Alma Mater Studiorum Università di Bologna

Dottorato di Ricerca in

Automatica e Ricerca Operativa

MAT/09

XXII Ciclo

Algorithms and Models For Combinatorial Optimization Problems

Albert Einstein Fernandes Muritiba

Il Coordinatore
Prof. Claudio Melchiorri

Il Relator
Prof. Paolo Toth

Esame finale 2010

Abstract

In this thesis we present some combinatorial optimization problems, suggest models and algorithms for their effective solution. For each problem, we give its description, followed by a short literature review, provide methods to solve it and, finally, present computational results and comparisons with previous works to show the effectiviness of the proposed approaches. The considered problems are: the *Generalized Traveling Salesman Problem* (GTSP), the *Bin Packing Problem with Conflicts*(BPPC) and the *Fair Layout Problem* (FLOP).

Dedicatória/Dedicatory:

Dedico esta tese aos mes pais, Fernando e Eliane, pelo apoio e ensinamentos que me fizeram dar os passos que eles não tiveram a oportunidade de dar. Aos meus irmãos, Lavoisier e Gladson que sempre me ajudaram e a minha esposa, Cristiane, que sempre esteve ao meu lados nesses longos anos de doutorado. Também dedico esta tese ao querido professor Negreiros, que sempre acreditou na minha capacidade e me propossionou esta oportunidade de estudar na universidade mais antigas do mundo e referência em nossa área.

I dedicate this thesis also to my friends who I've met in Bologna, Manuel Iori, Enrico Malaguti, Valentina Cacchiani, Federico Mascagne, Rosa Medina, Victor Veras Valdes, André Gustavo dos Santos, Fabio Furini and Alfredo Persiani for make me feel at home in Italy. Moreover, I dedicate this thesis to the professor Paolo Toth, who has helped me making much more than his duty as adviser.

Introduction.

Combinatorial Optimization Problems are very common in industrial processes and planning activities. They are problems where a solution is composed by a set of fundamental discrete decisions or assumptions. Every decision may influence the global cost and the feseability of the solution. The trivial way to solve a combinatorial optimization problem is to enumerate the elements of the correspondent feasible solutions set and pick up the best one. But, due to the combinatorial nature of the considered problems, in real cases the number of solutions (feasible or unfeasible) to be enumerated for a given problem is intractable even for very powerful computers.

To deal with the difficulty in solving Combinatorial Optimization Problems, some techniques have been proposed:

- *Branch and Bound* methods where the solution space is systematically divided and its subsets of solutions are evaluated according to their bounds on the objective function value;

- *Heuristic* methods, where the problem is solved through the application of experience-based techniques. When these techniques dirive from other generic or natural problems rather than the original problem, we call them *Meta-heuristic*;

- Methods based on *Integer/Mixer Programming* and *Hybrid* methods that combine some of previously mentioned approaches.

In this thesis, our goal was to obtain good results on different combinatorial optimization problems by choosing and appling one or more approaches for each case.

The first problem studied in this work is the *Fair Layout Optimization Problem* (FLOP). In this problem we have to place rectangular stands over a given non convex area. The stands must satisfy some fixed patterns. The papers [11] is our publication related to this problem.

The second problem is the *Generalized Traveling Salesman Problem* (GTSP), which is a generalization of the classical *Traveling Salesman Problem* (TSP) where the vertices are divided in clusters and the salesman must visit at least one vertex in each cluster. We wrote the paper [3] with our results for the GTSP.

The last considered problem is the *Bin Packing Problem with Conflicts* (BPPC), a generalization of the classical *Bin Packing Problem* (BPP) where a set of elements must be load into bins of a given capacity and some pair of elements cannot be placed in the same bin. The paper [12] contains our results to the BPPC.

This thesis is divided into 7 parts, starting with the Introduction, followed by Chapters 1, 2, 3, where we study and propose effective algorithms for repectively, the Fair Layout, the Generalized Traveling Salesman and Bin Packing with Conflicts Problems. In these chapters we give the description of the problems, suggestions of algorithms or models to solve them, computational results and comparisons of the proposed methods with the most effective methods from the literature, and conclusions on the obtained results. We finish with the Conclusions, the Acknowledgements and the Bibliography.

Contents

List of Figures

List of Tables

Chapter 1

Fair Layout Optimization Problem.

1.1 Introduction.

Fairs and expositions are nowadays fundamental tools for providing industrial exhibits and demonstrations. According to the International Association of Fairs & Exposition (IAFE), over 3 200 fairs are currently held in North America each year. The European Major Exhibition Centres Association (EMECA) estimates that more than 36 million visitors and upwards of 330 000 exhibitors take part in roughly 1 000 EMECA exhibitions. A relevant logistic issue in the organization of a fair concerns the way the stands have to be placed in the exhibition space so as to satisfy all constraints (security, ease of access, services, to mention just a few) arising in this kind of event, and to maximize the revenues coming from the exhibitors. Such issue is frequently manually solved by the organizers on the basis of experience and common sense.

In this work we present mathematical models and algorithms for the optimal solution of a basic version of the problem, and describe a real world application dealing with such version. We also instroduce variants and generalizations of the basic methods in order to deal with additional constraints

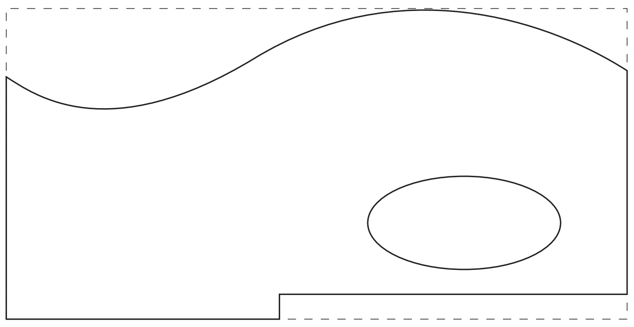


Figure 1.1: A non-convex exhibition area.

or requirements resulting from real world cases.

Given an exhibition surface having irregular (non-convex) shape such as the one shown by the solid lines in Figure 1.1 (disregard by the moment the dotted lines), and an unlimited number of identical rectangular stands, we want to find a feasible layout containing the maximum number of such stands. A layout is feasible if it fulfils a number of basic operational constraints (the stands cannot overlap and must completely lie within the exhibition area; the clients must have an easy access to the stands), plus additional constraints coming from specific requests of the organizers.

In this work we limit our search to a particular case of layout, quite common in practical contexts, in which the stands must all have the same orientation, and must be accommodated into vertical *strips*, with enough space between strips to allow an easy access of the clients. An example is shown in Figure 1.2.

This problem can remind a number of two-dimensional packing problems, such as the two-dimensional bin packing problem and the two-stage two-dimensional cutting stock problem (see, e.g., Lodi, Martello and Vigo [29] and Wäscher, Haußner and Schuman [52], respectively, for recent surveys). It can also remind the facility layout problem, for which we refer the reader to the works by Widmer [53] and Singh and Sharma [48]. Nevertheless, to our knowledge, no study has been devoted to the specific problem we consider.

Although a number of results on various facility layout problems can be found in the literature, only in recent years specific contributions on fair layout were presented. Schneuwly and Widmer [45] presented heuristic algorithms for a specific real world case at the regional fair in Romont (Switzerland). They referred indeed to a *rectangular* exhibition area, did not impose that the stands be placed into vertical strips, and considered stands of six different shapes.

The problem (a real-world case situation arising at the regional fair in Romont, Switzerland) was modeled by discretizing the rectangular exhibition area, and solved through three constructive heuristics, which first place the stands and then construct the space for the aisles. Three possible objective functions were considered: the space utilization, the total attractiveness of the exhibition and the visitor convenience.

In Section 1.2 we present the problems addressed and describe their main features. In Section 1.3 we introduce integer linear programming models based on discretization of the layout space and objects to be placed. A binary matrix encodes the available and not available exhibition areas. It is shown that the matrices associated with the corresponding constraint sets are total unimodular, hence allowing solution through linear programming. The models are tested in Section 1.4 on a real-world case study. In Section 1.5 we consider a case in which blocks of four (2×2) stands have to be used, instead of single stands. Section 1.6 deals with a common topological variant, in which the exhibition area is limited by walls, hence additional accessibility constraints have to be imposed. In Section 1.7 we show how the implemented decision support system can be easily used to optimize cases in which the orientation of the stands is not imposed. Some computational considerations are finally given in Section 1.8.

1.2 The Problem.

We are given:

(i) a non-convex two-dimensional surface that may contain holes (*exhibi-*

tion area);

(ii) an axis-aligned minimal rectangle which encapsulates the exhibition area and touches it on the borders (see the dotted lines in Figure 1.1);

(iii) an unlimited number of identical rectangular stands;

(iv) a minimum width needed for the aisles.

The *Fair Layout Optimization Problem* we consider consists in orthogonally allocating the maximum number of stands, without rotation, to vertical *strips* parallel to the vertical edges of the rectangle, by ensuring left and/or right (see below) side access to each stand.

Concerning the access constraint, we will consider two variants of the problem, that are frequently encountered in practice:

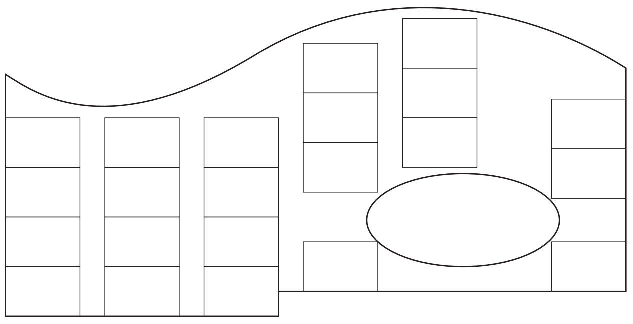


Figure 1.2: A single strip solution.

- FLOP1: it is required that each stand (i.e., each strip) can be accessed from both sides, as in the solution depicted in Figure 1.2;
- FLOP2: it is allowed to place pairs of strips with no space between them, thus obtaining stands that can be accessed from one side only, as in the solution depicted in Figure 1.3.

We will assume, without loss of generality, that the exhibition area can be accessed from all sides. For the cases where such assumption does not hold, the available area can be conveniently restricted (as it will be clear later).

