its effectiveness both in terms of solution quality and computing times: in a large majority of the instances heuristic H-DP obtains a smaller deviation and on average the computing times are about five times smaller than those of DIEOM. Moreover, for these instances the determination of the optimal solution turns out to be harder. The average computing times of exact approach F-DP are about one order of magnitude larger than the heuristic ones. However, even in the larger instances with $n = p = 500$ the computing times are within 11 minutes.

We finally considered the set of real-world ECG data compression instances used by Haugland, Heber and Husøy [93] to test their exact algorithm CCSP for 1D-DRP. This set is made

Table 12.6: Randomly generated 2D-DRP test problems. Computing times in Silicon Graphics INDY R10000sc CPU seconds.

			DIEOM		H-DP		F-DP
n	p	k	dev.	time	dev.	time	time
	50		0.059	0.02	0.026	0.11	0.50
100	75	6	0.049	0.04	0.027	0.10	0.70
	100		0.036	0.05	0.015	0.28	1.11
	50		0.059	0.04	0.039	0.05	0.44
100	75	10	0.037	0.08	0.037	0.06	0.65
	100		0.043	0.07	0.035	0.06	0.89
	50		0.044	0.08	0.040	0.03	0.39
100	75	20	0.038	0.13	0.031	0.03	0.59
	100		0.026	0.21	0.033	0.04	0.83
	100		0.034	0.21	0.027	0.44	7.16
200	150	13	0.036	0.32	0.030	0.64	10.93
	200		0.041	0.50	0.026	0.81	14.48
	100		0.035	0.37	0.034	0.21	6.82
200	150	20	0.035	0.65	0.038	0.30	10.41
	200		0.034	0.87	0.038	0.33	13.76
	100		0.033	1.36	0.038	0.09	5.92
200	150	40	0.034	2.07	0.028	0.15	8.99
	200		0.033	2.73	0.031	0.17	11.89
	150		0.029	1.15	0.038	0.71	34.92
300	225	20	0.032	1.43	0.031	1.25	52.43
	300		0.037	2.21	0.032	1.79	71.22
	150		0.031	1.37	0.030	0.59	33.43
300	225	30	0.029	2.30	0.025	1.13	50.40
	300		0.030	3.93	0.019	1.72	68.57
	150		0.022	0.47	0.013	0.25	23.24
300	225	60	0.029	0.74	0.016	0.33	35.17
	300		0.026	0.90	0.014	0.48	46.35
	200		0.038	2.40	0.038	1.40	107.99
400	300	26	0.041	4.49	0.041	2.25	164.66
	400		0.030	11.93	0.038	3.23	231.69
	200		0.032	6.58	0.022	1.29	98.13
400	300	40	0.033	10.22	0.026	1.80	148.66
	400		0.030	28.40	0.023	2.68	204.10
	200		0.010	1.08	0.000	0.38	43.48
400	300	60	0.010	1.94	0.000	0.54	65.62
	400		0.011	5.98	0.000	0.80	83.74
	250		0.031	7.20	0.029	2.91	261.98
500	375	33	0.026	27.80	0.033	4.11	446.80
	500		0.030	36.85	0.035	6.35	658.21
	250		0.062	6.86	0.018	1.76	220.16
500	375	50	0.060	36.20	0.019	3.03	349.20
	500		0.064	43.20	0.022	4.29	504.25
	250		0.019	2.41	0.000	0.64	57.70
500	375	100	0.020	10.65	0.000	0.93	85.55
	500		0.022	16.52	0.000	1.18	111.05

up by five ECG instances, with $n = 500$ samples each, from the MIT database [122]. As in [93], for each instance we considered six different values of $k = \lfloor \frac{n}{2} \rfloor, \lfloor \frac{n}{5} \rfloor, \lfloor \frac{n}{8} \rfloor, \lfloor \frac{n}{12} \rfloor, \lfloor \frac{n}{16} \rfloor$ and $\lfloor \frac{n}{20} \rfloor$. Table 12.7 compares our heuristic and exact algorithms with the exact algorithm CCSP. The code of CCSP has been kindly provided us by the authors. For each ECG instance and for each value of k, the table reports the optimal RMSE value and the computing times of CCSP and F-DP. The table also includes the relative deviation of the solution of the well-known heuristic FAN (see, Barr [15]), derived from Table 1 of [93], as well as the relative deviation and the computing time of heuristic H-DP. Also with these real-world instances the proposed exact and heuristic approaches have a very good performance. The new exact F-DP is considerably faster than the backward CCSP algorithm, requiring on average about half of the time required by CCSP, and always obtaining the optimal solutions in less than three seconds. Heuristic H-DP has an average deviation of 0.006, i.e., more than 50 times smaller than that of classical FAN method, and finds the optimal solution in half of the cases. The computing times of heuristic H-DP are generally smaller than half a second, thus being suitable for on-line ECG signal compression.

12.3.6 Conclusions

In this section we have presented innovative and effective exact and heuristic approaches for the solution of Data set Reconstruction Problems which find several practical applications, particularly in the biomedical field. The research was initially motivated by the optimization of the Computed Tomography (CT) scanning plan in the reconstruction of long bones, but the algorithms presented have a wide range of other practical applications, e.g., in the compression of Electrocardiography (ECG) data. The resulting problem can be modeled as that of determining a shortest path with prefixed cardinality on a suitably defined graph. Although, the problem is known to be polynomially solvable by Dynamic Programming, the computing times required by known exact approaches from the literature are generally not compatible with real-world applications.

The exact approach proposed in this section is based on an efficient implementation of a Dynamic Programming algorithm, and optimally solves real-world instances of femur data reconstruction in computing times which are comparable with those of heuristic algorithms from the literature. In addition, for the real-world applications we considered, the optimal solutions turn out considerably better than the heuristic ones: in the CT case the improvement is about 10 %, whereas in the ECG case is 30 %.

In order to obtain high quality solutions within much smaller computing times we have proposed an effective heuristic algorithm. The experimental testing has shown that the new heuristic outperforms previously proposed methods being able to determine near-optimal solutions in a few seconds.

The present work was carried out within the European Commission funded project named PROMISE (Pre and Post Processing of Medical Images on High performance Architectures). The results of PROMISE project, including the algorithms described in this section, are available through WWW (http://www.cineca.it/visit/promise/index.html) as decision support tools for radiology units.

Table 12.7: Test problems from 1D-DRP real-world ECG signal compression instances. Computing times in Silicon Graphics INDY R10000sc CPU seconds.

		Exact			FAN	H-DP		
Problem	k	RMSE*	CCSP	F-DP	dev.	dev.		time
	250	0.504	9.50	2.23	0.148	0.004		0.47
	100	1.776	5.12	2.66	0.296	0.015		0.36
	62	2.289	3.35	1.99	0.252	0.004		0.32
202_0	41	2.904	2.26	2.53	0.247	0.000	*	0.31
	31	3.856	1.66	1.39	0.315	0.000	*	0.34
	25	6.100	1.34	1.46	0.423	0.000	*	0.40
	250	0.597	9.12	2.33	0.257	0.000	*	0.48
	100	1.808	4.86	2.70	0.277	0.002		0.37
	62	2.332	3.21	2.10	0.328	0.002		0.36
203_0	41	2.736	2.19	2.71	0.386	0.000	*	0.39
	31	3.415	1.64	1.53	0.461	0.000	*	0.38
	25	4.553	1.33	1.56	0.478	0.000	*	0.44
	250	0.618	8.83	2.40	0.159	0.000	*	0.49
	100	1.806	4.80	2.70	0.332	0.002		0.36
	62	2.385	3.18	2.04	0.284	0.000	*	0.34
203_1	41	3.018	2.17	2.66	0.369	0.000	*	0.35
	31	3.692	1.63	1.47	0.410	0.027		0.33
	25	4.438	1.33	1.44	0.477	0.000	*	0.38
	250	0.465	10.12	2.29	0.225	0.000	*	0.48
	100	1.443	5.36	2.69	0.350	0.006		0.37
	62	1.882	3.49	2.11	0.300	0.003		0.35
207_0	41	2.299	2.35	2.76	0.299	0.000	*	0.35
	31	2.571	1.75	1.49	0.317	0.000	*	0.32
	25	2.900	1.41	1.43	0.301	0.001		0.34
	250	0.580	9.49	2.42	0.171	0.000	*	0.51
	100	1.910	5.17	2.54	0.291	0.001		0.36
	62	2.524	3.36	1.94	0.310	0.000		0.34
214_0	41	3.149	2.27	2.47	0.333	0.014		0.32
	31	3.769	1.69	1.31	0.411	0.029		0.28
	25	4.618	1.37	1.25	0.374	0.060		0.29
Average		2.565	3.85	2.09	0.319	0.006		0.37