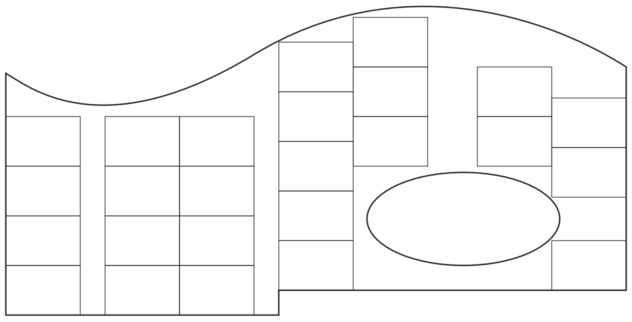


Figure 1.3: A double strip solution.

We will also assume that the width and height (W and H) of the exhibition area, the width and height (w and h) of the stands and the minimum aisle width (a) are integers, i.e., that they are expressed in the minimum unit length, say δ, that is convenient to evaluate (typical δ values in real-world applications are in the range 5–20 cm). This is obtained by rounding down the rectangle sizes, and by rounding up the stand sizes and the aisle width, so as to guarantee a feasible solution.

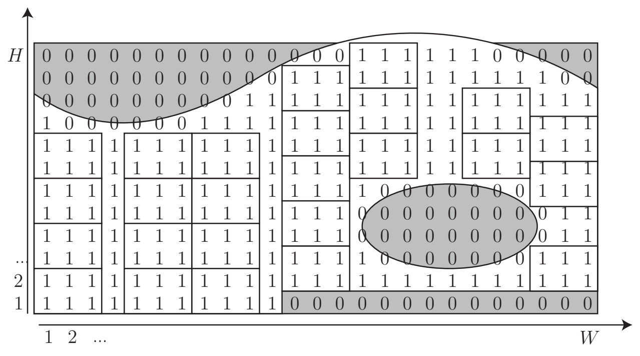


Figure 1.4: Matrix M associated with the exhibition area of Figure 1.1.

Let M be an $H \times W$ binary matrix corresponding to the encapsulating rectangle (see Figure 1.4). For convenience of notation, we will number the rows and columns of M starting from the bottom-left corner, so that the indices correspond to coordinates in the corresponding discretized space.

Matrix M is then defined as

$$M_{ij} = \begin{cases} 1 & \text{if the } \delta \times \delta \text{ square located at coordinate } (i,j) \text{ can be used for a stand;} \\ 0 & \text{otherwise,} \end{cases} \tag{1.1}$$

for $i = 1, \ldots, H$ and $j = 1, \ldots, W$.

We say that a column j is selected for the layout if a strip of stands is placed with the left edge of the stands on column j. For example, column 1 is selected for the first strip of Figures 1.2 and 1.3 (where $w = 3$, $h = 2$ and $a = 1$).

Consider any column j, and any maximal row interval $[i_r, i_s]$ such that matrix M has all 1's in the rectangle of width w and height $(i_s - i_r + 1)$ having its bottom-left corner in (i_r, j). The maximum number of stands that can be placed within such a rectangle is $\lfloor (i_s - i_r + 1)/h \rfloor$. In the example of Figure 1.4 we have, for $j = W - 3$, two maximal row intervals of width 3: one of height 2 and one of height 5. The corresponding strip can thus accommodate three stands in total.

For any column $j \leq W - w + 1$, the maximum number of stands v_j that can be placed with their left edges on column j can be obtained in a greedy way by iteratively determining the next group of consecutive rows satisfying the above condition. A straightforward implementation of such a method would require, for each column, $O(H)$ iterations, each of time complexity $O(w)$, i.e., in total $O(WHw)$ time. We next give a simple procedure that computes the same information v_j $(j = 1, \ldots, W - w + 1)$ in $O(WH)$ time, i.e., in the order of time proportional to the size of the input of matrix M.

We define, for each row i such that $M_{i,j} = 1$ (j being the current column), a pointer $p(i)$ to the last column $\hat{\jmath}$ such that $M_{i,j} = M_{i,j+1} = \cdots = M_{i,\hat{\jmath}} = 1$. If instead $M_{i,j} = 0$ for the current column j then $p(i)$ has any value less than j. In this way all maximal row intervals can be determined by examining each element of M a constant number of times. Counter k stores, for the current column j and the current row interval, the number of feasible rows. The detailed procedure is given in Algorithm 1.

Algorithm 1 Computation of the v_j values.

```
Procedure Alloc_Stands
for $i := 1$ to $H$ do
    $M_{i,W+1} := 0$;
    if $M_{i,1} = 1$ then $p(i) := \min\{k : M_{i,k} = 0\} - 1$ else $p(i) = 0$
end for;
for $j := 1$ to $W - w + 1$ do
    $v_j := k := 0$;
    for $i := 1$ to $H$ do
        if $p(i) - j + 1 \geq w$ then $k := k + 1$
        else
            $v_j := v_j + \lfloor k/h \rfloor$;
            $k := 0$;
            if $(p(i) < j$ and $M_{i,j+1} = 1)$ then $p(i) := \min\{q > j : M_{i,q} = 0\} - 1$
        end if
    end for;
    $v_j := v_j + \lfloor k/h \rfloor$
end for
end
```

The inner 'if' statement is executed $O(WH)$ times. The search for the new $p(i)$ value, say $p'(i)$, is only performed when $j > p(i)$, and requires a loop going from $j + 1$ to $p'(i) + 1$. The next such search will only be performed when $j > p'(i)$, implying that each element of M is examined at most once. The overall time complexity of the algorithm is thus $O(WH)$.

Our combinatorial optimization problem thus reduces to determining which columns j should be used for packing the strips of stands. (Note that, if a column j is selected, the placement of the v_j stands is straightforward.)

1.3 Mathematical Models.

Let us associate a binary variable x_j,

$$x_j = \begin{cases} 1 & \text{if a single strip of stands has its left edges on column } j, \\ 0 & \text{otherwise,} \end{cases} \tag{1.2}$$

to each column j where a strip can be selected, i.e., for $j = 1, \ldots, W - w + 1$. Our first problem can then be modeled as

$$\text{(FLOP1)} \qquad \max \sum_{j=1}^{W-w+1} v_j x_j \tag{1.3}$$

$$\text{s.t.} \sum_{j=k-a-w+1}^{k} x_j \leq 1 \qquad (k = w + a, \ldots, W - w + 1), \tag{1.4}$$

$$x_j \in \{0, 1\} \qquad (j = 1, \ldots, W - w + 1). \tag{1.5}$$

The set of constraints (1.4) imposes that the stands do not overlap and there is enough space left for the aisles: if column k is selected then columns $k - a - w + 1, \ldots, k - 1$ cannot be selected (see Figure 1.5 (a)). Note that the constraints (1.4) for $k = 1, \ldots w + a - 1$ are not imposed, as they are dominated by the first constraint ($k = w + a$).

Problem FLOP2 can be modeled in a similar way by introducing a second set of binary variables for each column j, namely

$$\xi_j = \begin{cases} 1 & \text{if a double strip of stands has its leftmost edges on column } j, \\ 0 & \text{otherwise,} \end{cases} \tag{1.6}$$

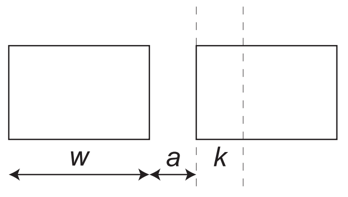
for $j = 1, \ldots, W - 2w + 1$. Observe that the maximum number of stands that can be placed in a double strip with the leftmost edges on column j is $v_j + v_{j+w}$. We thus obtain the model

$$\text{(FLOP2)} \ \max \sum_{j=1}^{W-w+1} v_j x_j + \sum_{j=1}^{W-2w+1} (v_j + v_{j+w})\xi_j \tag{1.7}$$

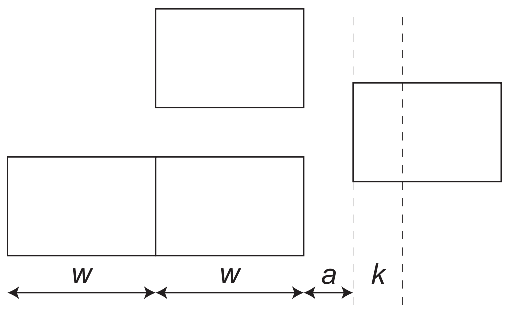
$$\text{s.t.} \sum_{j=k-a-w+1}^{k} x_j + \sum_{j=\max(1,\ k-a-2w+1)}^{\min(W-2w+1,k)} \xi_j \leq 1 \qquad (k = w + a, \ldots, W - w + 1), \tag{1.8}$$

$$x_j \in \{0, 1\} \qquad (j = 1, \ldots, W - w + 1), \tag{1.9}$$

$$\xi_j \in \{0, 1\} \qquad (j = 1, \ldots, W - 2w + 1). \tag{1.10}$$



(a) Constraint (1.4).



(b) Constraint (1.8).

Figure 1.5: Constraints visualization.

The set of constraints (1.8) imposes that the single and double strips of stands do not overlap, and there is enough space left for the aisles: if column k is selected (ether for a single or a double strip) then (see Figure 1.5 (b)) columns $k - a - w + 1, \dots, k - 1$ cannot be selected for a single strip, and columns $k - a - 2w + 1, \dots, k - 1$ cannot be selected for a double strip. The max and min operators in the second summation exclude indices j that are out of range.

The structure of the constraint matrix induced by (1.8) is shown in Figure 1.6, which provides the constraint matrix corresponding to the instance depicted in Figure 1.4. (Additionally note that the leftmost portion of such matrix gives the structure of the constraint matrix induced by (1.4).)

It is not difficult to see that our two models possess the following relevant property.