Bibliography

[1] E. Aarts, J. H. M. Korst, and P. J. M. van Laarhoven. Simulated annealing. In E. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 91–120. J. Wiley & Sons, Chichester, 1997.

[2] E. Aarts and J. K. Lenstra (eds.). *Local Search in Combinatorial Optimization*. John Wiley & Sons, Chichester, 1997.

[3] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows*. Prentice Hall, Englewood Cliffs, 1993.

[4] B. Alidaee, G. Kochenberger, and A. Ahmadian. Programming approach for the optimal solution of two scheduling problems. *International Journal of System Science*, 25:401–408, 1994.

[5] N. Ascheuer, M. Fischetti, and M. Grötschel. Solving ATSP with time windows by branch-and-cut. Technical report, ZIB Berlin, 1999.

[6] B. S. Baker, D. J. Brown, and H. P. Katseff. A 5/4 algorithm for two-dimensional packing. *Journal of Algorithms*, 2:348–368, 1981.

[7] B. S. Baker, E. G. Coffman, Jr., and R. L. Rivest. Orthogonal packing in two dimensions. *SIAM Journal on Computing*, 9:846–855, 1980.

[8] D. S. Baker and J. S. Schwarz. Shelf algorithms for two-dimensional packing problems. *SIAM Journal on Computing*, 12:508–525, 1983.

[9] E. Balas and E. J. Saltzman. Facets of the three-index assignment polytope. *Discrete Applied Mathematics*, 23:201–229, 1989.

[10] E. Balas and E. J. Saltzman. An algorithm for the three-index assignment problem. *Operations Research*, 39:150–161, 1991.

[11] P. Baptiste, C. Le Pape, and W. Nuijten. Efficient operations research algorithms in constraint-based scheduling. In *Proceedings of IJCAI'95*, 1995.

[12] F. Barahona. A solvable case of quadratic 0-1 programming. *Discrete Applied Mathematics*, 13:23–26, 1986.

[13] F. Barahona, M. Jünger, and G. Reinelt. Experiments in quadratic 0-1 programming. *Mathematical Programming*, 36:157–173, 1989.

[14] F. Barahona and A. R. Mahjoub. On the cut polytope. *Mathematical Programming*, 36:157–173, 1986.

[15] R. C. Barr. Adaptive sampling of cardiac waveforms. *J. Electrocard.*, 19:379–394, 1989.

[16] J. J. Bartholdi III, J. H. Vande Vate, and J. Zhang. Expected performance of the shelf heuristic for the 2-dimensional packing. *Operations Research Letters*, 8:11–16, 1989.

[17] J. E. Beasley. Algorithms for unconstrained two-dimensional guillotine cutting. *Journal of the Operational Research Society*, 36:297–306, 1985.

[18] J. E. Beasley. An exact two-dimensional non-guillotine cutting tree search procedure. *Operations Research*, 33:49–64, 1985.

[19] J.E. Beasley and N. Christofides. An algorithm for the resource constrained shortest path problem. *Networks*, 19:379–394, 1989.

[20] R. E. Bellman and S. E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, 1971.

[21] B. E. Bengtsson. Packing rectangular pieces – a heuristic approach. *The Computer Journal*, 25:353–357, 1982.

[22] H. Beringer and B. De Backer. Combinatorial problem solving in constraint logic programming with cooperating solvers. In C. Beierle and L. Plumer, editors, *Logic Programming: formal Methods and Practical Applications*. North Holland, 1995.