Property 1. *The matrices associated with the constraint sets* (1.4) *of FLOP1 and* (1.8) *of FLOP2 are totally unimodular.*

Proof. Recall that a $0 - 1$ matrix in which each column has its ones consecutively is called an *interval matrix.* It is known (see, e.g., Schrijver [46], Section 19.3) that interval matrices are totally unimodular. The coefficients in each row i induced by (1.4) are all zeroes but a set of $w + a$ consecutive ones starting in column i (see the leftmost part of Figure 1.6), hence the resulting constraint matrix is an interval matrix. The coefficients

$$
\begin{array}{r|*{23}{c}|*{20}{c}|}
 & x_1 & x_2 & \dots &&&&&&&&&&&&&&&&&&&&& \xi_1 & \xi_2 & \dots \\
\hline
k{=}w{+}a & 1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 & 1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
k{=}w{+}a{+}1 & 0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 & 1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
\dots & 0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 & 1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
 & 0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 & 1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0 \\
 & 0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 & 0&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0 \\
 & 0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0 & 0&0&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0 \\
 & 0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0 & 0&0&0&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0 \\
 & 0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0 & 0&0&0&0&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0 \\
 & 0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0 & 0&0&0&0&0&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0 \\
 & 0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0 & 0&0&0&0&0&0&1&1&1&1&1&1&1&0&0&0&0&0&0&0 \\
 & 0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0 & 0&0&0&0&0&0&0&1&1&1&1&1&1&1&0&0&0&0&0&0 \\
 & 0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0 & 0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&0&0&0&0&0 \\
 & 0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0 & 0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&0&0&0&0 \\
 & 0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0 & 0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&0&0&0 \\
 & 0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0 & 0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&0&0 \\
 & 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0 & 0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&0 \\
 & 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0 & 0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1 \\
 & 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0 & 0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1 \\
 & 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0 & 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1 \\
 & 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1 & 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1 \\
\hline
\end{array}
$$

Figure 1.6: Constraint matrix for model FLOP2 associated with Figure 1.4.

in each row i induced by (1.8) are all zeroes but two sets of consecutive ones: one starting in column i for the x variables, and one starting in column $\max\{W-w+2, W-w+i-2\}$ for the ξ variables (refer again to Figure 1.6). Each column of the resulting constraint matrix has thus all zeroes but a set of consecutive ones, from which one has the thesis. $\triangle$

It follows from Property 1 (see, e.g., Schrijver [46]) that both problems FLOP1 and FLOP2 can be solved in polynomial time by relaxing the integrality constraints, i.e., by replacing (1.5) and (1.9) with

$$0 \leq x_j \leq 1 \qquad (j = 1, \dots, W - w + 1) \tag{1.11}$$

and (1.10) with

$$0 \leq \xi_j \leq 1 \qquad (j = 1, \dots, W - 2w + 1), \tag{1.12}$$

and solving the resulting linear programming problem.

We conclude with the following

Observation 1. *The generalization of* (1.7)-(1.10) *to handle triple, quadruple, ..., k-tuple strips produces ILP models that maintain the total unimodularity property.*

Indeed the resulting constraint matrix has the same structure as the one shown in Figure 1.6 with the addition, to the right, of $k-2$ blocks having the same shape as the rightmost block in the figure. Hence each column still consists of all zeroes but a set of consecutive ones. This observation is mainly of theoretical interest: in our specific application the stands in the second, third, ..., $(k-1)$th column of a k-tuple strip $(k > 2)$ would be unaccessible.

1.4 Model Application.

The Java implementation of the above models and the linear programming solver `lp_solve` Version 5.5. (see `http://sourceforge.net/projects/lpsolve/`) were used for constructing a decision support system. We first tested it by determining layouts for the permanent "Beira Mar" handcraft fair of Fortaleza (Brazil), whose exhibition area is shown in Figure 1.7.

The fair is located in an area approximately 200 meters wide and 55 meters high. Its current configuration consists of 617 stands, placed along 26 double strips roughly similar to those produced by FLOP2. The width and height of each stand are 2 meters.

The available area for placing the stands is depicted in grey in Figure 1.7. The area is non-convex due to the presence of palm trees, small shopping facilities, light poles, a monument and some sub-areas which cannot be used

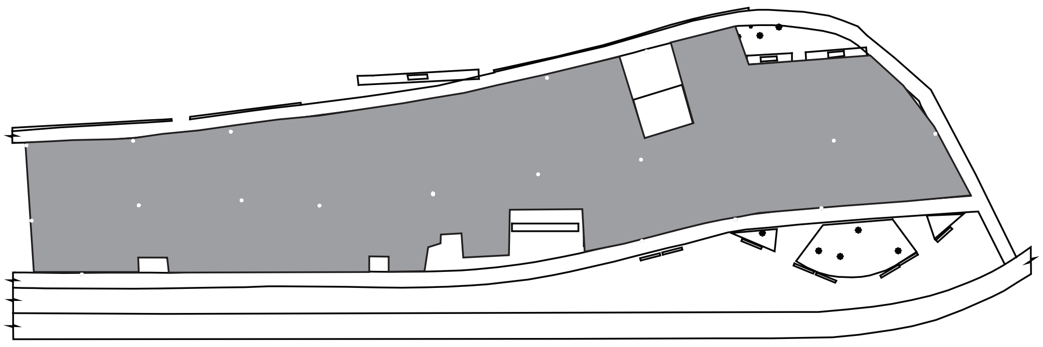


Figure 1.7: The Beira Mar handcraft fair exhibition area.

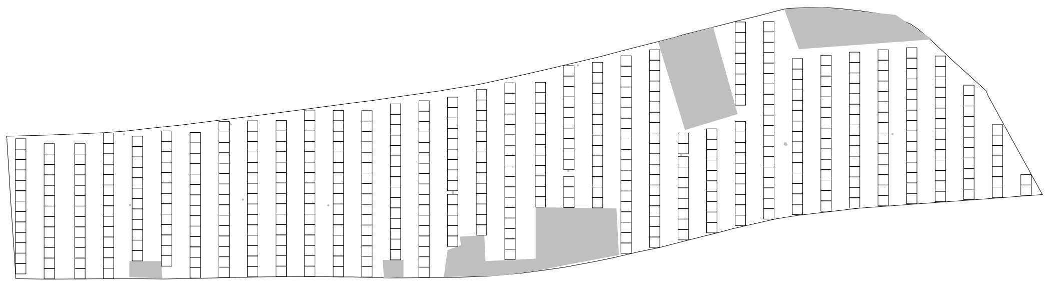


Figure 1.8: Solution produced by FLOP1.

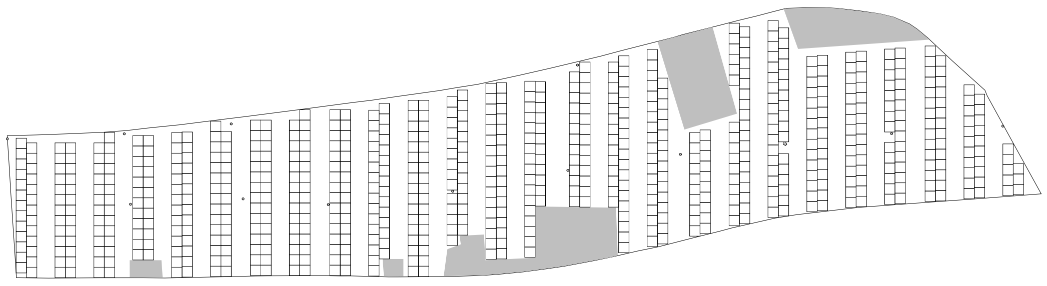


Figure 1.9: Solution produced by FLOP2.

for the stands.

We solved the problem by setting $\delta = 5$ cm, thus obtaining $w = h = 40$, $W = 4090$ and $H = 1084$. The aisle width was set to the same value currently adopted, i.e., $a = 70$ (3.5 meters). In order to test the algorithm we solved both FLOP1 (very different from the current configuration) and FLOP2.

Model FLOP1 was constructed and solved in 1.6 CPU seconds. The resulting solution, shown in Figure 1.8, consists of 505 stands placed along 36 single strips, i.e., with less than 20% decrease with respect to the current double strip configuration.

Model FLOP2 was constructed and solved in 4.81 CPU seconds. The solution, shown in Figure 1.9, includes 742 stands placed along 26 double strips, increasing by 20% the current number of stands. The decision makers wanted to also test the possibility of allowing the use of the roughly triangular area located in the upper right corner of the exhibition area. The outcome was an increase to 759 stands placed along 26 double strips. None of the solutions produced by FLOP2 combined double and single strips, probably due to the fact that the aisles are relatively quite large, thus making their use not convenient.

1.5 Blocks of stands.

Upon examining the solution shown in Figure 1.9, the fair organizers wanted to examine a different layout, obtained by imposing *blocks* of four contiguous stands (2×2), arranged into single strips of blocks with horizontal aisles of minimum height b separating adjacent blocks, besides the vertical aisles of minimum width a separating the strips (see, e.g., Figure 1.10). Although it was clear that the resulting number of stands would be smaller, the rationale was to facilitate the circulation of visitors around the stands.

The resulting problem was solved by defining in a different way the the v_j values used in the model. For each column j ($j = 1, \ldots, W - 2w + 1$), let v_j be the number of stands that can be arranged in a single strip of blocks having the left edges of the blocks on column j. This can be obtained by computing, for the given j, all maximal sets of consecutive row intervals $[r, s]$ such that $M_{ik} = 1$ for $i = r, r+1, \ldots, s$ and for $k = j, j+1, \ldots, j + 2w - 1$. For each such interval, the maximum number of blocks that can be feasibly arranged is then given by

$$\beta(r, s) = 1 + \left\lfloor \frac{(s - r + 1) - 2h}{2h + b} \right\rfloor \tag{1.13}$$

(note that no horizontal aisle is needed below the first block or above the last one). The required total number of stands v_j is then four times the sum, over all intervals $[r, s]$ of column j, of the $\beta(r, s)$ values.

Once the new v_j values have been computed, the problem is solved by FLOP1, with 'w' replaced by '$2w$' to take into account the fact that each single strip has length $2w$. In Figure 1.10 we show the solution obtained for the same input data as before and aisles of minimum size $b40$. The layout

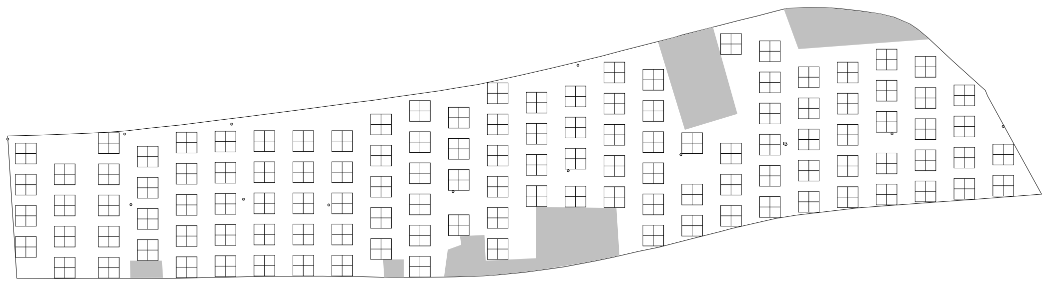


Figure 1.10: Second Fortaleza layout.

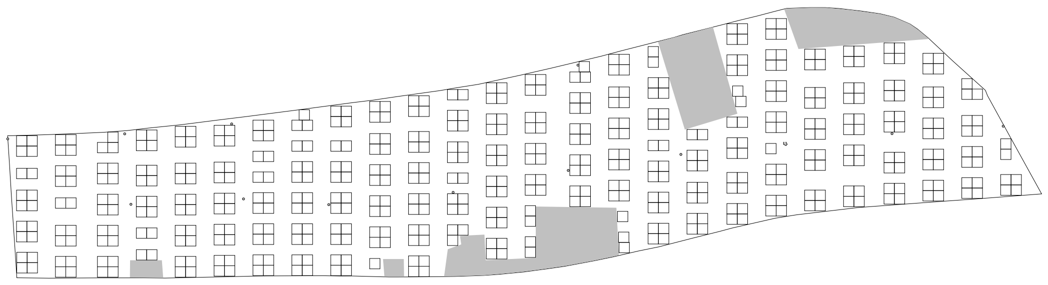


Figure 1.11: Third Fortaleza layout.

consists of 26 strips of blocks, allocating in total 21 blocks (484 stands).

A third intermediate solution was also requested, in which incomplete blocks (due, for example, to forbidden areas) are accepted. This can be modeled by considering a strip of 2×2 blocks at a time. For each column j, we consider the exhibition sub-area given by columns $j, \dots, j + 2w - 1$, and rotate it by 90° degrees. We then compute the optimal solution given by FLOP2 for the resulting sub-area. Note that such sub-area is encapsulated by a rectangle of width $\widetilde{W} = H$ and height $\widetilde{H} = 2w$, and that FLOP2 has to be run with aisle size $\widetilde{a} = b$, and stand sizes $\widetilde{w} = h$ and $\widetilde{h} = w$. The number of stands given by the double strip solution found by FLOP2 is then assigned to v_j, to represent the number of (possibly incomplete) blocks having their leftmost edges on column j. For example, the rotated counterpart of the leftmost column of blocks in Figure 1.10 produces a FLOP2 solution consisting of 3 consecutive double strips, followed by a single strip, followed by a double strip (see the leftmost strip of blocks in Figure 1.11). The corresponding v_j value is thus $v_1 = 18$.

Once all v_j values have been computed, the overall solution is obtained as before, by executing FLOP1 on the original exhibition area, with 'w' replaced by '$2w$'. The solution obtained for the Fortaleza fair is shown in Figure 1.11. Note several 2×1 partial blocks, some single stand blocks and some blocks consisting of 3 stands. The solution still consists of 26 strips of blocks, but the overall number of stands is now 512, with an increase of roughly 6%. This was the solution preferred by the organizers.

1.6 Non-accessible borders.

The exhibition area of the Fortaleza fair discussed in the previous sections takes place in an open space, so there is no problem of accessibility to the stands touching the borders. A different situation occurred for the fair of Reggio Emilia (Italy), whose exhibition area is inside a building, i.e., the borders are constituted by walls. In Figure 1.12 the walls are represented by the thick black lines, while the forbidden (open) spaces are represented by the grey non-dashed areas. The grey dashed areas (to be discussed below) are non-forbidden areas. The input data were $\delta = 5$ cm, $w = h = 80$, $W = 2850$, $H = 2394$ and $a = 60$.

Executing FLOP2 with no restriction produced a solution with 514 stands. Such solution was however clearly unacceptable, as it included long strips of stands terminating on the walls. Although all stands were accessible from the central large aisle that is visible in the figure, visitors would have been obliged to walk back along the same aisle once the wall was reached. In order to evaluate acceptable layouts, we thus implemented two variations of the basic model, by

1. preliminary imposing an alley of width a along the walls, or
2. preliminary allocating stands along the walls.

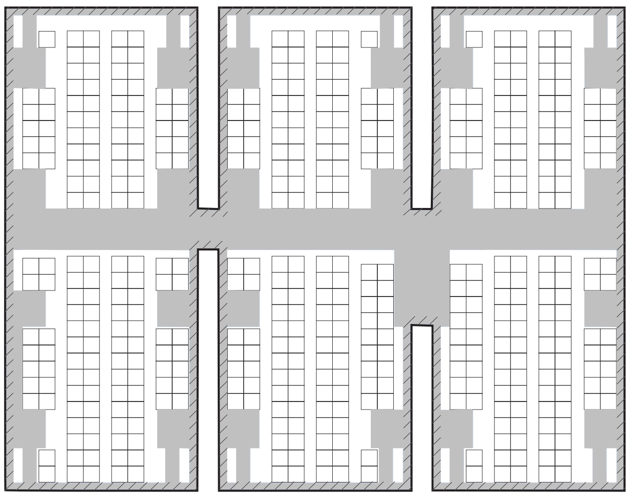


Figure 1.12: First Reggio Emilia layout.

The first run was thus executed by forbidding the grey dashed areas of Figure 1.12, and executing FLOP2 on the resulting (white) areas. The layout, shown in the same figure, consists of 461 stands. All stands are easily accessible, but the number of stands is considerably smaller than the one in the above mentioned (unacceptable) solution.

The second layout was obtained by preliminary allocating in a greedy way a series of stands touching the walls on one side and having an aisle of width a on the opposite side, as shown in Figure 1.13. This kind of solution turns out to be frequently adopted in practice, as can be seen from fair layouts available on the internet such as, e.g., in the Italian market, those of Modena, Carrara and Lingotto (Turin). The overall solution shown in Figure 1.13 was then obtained by running FLOP2 on the remaining available area. Quite surprisingly, the resulting number of stands was 514, i.e., exactly the same as in the unacceptable solution mentioned above.

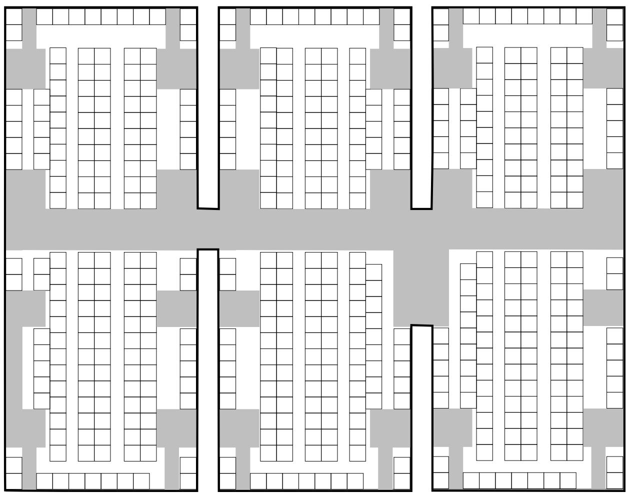


Figure 1.13: Second Reggio Emilia layout.

1.7 Non-oriented layouts.

In the layouts considered so far the orientation of the stands (and hence of the strips) was prefixed. This is quite usual when the exhibition area is inside a building, although rotations by 90° can sometimes be evaluated. For the case of open space exhibition instead, any stand rotation can virtually be implemented.

We thus added to our decision support system a feature that allows one to examine different orientations, as well as a tool that outputs the orientation producing the highest number of stands. Let ϑ denote the number of degrees by which the stands are rotated (clockwise) with respect to the basic orientation, and let $z(\vartheta)$ be the number of stands that FLOP2 allocates with ϑ rotation. Observe that $z(\vartheta)$ is a general non-convex function (see Figure 1.14). For each ϑ value we thus produced the corresponding 0-1 matrix M (see Section 1.2) and found the optimal solution.

We experimented the above tool on the open space Beira Mar fair examined in Section 1.4. For a general case rotations between 0 and 180° should be evaluated. As in this specific case the stands are squares, it was enough to evaluate the range 0-90°. The result is shown in Figure 1.14. Remind

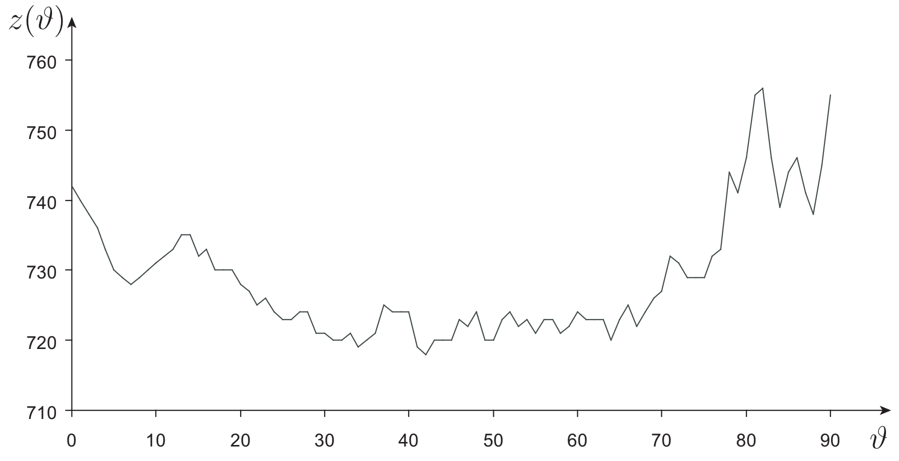


Figure 1.14: Rotation.

that the orientation currently adopted, corresponding to $\vartheta = 0$, allocates 742 stands. The highest values of $z(\vartheta)$ were obtained in the range 80°–90°, with an absolute maximum of 756 stands for 82°. It should be observed, however, that implementing such solutions could require additional aisles to avoid too long strips.

1.8 Computational Considerations.

In Table 2.3 we give some details on the computational performance of the various models. Each line refers to a specific experiment. Columns H, W, h, w, a and b report the input data. Columns T, T–M, T–v_j and T–solve give, respectively, the total CPU time, and the times needed to produce matrix M, to determine the v_j values and to solve the linear program. All runs have been executed on a Pentium IV 3.2 GHz, 1GB RAM, running under a Linux operating system.

All solution times are quite low, with the only exception of the time needed to compute the v_j values for the solution made up by incomplete blocks of stands (Figure 1.11). On the other hand such computational effort

Table 1.1: Computational results.

Instance	H	W	h	w	a	b	T	T–M	T–v_j	T–solve	# stands
Figure 1.8	1084	4090	40	40	70	–	5.09	3.34	0.13	1.63	504
Figure 1.9	1084	4090	40	40	70	–	8.27	3.34	0.13	4.81	742
Figure 1.10	1084	4090	40	40	70	40	4.93	3.34	0.11	1.49	484
Figure 1.11	1084	4090	40	40	70	40	448.77	3.34	443.80	1.63	512
Figure 1.12	2394	2850	80	80	60	–	3.27	2.13	0.06	1.08	461
Figure 1.13	2394	2850	80	80	60	–	2.82	1.84	0.07	0.91	514

was rewarded by a considerable increase of the overall number of stands (from 484 to 512).