[23] J. O. Berkey and P. Y. Wang. Two dimensional finite bin packing algorithms. *Journal of the Operational Research Society*, 38:423–429, 1987.

[24] D. P. Bertsekas and P. Tseng. Relaxation methods for minimum cost ordinary and generalized network flow problems. *Operations Research*, 36:93–114, 1988.

[25] L. Bianco, A. Mingozzi, and S. Ricciardelli. A set partitioning approach to the multiple depot vehicle scheduling problem. *Optimization Methods and Software*, 3:163–194, 1994.

[26] A. Bockmayr and T. Kasper. Branch-and-Infer: A unifying framework for Integer and Finite Domain Constraint Programming. *INFORMS Journal on Computing*, 10:287–300, 1998.

[27] D. J. Brown. An improved BL lower bound. *Information Processing Letters*, 11:37–39, 1980.

[28] R. E. Burkard and R. Rudolf. Computational investigation on 3-dimensional axial assignment problems. *Belgian J. Oper. Res. Statist. Comput. Sci.*, 32:85–98, 1993.

[29] R. E. Burkard, R. Rudolf, and G. Woeginger. Three-dimensional axial assignment problems with decomposable coefficients. *Discrete Applied Mathematics*, 65:123–139, 1996.

[30] P. Camerini, L. Fratta, and F. Maffioli. On improving relaxation methods by modified gradient techniques. *Math. Prog. Study*, 3:26–34, 1975.

[31] A. Caprara and M. Fischetti. Branch-and-cut algorithms. In M. Dell'Amico, F. Maffioli, and S. Martello, editors, *Annotated Bibliographies in Combinatorial Optimization*, pages 45–63. John Wiley & Sons, Chichester, 1997.

[32] A. Caprara and P. Toth. Lower bounds and algorithms for the 2-dimensional vector packing problem. Technical report OR/97/3, DEIS - University of Bologna, 1997.

[33] G. Carpaneto, M. Dell'Amico, M. Fischetti, and P. Toth. A branch and bound algorithm for the multiple depot vehicle scheduling problem. *Networks*, 19:531–548, 1989.

[34] G. Carpaneto, S. Martello, and P. Toth. Algorithms and codes for the assignment problem. *Annals of Operations Research*, 13:193–223, 1988.

[35] Y. Caseau and F. Laburthe. Improving Branch and Bound for Jobshop Scheduling with Constraint Propagation. In M. Deza, R. Euler, and Y. Manoussakis, editors, *Combinatorics and Computer Science, LNCS 1120*, pages 129–149. Springer Verlag, 1995.

[36] Y. Caseau and F. Laburthe. Solving small TSPs with constraints. In *Proceedings of the Fourteenth International Conference on Logic Programming - ICLP'97*, pages 316–330, 1997.

[37] Y. Caseau and F. Laburthe. Solving various weighted matching problems with constraints. In *Proceedings of Constraint Programming - CP'97*, 1997.

[38] S. Chand, S. P. Sethi, and J.-M. Proth. Existence of forecast horizons in undiscounted discrete-time lot size models. *Operations Research*, 38:884–892, 1990.

[39] P. Chardaire and A. Sutter. A decomposition method for quadratic zero-one programming. *Management Science*, 41:704–712, 1995.

[40] B. Chazelle. The bottom-left bin packing heuristic: An efficient implementation. *IEEE Transactions on Computers*, 32:697–707, 1983.

[41] N. Christofides and C. Whitlock. An algorithm for two-dimensional cutting problems. *Operations Research*, 25:30–44, 1977.

[42] 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.

[43] E. G. Coffman, Jr., M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9:801–826, 1980.

[44] E. G. Coffman, Jr. and P. W. Shor. Average-case analysis of cutting and packing in two dimensions. *European Journal of Operational Research*, 44:134–144, 1990.

[45] T. H. Cormen, C. E. Leiserson, and R. R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[46] C. De Simone. The cut polytope and the boolean quadratic polytope. *Discrete Mathematics*, 79:71–75, 1990.

[47] M. Dell'Amico. On the continuous relaxation of packing problems. Technical Report 182, Dipartimento di Economia Politica, Università di Modena, 1999.

[48] M. Dell'Amico, M. Fischetti, and P. Toth. Heuristic algorithms for the multiple depot vehicle scheduling problem. *Management Science*, 39:115–125, 1993.

[49] M. Dell'Amico, M. Labbé, and F. Maffioli. Spanning tree problems with leaf-dependent objective function. *Networks*, 27:175–181, 1996.

[50] M. Dell'Amico, A. Lodi, and F. Maffioli. Solution of the cumulative assignment problem with a well-structured tabu search method. *Journal of Heuristics*, 5:123–143, 1999.

[51] M. Dell'Amico, A. Lodi, and S. Martello. Efficient algorithms and codes for k-cardinality assignment problems. *Discrete Applied Mathematics*, 1999 (to appear).

[52] M. Dell'Amico and F. Maffioli. A new tabu search approach to the 0–1 equicut problem. In I.H. Osman and P. Kelly, editors, *Meta-Heuristics: theory and applications*, pages 361–377. Kluwer Academic Publishers, 1996.

[53] M. Dell'Amico, F. Maffioli, and M. Trubian. New bounds for optimum traffic assignment in satellite communication. *Computers & Opererations Research*, 25:729–743, 1998.

[54] M. Dell'Amico and S. Martello. The k-cardinality assignment problem. *Discrete Applied Mathematics*, 76:103–121, 1997.

[55] M. Dell'Amico and S. Martello. Linear assignment. In M. Dell'Amico, F. Maffioli, and S. Martello, editors, *Annotated Bibliographies in Combinatorial Optimization*, pages 355–371. Wiley, Chichester, 1997.

[56] M. Dell'Amico, S. Martello, and D. Vigo. An exact algorithm for non-oriented two-dimensional bin packing problems. In preparation, 1998.

[57] M. Dell'Amico and S.Martello. Optimal scheduling of tasks on identical parallel processors. *ORSA Journal on Computing*, 7:191–200, 1995.

[58] M. Dell'Amico and M. Trubian. Solution of large weighted equicut problems. *European Journal of Operational Research*, 106:500–521, 1997.

[59] E. A. Dinic. Algorithms for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady*, 11:1277–1280, 1970.

[60] K. A. Dowsland and W. B. Dowsland. Packing problems. *European Journal of Operational Research*, 56(1):2–14, 1992.

[61] S.E. Dreyfus and A. M. Law. *The Art and The Theory of Dynamic Programming.* Academic Press, New York, 1977.

[62] H. Dyckhoff. A typology of cutting and packing problems. *European Journal of Operational Research*, 44:145–159, 1990.

[63] H. Dyckhoff and U. Finke. *Cutting and Packing in Production and Distribution.* Physica Verlag, Heidelberg, 1992.

[64] H. Dyckhoff, G. Scheithauer, and J. Terno. Cutting and Packing (C&P). In M. Dell'Amico, F. Maffioli, and S. Martello, editors, *Annotated Bibliographies in Combinatorial Optimization*. John Wiley & Sons, Chichester, 1997.

[65] 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.

[66] O. Færø, D. Pisinger, and M. Zachariasen. Guided local search for the three-dimensional bin packing problem. Technical report, DIKU - University of Copenhagen, 1999.

[67] G. Fischetti, M. Laporte and S. Martello. The delivery man problem and cumulative matroids. *Operations Research*, 41:1055–1064, 1993.

[68] M. Fischetti, A. Lodi, and P. Toth. A branch-and-cut algorithm for the multiple depot vehicle scheduling problem. Technical Report OR/99/1, DEIS - Università di Bologna, 1999.

[69] M. Fischetti and P. Toth. An additive bounding procedure for the asymmetric traveling salesman problem. *Mathematical Programming*, 53:173–197, 1992.

[70] M. Fischetti and P. Toth. A polyhedral approach to the asymmetric traveling salesman problem. *Management Science*, 43:1520–1536, 1997.

[71] F. Focacci, A. Lodi, and M. Milano. Cost-based domain filtering. In J. Jaffar, editor, *Principle and Practice of Constraint Programming - CP'99, LNCS 1713*, pages 189–203. Springer-Verlag, Berlin Heidelberg, 1999.