1.9 Conclusions.

We addressed a particular layout problem derived from a real-world situation. We proposed two ILP models and showed that their constraint matrices are totally unimodular. The solutions obtained have been given to the local authorities, that are currently discussing the new fair layout configuration to be implemented.

Our models are based on discretization of the layout space and objects to be placed. The same technique could be used to model other packing problems, such as the two-dimensional bin packing problem or the two-stage two-dimensional cutting stock problem. In these problems the layout space is a special case of the one we considered, as it is restricted to rectangles. The objects to be placed, however, are more general, as they consist of non-identical rectangles that need not to be arranged into strips. The latter consideration shows however that the resulting models would not possess the total unimodularity property, that directly comes from our specific constraints (see the proof of Property 1).

We have examined a relevant logistic issue arising in the organization of fairs, namely the optimization of the stands allocation in the exhibition space.

We have reviewed two mathematical models for the optimal solution of a basic version of the problem. In real world applications the constraints and requirements (secu- rity, ease of access, services) can vary considerably from case to case. We have examined a number of practical situations, and the corresponding requirements, and we have shown how the two basic mathematical models, and the corresponding decision support system, can be adapted to handle them. The conclusion of this study is that these mathematical models are flexible enough to be used with satisfactory results in a variety of different real world situations.

Chapter 2

Generalized Traveling Salesman Problem.

2.1 Introduction.

The *Generalized Traveling Salesman Problem* (*GTSP*) is a variant of the Traveling Salesman Problem (*TSP*). We are given an undirected graph $G = (V, E)$, where $V = \{1, \ldots, n\}$ is the set of vertices and E is the set of edges, each edge (i, j) having an associated cost c_{ij}. The set of vertices V is partitioned into m clusters $V_1, \ldots, V_m$. *GTSP* is to find an elementary cycle visiting at least one vertex for each cluster, and minimizing the sum of the costs of the traveled edges. If a directed graph is considered, the problem is denoted as *Asymmetric GTSP* (*AGTSP*). We focus on the commonly considered version of the problem, i.e. the so-called *Equality GTSP* (*E-GTSP*), in which the cycle has to visit exactly one vertex for each cluster.

Both the *GTSP* and the *E-GTSP* are generalizations of the *TSP*: we obtain the *TSP* in the particular case where each cluster is composed by just one vertex. Thus both problems are NP-Hard since the *TSP* is NP-Hard.

In the next section, we give a short literature review of some important previous works on the *GTSP*.

2.2 Literature Review.

The *GTSP* was introduced simultaneously by Srivastava et al. [50] and Henry-Labordere [19] in 1969. In 1970 Saksena [44] studied the symmetric and asymmetric cases. The *GTSP* has been studied later by Laporte et al. in [28] and [27], by Noon in his Ph.D. thesis [38] and by Noon and Bean in [39] and [40]. The most studied version of the problem is the one called *E-GTSP*. To the best of our knowledge, the name *E-GTSP* was first introduced by Fischetti et al. [14], even if already in previous papers this version of the problem was studied. In [13] Fischetti et al. proposed a Branch and Cut algorithm to solve *E-GTSP* to optimality. They also proposed two heuristics for *E-GTSP*. The first one is a cycle construction algorithm, which is an adaptation of the Farthest Insertion TSP procedure. Analogous algorithms can be obtained by adapting the Nearest Insertion and the Cheapest Insertion TSP procedures. This algorithm is combined with two cycle improvement procedures: the first one is based on 2-opt and 3-opt exchanges; the second one starts from a given sequence of the clusters and computes the best feasible cycle, visiting the clusters in the given order, by using a Layered Network and solving shortest path problems. Consider the visiting order of the clusters to be fixed as $V_1, \dots, V_m$. The Layered Network has $m+1$ layers corresponding to the m clusters $V_1, \dots, V_m$ and to an additional cluster, V_{m+1}, which is a copy of cluster V_1. The Layered Network contains all the vertices of G, and one copy of each vertex in V_1. It contains an arc (i, j), for each $i \in V_h$ and $j \in V_{h+1}$, $(h = 1, \dots, m)$, with cost c_{ij}. Any path in the Layered Network, going from a vertex u in V_1 to the corresponding vertex v in V_{m+1}, is a feasible solution to *E-GTSP*, since it defines a cycle in G which visits each cluster exactly once. Moreover, every *E-GTSP* solution which visits the clusters in the order $V_1, \dots, V_m$ corresponds to a path in the described Layered Network. Thus, finding the best cycle, once the visiting order of the clusters is fixed, can be done by finding the shortest path from each vertex u in V_1 to the corresponding vertex v in V_{m+1}. In the following we will denote this method as the *Layered Network Method*. The second heuristic proposed by Fischetti et al. [13] is based on a Lagrangian Relaxation of the problem, followed by the second improvement procedure, and is executed in a subgradient optimization framework. Fischetti et al. in [13] also introduced a clustering procedure to obtain benchmark instances of the *E-GTSP* starting from instances of the TSPLIB Library (Reinelt [42]). In a previous paper [14], Fischetti et al. studied the facial structure of the polytopes associated with the *GTSP* and the *E-GTSP*.

In [43] Renaud and Boctor presented a classification of the different heuristics that can be used to solve the problem. The algorithms can be divided into three classes: the decomposition algorithms, the construction algorithms and the solution improvement algorithms. The algorithms of the first two classes produce a feasible solution to the problem, which can be improved by the heuristics of the third class. The decomposition algorithms subdivide the problem into two phases. In the first phase the algorithm selects the vertices to be visited (one for each cluster) and in the second phase it constructs a cycle, for example using a TSP heuristic. Alternatively, in the first phase the algorithm determines the order in which the clusters should be visited (for example by using a TSP heuristic) and in the second phase constructs the shortest cycle by using the Layered Network Method. The construction heuristics simultaneously choose the vertices to be visited and construct the cycle, for example by using adaptations of TSP construction heuristics (e.g., nearest neighbor, cheapest insertion, farthest insertion). A well-known construction algorithm is the one proposed by Noon in [38]: this is a Nearest Neighbor algorithm, adapted for the *E-GTSP*. Finally, the solution improvement algorithms start from a feasible solution and try to improve it, by using local search procedures, such as 2-opt or 3-opt or specific adaptations for the *E-GTSP*. Renaud and Boctor [43] proposed a composite heuristic, composed of three phases. They call this algorithm GI^3, which stands for Generalized Initialization, Insertion and Improvement. In the first phase they construct a sub-cycle, and in the following phase they apply an insertion procedure to obtain a cycle which visits exactly one vertex in each cluster. Finally, they apply a solution improvement procedure.

In [49] Snyder and Daskin proposed a random-key genetic algorithm. The encoding of an E-GTSP solution is as follows: each cluster V_i, $(i = 1, \ldots, m)$, has a *gene* consisting of an integer part (drawn from $\{1, \ldots, |V_i|\}$) and a fractional part (drawn from $[0, 1)$). The integer part indicates which vertex from the cluster is included in the cycle, and the vertices are sorted according to their fractional part to indicate the order of the clusters in the cycle. In the *crossover* operator, two parent solutions are randomly chosen from the current population. One offspring is then generated from the two parents by using a parametrized uniform crossover, i.e. by taking each gene from parent 1 with probability 0.7 and from parent 2 with probability 0.3. The initial population is created by randomly generating 100 solutions, drawing the gene for cluster V_i, $(i = 1, \ldots, m)$ uniformly from $[1, |V_i| + 1)$. At each iteration 20% of the population comes directly from the previous population via *reproduction*, i.e. by copying the best solutions, 70% is obtained via crossover and the remaining 10% is generated via *immigration* (by generating new solutions randomly, as done for the initial population). Every time a new solution is created, an attempt is done to improve it by using 2-opt and "swap" procedures. The "swap" procedure involves the removal of the vertex currently selected for a cluster V_i and the insertion of a different vertex from V_i into the cycle. The insertion is done by using a modified cheapest insertion criterion, so that the new vertex may be inserted into the cycle in a position different from that of the original vertex.

Pintea et al. in [41] proposed a meta-heuristic, based on an Ant Colony System. They call the algorithm *Reinforcing Ant Colony System* (RACS), since it uses new pheromone rules and pheromone evaporation techniques. They present an experimental comparison of their algorithm with the Nearest Neighbor algorithm by Noon [38], the composite heuristic by Renaud and Boctor [43]) and the genetic algorithm by Snyder and Daskin [49]. However, the authors do not report the corresponding computing times (they only say that the algorithm was run with a time limit of 10 minutes).

In a recent work [47], Silberholz and Golden proposed a new genetic algorithm. Each chromosome is given by a sequence of vertices, representing the visited cycle. This is a simple representation, called path representation, which however has the drawback that a randomly selected representation has no guarantee to be feasible for E-GTSP, unless specialized procedures are used. At the beginning, each chromosome is generated by selecting random vertices and adding them to the new chromosome one by one, provided that another vertex from the same cluster had not yet been selected. The initial population consists of 50 chromosomes. Two new crossover operators are introduced, which are based on the TSP ordered crossover (OX): the rotational and reverse crossover (rOX) and its modification (mrOX). Since the new operators are characterized by many features, we suggest the readers to refer to [47] for further details. The authors proposed also a new population structure, which considers several independent small groups of chromosomes for a relatively short computing time at the beginning of the solution procedure, and uses less computationally intensive genetic procedures and local improvement to rapidly generate a subset of reasonable solutions for each group. Then, the best chromosomes from each group are merged into a final population, which is improved by using a standard genetic algorithm structure. Local improvements are also applied: the 2-opt procedure and the swap operator introduced in [49]. To favor population diversity, each chromosome has a 5% probability of being selected for mutation. The algorithm is terminated when it does not improve the incumbent solution for 150 iterations. The algorithm was tested on the benchmark instances by Fischetti et al. [13], and on a new set of larger instances obtained from the TSPLIB data sets.

A condensed literature review is shown in Table 2.1.