[72] F. Focacci, A. Lodi, and M. Milano. Integration of CP and OR methods for Matching Problems. In *Proceedings of the Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems - CP-AI-OR'99*, 1999.

[73] F. Focacci, A. Lodi, and M. Milano. Solving TSP with Time Windows with constraints. In D. De Schreye, editor, *Logic Programming - Proceedings of the 1999 International Conference on Logic Programming*, pages 515–529. The MIT-press, Cambridge, Massachusetts, 1999.

[74] F. Focacci, A. Lodi, M. Milano, and D. Vigo. Solving TSP through the integration of OR and CP techniques. *Electronic Notes in Discrete Mathematics*, 1, 1999. http://www.elsevier.com/cas/tree/store/disc/free/endm/.

[75] M. A. Forbes, J. N. Holt, and A. M. Watts. An exact algorithm for the multiple depot bus scheduling problem. *European Journal of Operational Research*, 72:115–124, 1994.

[76] J. B. Frenk and G. G. Galambos. Hybrid next-fit algorithm for the two-dimensional rectangle bin-packing problem. *Computing*, 39:201–217, 1987.

[77] G. Gallo, P. L. Hammer, and B. Simeone. Quadratic knapsack problems. *Mathematical Programming*, 12:132–149, 1980.

[78] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W.H. Freeman, San Francisco, 1979.

[79] F. Glover. A template for scatter search and path relinking. In J. K. Hao, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers, editors, *Lecture Notes in Computer Science*, volume 1363, pages 1–45. 1997.

[80] F. Glover, G. A. Kochenberger, and B. Alidaee. Adaptive memory tabu search for binary quadratic programs. *Management Science*, 44:336–345, 1998.

[81] F. Glover, G. A. Kochenberger, B. Alidaee, and M. Amini. Tabu search with critical event memory: an enhanced application for binary quadratic programs. In S. Voss, S. Martello, I.H. Osman, and C. Roucairol, editors, *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, pages 93–110. Kluwer Academic Publishers, Boston, 1998.

[82] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Boston, 1997.

[83] I. Golan. Performance bounds for orthogonal oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 10:571–582, 1981.

[84] A. V. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *Journal of Algorithms*, 22:1–29, 1997.

[85] D. Goldfarb and M. D. Grigoriadis. A computational comparison of the dinic and network simplex methods for maximum flow. In B. Simeone, P. Toth, G. Gallo, F. Maffioli, and S. Pallottino, editors, *FORTRAN Codes for Network Optimization*, pages 83–103. Baltzer, Basel, 1988.

[86] M. Grötschel, L. Lovász, and Schrijver A. *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag, Berlin Heidelberg, 1988.

[87] O. Haas. The cumulative assignment problem - local search and heuristics. Master thesis, Department of Mathematics, University of Kaiserslautern, Germany, 1995.

[88] E. Hadjiconstantinou and N. Christofides. An exact algorithm for general, orthogonal, two-dimensional knapsack problems. *European Journal of Operational Research*, 83:39–56, 1995.

[89] E. Hadjiconstantinou and N. Christofides. An exact algorithm for the orthogonal, 2-d cutting problems using guillotine cuts. *European Journal of Operational Research*, 83:21–38, 1995.

[90] P. L. Hammer. Some network flow problems solved with pseudo-boolean programming. *Operations Research*, 13:388–399, 1965.

[91] P. L. Hammer, P. Hansen, and B. Simeone. Roof duality, complementation and persistence in quadratic 0-1 programming. *Mathematical Programming*, 28:121–155, 1984.

[92] P. L. Hammer and V. R. Mahadev. Bithreshold graphs. *SIAM J. Appl. Math.*, 6:497–506, 1985.

[93] D. Haugland, J. G. Heber, and J. H. Husøy. Optimisation algorithms for ECG data compression. *Medical & Biological Engineering & Computing*, 35:420–424, 1997.

[94] M. Hifi. Exact algorithms for large-scale unconstrained two and three staged cutting problems. In *Contribution à la Résolution de Quelques Problèmes Difficiles de l'Optimisation Combinatoire*. Thèse d'Habilitation à Diriger des Recherches en Informatique, Université de Versailles–Saint Quentin en Yvelines, 1998.

[95] S. Høyland. Bin-packing in 1.5 dimension. In R.G. Karlsson and A. Lingas, editors, *Lecture Notes in Computer Science*, pages 129–137. Springer-Verlag, Berlin, 1988.

[96] R. Jain, J. Werth, and J. C. Browne. A note on scheduling problems arising in satellite communication. *JORS*, 48:100–102, 1997.

[97] D. S. Johnson. *Near-optimal bin packing algorithms*. PhD thesis, MIT, Cambridge, MA, 1973.

[98] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3:299–325, 1974.

[99] R. Jonker and T. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38:325–340, 1987.

[100] M. Jünger, G. Reinelt, and G. Rinaldi. The travelling salesman problem. In M. Dell'Amico, F. Maffioli, and S. Martello, editors, *Annotated Bibliographies in Combinatorial Optimization*, pages 199–221. Wiley, 1997.

[101] M. Laguna and F. Glover. Tabu search. In C. R. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems*, pages 70–141. Blackwell Scientific Publications, 1993.

[102] 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 Solving from Nature*, pages 549–603, Berlin, 1996.

[103] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York, 1976.

[104] K. Li and K.-H. Cheng. On three-dimensional packing. *SIAM Journal on Computing*, 19:847–867, 1990.

[105] K. Li and K.-H. Cheng. Heuristic algorithms for on-line packing in three dimensions. *Journal of Algorithms*, 13:589–605, 1992.

[106] A. Lodi, K. Allemand, and T. M. Liebling. An evolutionary heuristic for quadratic 0–1 programming. *European Journal of Operational Research*, 119:662–670, 1999.

[107] 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, editors, *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, pages 125–139. Kluwer Academic Publishers, Boston, 1998.

[108] 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, 1999.

[109] A. Lodi, S. Martello, and D. Vigo. Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. *INFORMS Journal on Computing*, 11:345–357, 1999.

[110] A. Lodi, S. Martello, and D. Vigo. Recent advances on two-dimensional bin packing problems. Technical Report OR/99/2, DEIS - Università di Bologna, 1999.

[111] A. Lodi, D. Vigo, and C. Zannoni. Exact and heuristic algorithms for data sets reconstruction. *European Journal of Operational Research*, 124:139–150, 2000.

[112] A. Löbel. *Optimal Vehicle Schedule in Public Transport.* PhD thesis, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Germany, 1998.

[113] A. Löbel. Vehicle scheduling in public transit and lagrangean pricing. *Management Science*, 44:1637–1649, 1998.

[114] R. A. Lundin and T. E. Morton. Planning horizons for the dynamic lot size model: Zabel vs. protective procedures anc computational results. *Operations Research*, 23:711–734, 1975.

[115] S. Martello, D. Pisinger, and D. Vigo. An algorithm for the three-dimensional bin packing problem. Technical Report OR/98/10, DEIS - Università di Bologna, 1999.

[116] S. Martello, D. Pisinger, and D. Vigo. The three-dimensional bin packing problem. *Operations Research*, 1999 (to appear).

[117] S. Martello and P. Toth. Linear assignment problems. In S. et al. Martello, editor, *Surveys in Combinatorial Optimization*, pages 259–282. North-Holland, Amsterdam, 1987.

[118] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley & Sons, Chichester, 1990.

[119] S. Martello and D. Vigo. Exact solution of the two-dimensional finite bin packing problem. *Management Science*, 44:388–399, 1998.

[120] R.D. McBride and J.S. Yormark. An implicit enumeration algorithm for quadratic integer programming. *Management Science*, 26:282–296, 1980.

[121] F. K. Miyazawa and Y. Wakabayashi. An algorithm for the three-dimensional packing problem with asymptotic performance analysis. *Algorithmica*, 18:122–144, 1997.

[122] G. Moody. *MIT-BIH arrhythmia database CD-ROM.* MIT, 1992. 2^{nd} edn.