In this work we present a new heuristic. It is a multi-start algorithm, which iteratively starts with a randomly chosen set of vertices and applies a decomposition approach, combined with improvement procedures. The decomposition approach considers a first phase to determine the visiting order of the clusters and a second phase to find the corresponding minimum cost

Authors	Date	Topic
Srivastava et al. [50]	1969	Dynamic programming (GTSP)
Henry-Labordere [19]	1969	Dynamic programming (AGTSP)
Saksena [44]	1970	Application in scheduling problems
Laporte and Nobert [28]	1983	Branch & bound (GTSP)
Laporte et al. [27]	1987	Branch & bound (AGTSP)
Noon [38]	1988	Nearest Neighbor heuristic (AGTSP)
Noon and Bean [39]	1991	Lagrangian based approach (AGTSP)
Noon and Bean [40]	1993	Transformation of the AGTSP into a Clustered TSP, Transformation of the GTSP into an asymmetric TSP
Fischetti et al. [14]	1995	Facial structure of polytopes (GTSP and E-GTSP)
Laporte et al. [25]	1996	Applications
Fischetti et al. [13]	1997	Branch-and-Cut (E-GTSP) Heuristic and improvement procedures (E-GTSP), Benchmark instances up to 442 vertices, solved to optimality (E-GTSP)
Renaud and Boctor [43]	1998	Composite Heuristic GI^3 (E-GTSP), instances up to 442 vertices
Snyder and Daskin [49]	2006	Random-Key Genetic Algorithm (E-GTSP) instances up to 442 vertices
Pintea et al. [41]	2007	Reinforcing Ant Colony System (E-GTSP) instances up to 442 vertices
Silberholz and Golden [47]	2007	Genetic Algorithm (E-GTSP), Benchmark instances up to 1084 vertices

Table 2.1: Literature Review

cycle.

The chapter is organized as follows: we present the multi-start heuristic in Section 2.3; in Section 2.4 we give computational results on benchmark instances from the literature and a comparison between the results of the proposed algorithm and the best results found in the literature. Finally we give some conclusions in Section 2.5.

2.3 A Multi-Start Heuristic.

In a decomposition algorithm, the problem is subdivided into two subproblems. As previously mentioned, there are two ways of decomposing the problem (see Renaud and Boctor [43]). One possibility is to select the vertices to be visited, and then construct a minimum cost cycle that visits the selected vertices; another possibility is to determine the order in which the clusters are visited, and then construct the corresponding minimum cost cycle, by selecting one vertex for each cluster.

Our method is an extension of this class of decomposition algorithms, since it combines these two approaches, alternating these two ways of decomposition of the problem and introducing as well some randomness to explore a larger solution space. Moreover, we apply local search procedures, to improve the solution found. The name *"Multi-Start Algorithm" (MSA)* underlines that we iteratively repeat the algorithm, starting at each *iteration* with a different initial set of vertices. *MSA* considers a first phase to determine the visiting order of the clusters and a second phase to find the corresponding minimum cost cycle. The visiting order of the clusters is obtained as follows: at the first iteration we randomly choose, with uniform probability, one vertex in each cluster; we then compute, by using the Farthest Insertion TSP (FITSP) procedure, followed by a Lin-Kernighan (L-K) and a swap improvement procedures, a TSP feasible solution for the subgraph induced by the selected vertices.

The FITSP procedure is applied to get the initial visiting order of the clusters for the first iteration and whenever the algorithm achieves $MAXI$ consecutive iterations without improvement of the *reference solution*. We call *reference solution* the solution which is initially determined by the FITSP procedure and is updated when a lower cost current solution is detected. Note that the *reference solution* may be different from the best solution found so far, since we want to escape from possible local minima. In the other iterations, the visiting order of the clusters is determined by the corresponding order in the *reference solution*.

Once the order of the clusters is fixed, the second phase starts: the Layered Network Method is applied. It computes the shortest cycle which visits exactly one vertex in each cluster. This phase gives a set of vertices which can be different from the one obtained in the first phase. In this case, we apply the L-K and swap improvement procedures to the new vertex sequence, allowing a change in the order of the clusters. If the order of the clusters is changed, we apply again the Layered Network Method in an iterative way. Otherwise the current solution cannot be further improved, so we start a new iteration with a new set of randomly chosen vertices. In addition, we apply a probabilistic step: with probability p each vertex in the chosen set is substituted by the vertex corresponding to the same cluster in the current *reference solution*. The approach iteratively repeats these steps until a stop condition (e.g., time limit) is reached.

Since the probability p is a very important parameter of our approach, as it makes the method more or less conservative, we change the probability during the execution. In particular, its value increases from an initial probability p_{in} to a final probability p_{fin}. The probability is changed every $MAXP$ iterations. At the beginning the steps for changing the probability are larger and later on they become smaller. The rule for defining the probability is the following: $p = p_{fin} - \lfloor p_{fin}/(C_p + 2) \rfloor$, where C_p represents the number of probability values already considered (initially $C_p = 0$). When the final value p_{fin} is reached, in order to avoid to be stuck at this value, the probability p is perturbed with a random offset of $\pm 10\%$.

The swap improvement procedure considers in turn each vertex in the current cycle and tries to determine a better feasible solution by removing the vertex from its current position and by inserting it, or any other vertex belonging to the same cluster, in every other position of the current cycle. The L-K procedure is an adaptation of the classical Lin-Kernighan TSP improvement method, where we initially restrict the maximum number of edges involved in a single move to $k_{max} = 15$. To avoid too high computing times for the L-K procedure, we impose a maximum limit equal to $15m$ on the number of attempted k-opt moves with $k > 2$. In addition, we change the value of k_{max} by decreasing it of one unit whenever the number of attempted moves reaches its maximum limit (i.e. $15m$), and increasing it of one unit otherwise (never exceeding the initial value of 15).

In the following we present in detail the sequence of steps performed by the algorithm. A *feasible vertex set* C is given by a vertex subset of V containing one vertex for each cluster. A *feasible solution* T is given by a sequence of the vertices belonging to a feasible vertex set C. The cost of the solution is given by the sum of the costs of the edges in the sequence. We denote with SB the best known feasible solution found so far, and with S the current reference solution. In addition, we call $H(C)$ the subgraph induced by the feasible vertex set C. The following counters are used in the description of the algorithm:

- *count* = number of iterations executed with no improvement of the reference solution S;

- *count*$_p$ = number of iterations executed from the last change of the probability value.

0. Set $SB := \emptyset$, $S := \emptyset$, $count := 0$, $count_p := 0$, $C_p := 0$, $p := p_{fin} - \lfloor p_{fin}/(C_p + 2) \rfloor$.

1. Define a feasible vertex set C by randomly selecting, with uniform probability, one vertex from each cluster.

2. If $S \neq \emptyset$, then substitute, with *probability* p, each vertex v of C with the vertex $u \in S$ such that u belongs to the same cluster as v. Consider the corresponding induced subgraph $H(C)$.

3. If *count* $= MAXI$ or $S = \emptyset$, then find a TSP feasible solution on graph $H(C)$, by applying the FITSP procedure. Otherwise, the current feasible solution is defined by the visiting order of the clusters corresponding to S. Apply the L-K and swap improvement procedures, and obtain

the sequence of vertices $T = (v_1, \ldots, v_m)$. Let $T_V = \{V_{s(1)}, \ldots, V_{s(m)}\}$ be the corresponding sequence of clusters.

4. Apply the Layered Network Method (taking T_V as visiting order of the clusters) and obtain a new sequence of vertices T_{new} (this is the best sequence of vertices with the fixed cluster order).

5. If the sequence T_{new} can be improved through the L-K and/or swap improvement procedures, then update T_{new} and T_V and go to step 4.

6. If $S = \emptyset$ or the cost of S is worse than the cost of T_{new}, then:

   6.a set $S := T_{new}$ and *count* $:= 0$ (consider the current solution as the reference one);

   6.b If $SB = \emptyset$ or the cost of SB is worse than the cost of T_{new}, then set $SB := T_{new}$;

   Otherwise:

   6.c if *count* $< MAXI$ then set *count* $:=$ *count* $+ 1$ else set $S := T_{new}$ and *count* $:= 0$;

7. 7.a If a given time limit is reached, then stop the algorithm.

   7.b Set $count_p := count_p + 1$. If $count_p = MAXP$, then set $C_p := C_p + 1$, $p := p_{fin} - \lfloor p_{fin}/(C_p + 2) \rfloor$, $count_p := 0$. If $p = p_{fin}$, then $p :=$ uniformly distributed random integer in the range $[0.9p_{fin}, 1.1p_{fin}]$.

   7.c Go to step 1.

For a better understanding, in Figures 2.1 - 2.4 we show an example of the first iteration of the described heuristic on the instance $10att48$ taken from the literature. The costs of the edges are assumed to be proportional to the Euclidean distances between the corresponding extreme vertices (i.e., the cost of the edge having as extreme vertices those with coordinates (x_i, y_i) and (x_j, y_j) is given by $\lceil \sqrt{((x_i - x_j)^2 + (y_i - y_j)^2) * 10} \rceil$).

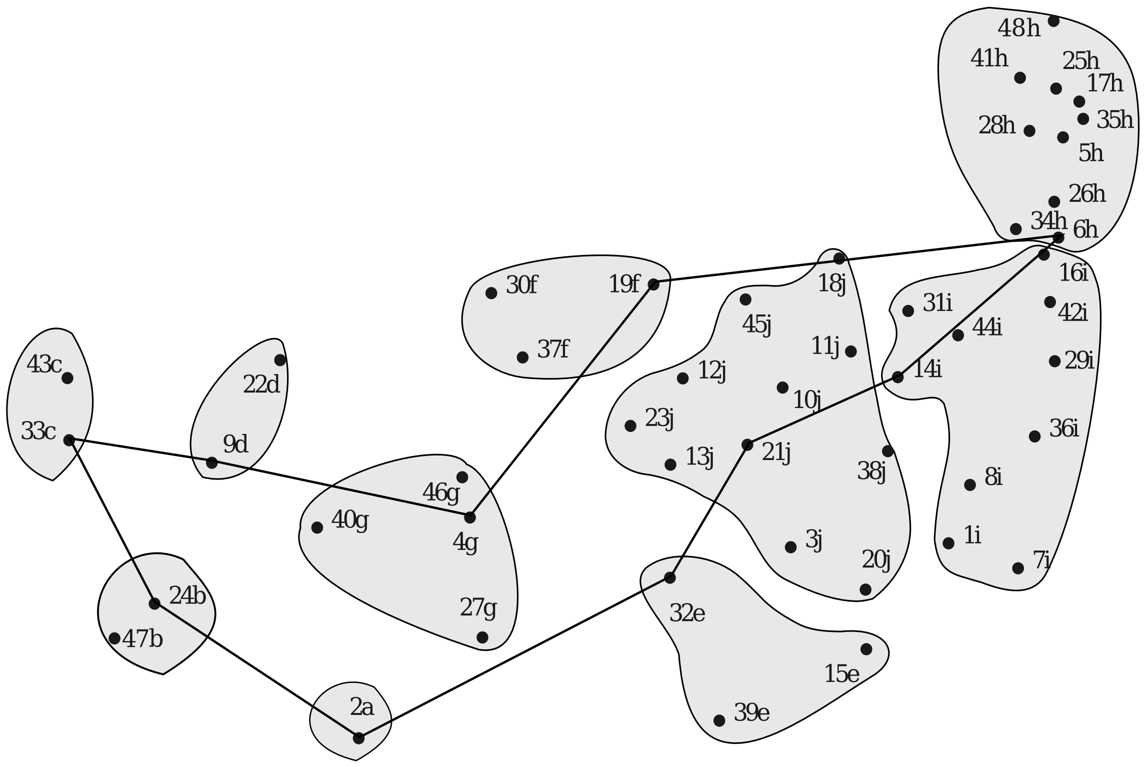


Figure 2.1: Example of steps 0-3 of the Multi-Start Algorithm: one vertex per cluster is randomly selected with uniform probability, and the sequence of vertices in the figure is obtained by applying the FITSP procedure followed by the L-K and swap procedures.

In the example we have 10 clusters and 48 vertices. The vertices are numbered from 1 to 48 and the clusters are identified with letters from a to j and represented by grey sets. In Figure 2.1 vertices are randomly selected, one for each cluster, and the sequence T is obtained by applying the FITSP procedure followed by the L-K and swap procedures. T is given by $(2a, 24b, 33c, 9d, 4g, 19f, 6h, 14i, 21j, 32e)$. The sequence of clusters T_V is given by $\{a, b, c, d, g, f, h, i, j, e\}$. The cost of the obtained solution is 5799. In Figure 2.2, the Layered Network Method is applied, taking the order of the clusters at the previous step as fixed. The new sequence of vertices

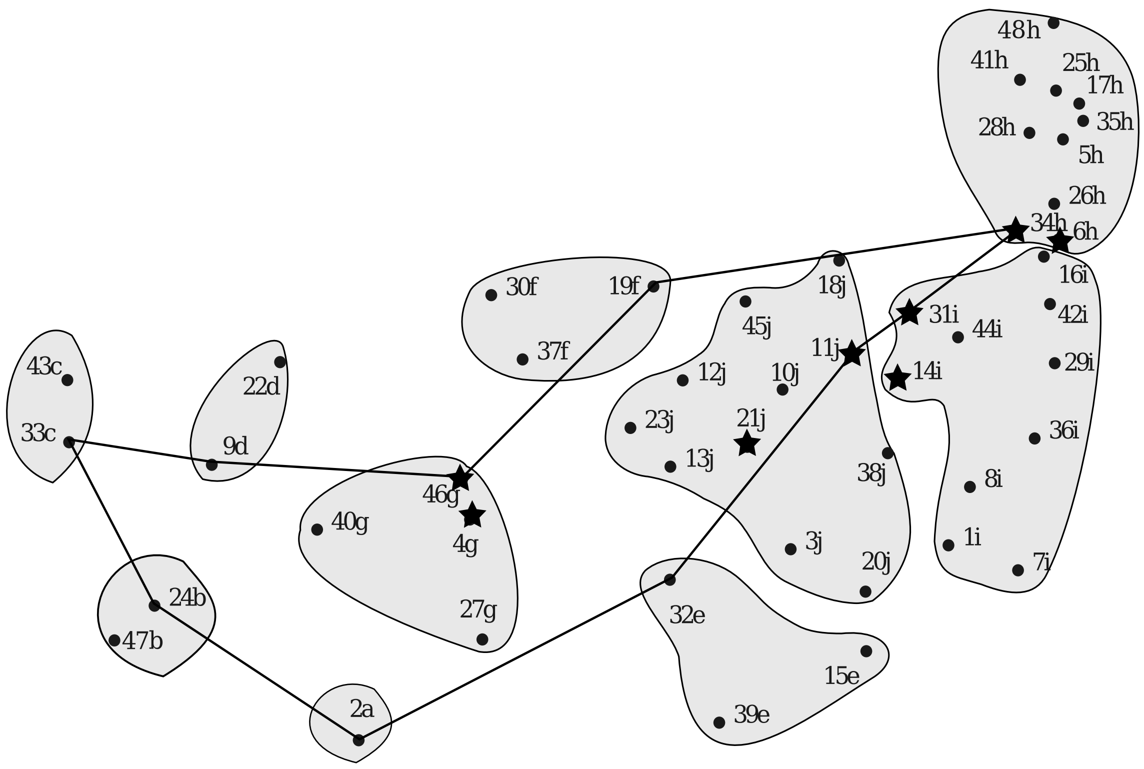


Figure 2.2: Example of step 4 of the Multi-Start Algorithm: the Layered Network Method is applied, giving the new represented sequence of vertices, where the "stars" correspond to the replacement of vertices.

T_{new} is given by $(2a, 24b, 33c, 9d, 46g, 19f, 34h, 31i, 11j, 32e)$, where vertices $\{4g, 6h, 14i, 21j\}$ are replaced by vertices $\{46g, 34h, 31i, 11j\}$, respectively. The changes with respect to the previous step are indicated with stars in the figure. The cost of the obtained solution is 5523. In Figure 2.3, the L-K and swap procedures are applied and the new sequence of vertices $T_{new} = (2a, 24b, 33c, 9d, 19f, 18j, 34h, 31i, 32e, 27g)$ is obtained. The corresponding new sequence of clusters is given by $T_V = \{a, b, c, d, f, j, h, i, e, g\}$. The cost of the obtained solution is 5426. Since the visiting order for the clusters has been changed, we repeat step 4 applying the Layered Network Method and obtaining the new sequence of vertices $\{2a, 24b, 33c, 22d, 19f, 18j, 34h, 31i, 32e, 27g\}$,

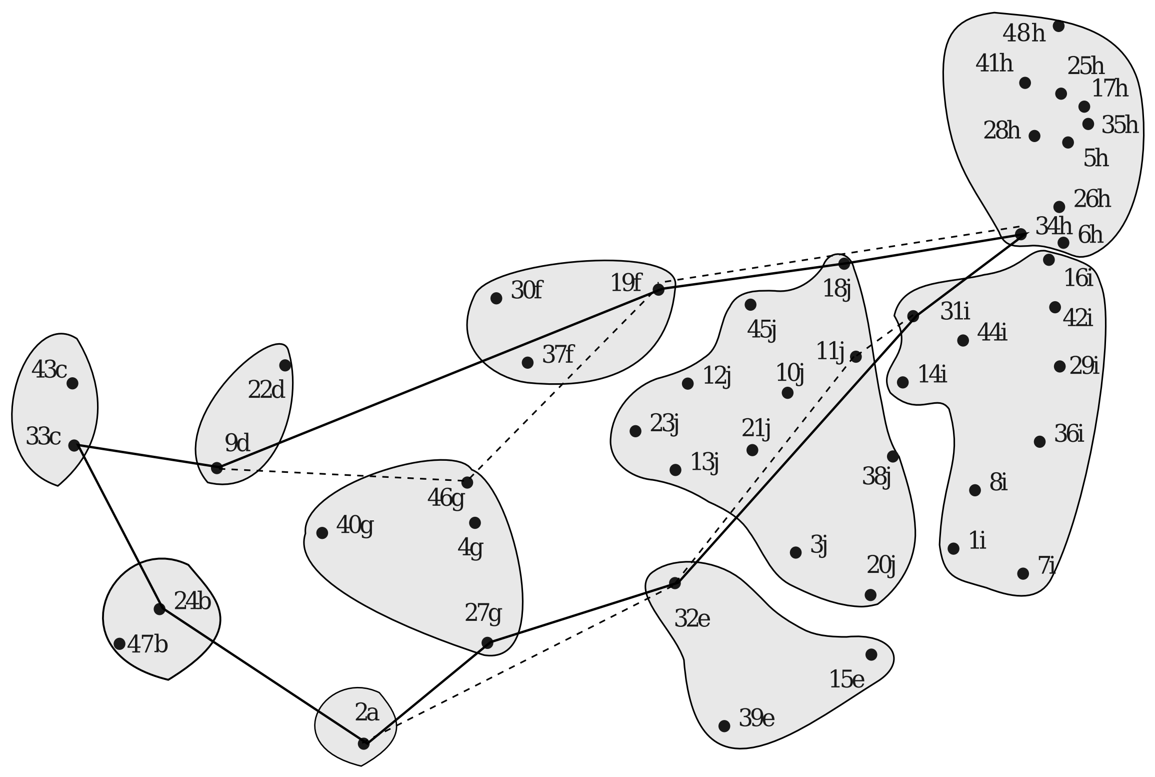


Figure 2.3: Example of step 5 of the Multi-Start Algorithm: the L-K and swap procedures are applied, and as a result the order of the clusters is changed; the dashed lines represent the previous edges.

where vertex $9d$ is replaced by vertex $22d$ (see Figure 2.4). The cost of the obtained solution is 5394, which is the optimal solution value. The L-K and swap procedures produce no further improvement, thus we store the solution in S and SB and end the first iteration of the algorithm.

Notice that steps 4 and 5 constitute a local search scheme which tries to improve the current solution T_{new} by alternating the Layered Network Method and the L-K and swap procedures. The change of neighborhood (or restart) is performed by steps 1 and 2, which lead the algorithm to reach new neighborhoods, tuned by parameter p.