[123] H. Mühlenbein. Genetic algorithms. In E. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 137–191. John Wiley & Sons, Chichester, 1997.

[124] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley and Sons, New York, 1988.

[125] M. Padberg. The boolean quadratic polytope: Characteristics, facets and relatives. *Mathematical Programming*, 45:139–172, 1989.

[126] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of a large-scale symmetric traveling salesman problems. *SIAM Review*, 33:60–100, 1991.

[127] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, Englewood Cliffs, N.J., 1988.

[128] P. M. Pardalos and G. P. Rodgers. Computational aspects of a branch and bound algorithm for quadratic zero-one programming. *Computing*, 45:131–144, 1990.

[129] G. Pesant, M. Gendreau, J. Y. Potvin, and J. M. Rousseau. An exact constraint logic programming algorithm for the travelling salesman problem with time windows. *Transportation Science*, 32:12–29, 1998.

[130] G. Pesant, M. Gendreau, J. Y. Potvin, and J. M. Rousseau. On the flexibility of Constraint Programming models: From Single to Multiple Time Windows for the Travelling Salesman Problem. *European Journal of Operational Research*, 117:253–263, 1999.

[131] J. C. Picard and H. D. Ratliff. Minimum cuts and related problems. *Networks*, 5:357–370, 1974.

[132] W. P. Pierskalla. The multidimensional assignment problem. *Operations Research*, 16:422–431, 1968.

[133] C. A. Pomalaza-Ráez. A note on efficient SS/TDMA assignment algorithms. *IEEE Trans. on Communications*, 36:1078–1082, 1988.

[134] C. Prins. An overview of scheduling problems arising in satellite communications. *JORS*, 45:611–623, 1994.

[135] J.F. Puget. A C++ implementation of CLP. Technical Report 94-01, ILOG Headquarters, 1994.

[136] L. Qi, E. Balas, and G. Gwan. A new facet class and a polyhedral method for the three-index assignment problem. In D. Z. Du, editor, *Advances in Optimization and Approximation*, pages 256–274. Kluwer Academic Publishers, 1994.

[137] J.C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of AAAI'94*, 1994.

[138] G. Reinelt. TSPLIB - a Travelling Salesman Problem Library. *ORSA Journal on Computing*, 3:376–384, 1991.

[139] G. Reinelt. Oral communication, 1999.

[140] C. Ribeiro and F. Soumis. A column generation approach to the multiple depot vehicle scheduling problem. *Operations Research*, 42:41–52, 1994.

[141] R. Rodosek, M. Wallace, and M. T. Hajian. A new approach to integrating mixed integer programming and constraint logic programming. *Annals of Operational Research*, 1997.

[142] R. Saigal. A constrained shortest route problem. *Operations Research*, 16:205–209, 1968.

[143] W. Schneider. Trim-loss minimization in a crepe-rubber mill; optimal solution versus heuristic in the 2 (3)-dimensional case. *European Journal of Operational Research*, 34:273–281, 1988.

[144] H. Simonis. Calculating lower bounds on a resource scheduling problem. Technical report, Cosytec, 1995.

[145] D. Sleator. A 2.5 times optimal algorithm for packing in two dimensions. *Information Processing Letters*, 10:37–40, 1980.

[146] A. Steinberg. A strip-packing algorithm with absolute performance bound 2. *SIAM Journal on Computing*, 26:401–409, 1997.

[147] E. Taillard. Heuristic methods for large centroid clustering problems. Technical Report IDSIA-96-96, IDSIA, Lugano, Switzerland, 1996.

[148] F. J. Vasko, F. E. Wolf, and K. L. Stott. A practical solution to a fuzzy two-dimensional cutting stock problem. *Fuzzy Sets and Systems*, 29:259–275, 1989.

[149] C. Voudouris and E. Tsang. Guided local search and its application to the traveling salesman problem. *European Journal of Operational Research*, 113:469–499, 1999.

[150] C. Zannoni, A. Cappello, and M. Viceconti. Optimal CT scanning plan for long-bone 3-D reconstruction. *IEEE Transactions on Medical Imaging*, 17:663–666, 1998.