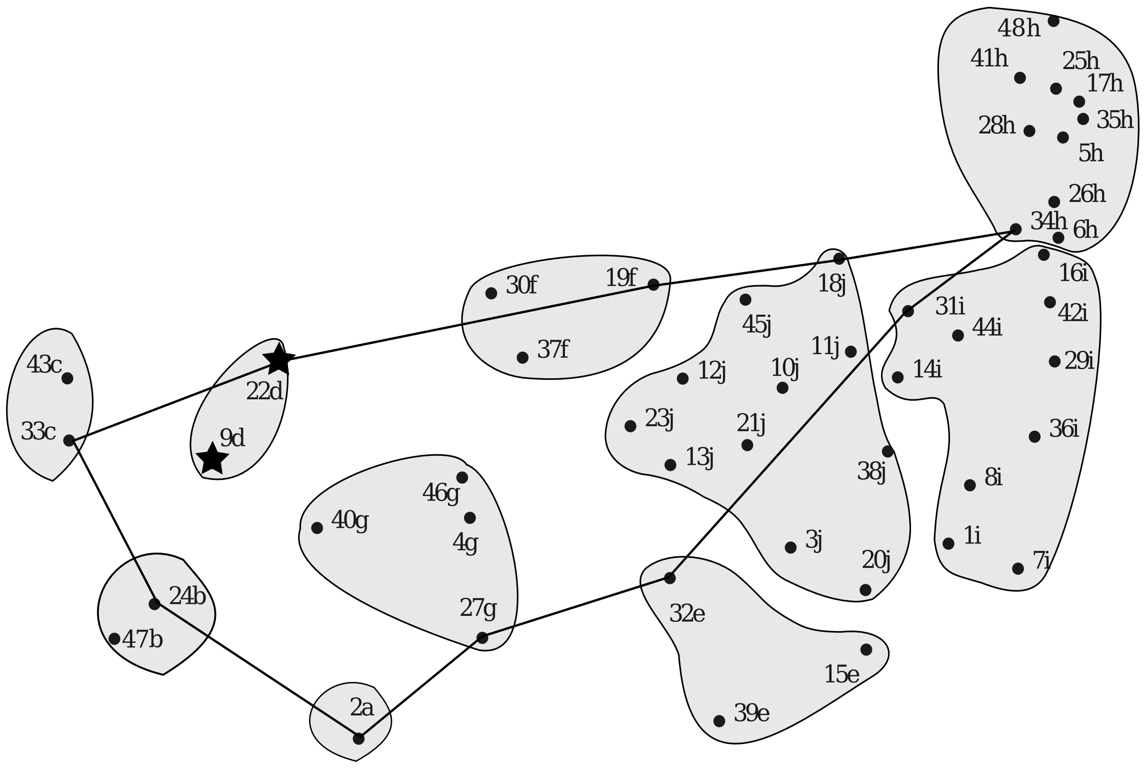


Figure 2.4: Example of steps 4-6 of the Multi-Start Algorithm: the Layered Network Method is applied; it replaces vertex $9d$ with vertex $22d$ and finds the optimal solution.

2.4 Computational Results.

The Multi-Start Algorithm *MSA* described in Section 2.3 was implemented in Java (JDK and JRE 5.0 Update 11) and tested on a *PC* Pentium(R) IV, 1 Gb RAM, 3.4Ghz.

The algorithm was tested on a set of benchmark instances obtained by the clustering procedure introduced by Fischetti et al. [13] applied to instances from the TSPLIB Library [42]. All the instances have triangular costs. For each instance, the clustering of the corresponding n vertices has been done so as to simulate geographical regions, with a number of clusters equal to $\lceil n/5 \rceil$. These 41 instances are generally used to test the performance of the algorithms for the *E-GTSP*. The names of the instances *xxnameyyy* indicate the number of clusters (xx) and the number of vertices yyy.

The results obtained by MSA in a single run, corresponding to a given random seed, are compared with the optimal solution values obtained by the Branch-and-Cut algorithm ($B\&C$) presented in [13], and with the results obtained by the following heuristics:

1. the Genetic Algorithm $mrOX$ by Silberholz and Golden [47],

2. the Reinforcing Ant Colony System $RACS$ by Pintea et al. [41],

3. the Genetic Algorithm GA by Snyder and Daskin [49],

4. the composite algorithm GI^3 by Renaud and Boctor [43],

5. the Nearest Neighbor approach NN by Noon [38],

6. the heuristics (*FST-Lagr* and *FST-root*) by Fischetti et al. [13], applied at the root node of the decision tree.

In order to perform a fair comparison on the computing times, we refer to Dongarra [7] for the evaluation of the speed of the computers used in the experiments. The computer factors are shown in Table 2.2. The columns have the following meaning:

- *Computer* describes the used computer;

- *Mflops* reports the corresponding calculated amount of Mflops per second;

- r reports the computer factor, i.e. the ratio between the Mflops of the computer and 295 (our computer Mflops);

- *Method* describes the corresponding algorithm.

Since in [41] the computer was not reported, for algorithm $RACS$ we assume a computer factor equal to 1.

Computer	Mflops	r	Method
Gateway Profile 4MX	230	0.78	GA
Sun Sparc Station LX	4.6	0.015	GI^3, NN
HP 9000/720	2.3	0.007	FST-Lagr, FST-Root, B&C
Unknow	-	1	RACS
Dell Dimension 8400	-	1	mrOX
Our	295	1	MSA

Table 2.2: Computers comparison

MSA was run with a time limit of 60 seconds. We used the following values for the parameters: final percentage probability $p_{fin} = 75$ (hence the initial percentage probability is equal to $75 - \lfloor 75/2 \rfloor = 38$), maximum number of iterations without improvement $MAXI = 500$, and maximum number of iterations without changing the probability $MAXP = 20$.

Table 2.3 compares the results of MSA with those of the best methods in the literature. For each instance, we report the percentage gap with respect to the optimal solution value and the computing time (expressed in seconds and scaled according to the computer factors given in Table 2.2) for all the methods but for *B&C* (for which we report only the computing time). Note that the results of $mrOX$ [47] and $RACS$ [41] represent, for each instance, the average of the best solutions found in 5 runs, each corresponding to a different random seed (a time limit of 10 minutes was imposed on each run of $RACS$). The results of all the other algorithms refer to a single run for each instance. In the last four rows of Table 2.3 we report, for each algorithm, the average percentage gap and the average computing time on the 36 instances tested by all the methods and on the 41 instances for the algorithms which consider the remaining 5 instances. We report as well the number of times the optimum was reached. As Table 2.3 shows, MSA finds the optimal solution for all the instances, and the computing times are always much smaller than those of the Branch-and-Cut algorithm *B&C* (and, as well, than those of the *FST-root* heuristic applied at the root node of the corresponding decision tree). The table also shows that MSA clearly outperforms the other heuristics.

Instance	Optval	MSA		$mrOX$		$RACS$	GA		GI^3		NN		$FST-lagr$		$FST-Root$		$B\&C$
		gap	time	gap	time	gap	gap	time	gap	time	gap	time	gap	time	gap	time	time
10att48	5394	0	0	0	0.4	-	0	0	-	-	-	-	0	0	0	0	0.0
10gr48	1834	0	0	0	0.3	-	0	0.4	-	-	-	-	0	0	0	0	0.0
10hk48	6386	0	0	0	0.3	-	0	0.2	-	-	-	-	0	0	0	0	0.0
11eil51	174	0	0	0	0.3	0	0	0.1	0	0	0	0	0	0	0	0	0.0
12brazil58	15332	0	0	0	0.8	-	0	0.2	-	-	-	-	0	0	0	0	0.0
14st70	316	0	0	0	0.4	0	0	0.2	0	0	0	0	0	0	0	0.1	0.1
16eil76	209	0	0	0	0.4	0	0	0.2	0	0	0	0	0	0	0	0.1	0.1
16pr76	64925	0	0	0	0.5	0	0	0.2	0	0	0	0	0	0	0	0.1	0.1
20rat99	497	0	0	0	0.5	0	0	0.5	0	0.1	0	0.1	0	0	0	0.4	0.4
20kroA100	9711	0	0	0	0.6	0	0	0.3	0	0.1	0	0.1	0	0	0	0.1	0.1
20kroB100	10328	0	0	0	0.6	0	0	0.3	0	0.1	0	0	0	0	0	0.2	0.2
20kroC100	9554	0	0	0	0.6	0	0	0.2	0	0.1	0	0.1	0	0	0	0.1	0.1
20kroD100	9450	0	0	0	0.7	0	0	0.3	0	0.1	0	0.1	0	0	0	0.1	0.1
20kroE100	9523	0	0	0	0.6	0	0	0.6	0	0.1	0	0	0	0	0	0.1	0.1
20rd100	3650	0	0	0	0.5	0	0	0.2	0.08	0.1	0.08	0.1	0.08	0	0	0.1	0.1
21eil101	249	0	0	0	0.5	0	0	0.2	0.4	0.1	0.4	0	0	0	0	0.2	0.2
21lin105	8213	0	0	0	0.6	0	0	0.2	0	0.2	0	0.1	0	0	0	0.1	0.1
22pr107	27898	0	0	0	0.5	0	0	0.3	0	0.1	0	0.1	0	0	0	0.1	0.1
24gr120	2769	0	0	0	0.7	-	0	0.4	-	-	-	-	1.99	0	0	0.3	0.3
25pr124	36605	0	0	0	0.7	0	0	0.5	0.43	0.2	0	0.2	0	0	0	0.2	0.2
26bier127	72418	0	0	0	0.8	0	0	0.4	5.55	0.5	9.68	0.1	0	0.1	0	0.2	0.2
28pr136	42750	0	0	0	0.8	0	0	0.4	1.28	0.2	5.54	0.1	0.82	0.1	0	0.3	0.3
29pr144	45886	0	0	0	1.0	0	0	0.2	0	0.2	0	0.2	0	0	0	0.1	0.1
30kroA150	11018	0	0	0	1.0	0	0	1.0	0	0.3	0	0.3	0	0.1	0	0.7	0.7
30kroB150	12196	0	0	0	1.0	0	0	0.8	0	0.2	0	0.3	0	0.1	0	0.4	0.4
31pr152	51576	0	0	0	1.0	0	0	1.2	0.47	0.3	1.8	0.2	0	0.1	0	0.4	0.7
32u159	22664	0	0	0	1.0	0.01	0	0.5	2.6	0.3	2.79	0.4	0	0.1	0	1.0	1.0
39rat195	854	0	0.1	0	1.4	0	0	0.5	0	0.6	1.29	1.3	1.87	0.1	0	1.7	1.7
40d198	10557	0	0	0	1.6	0.01	0	0.9	0.6	0.9	0.6	1.8	0.48	0.1	0	5.3	5.3
40kroA200	13406	0	0	0	1.7	0.01	0	2.1	0	0.4	5.25	0.8	0	0.1	0	1.3	1.3
40kroB200	13111	0	0	0.05	1.6	0	0	1.1	0	0.5	0	2.0	0.05	0.1	0	1.9	1.9
45ts225	68340	0	2.1	0.14	1.7	0.02	0	1.9	0.61	1.3	0	1.8	0.09	0.1	0.09	9.1	265.1
46pr226	64007	0	0	0	1.5	0.03	0	0.8	0	0.4	2.17	1.0	0	0.1	0	0.7	0.7
53gil262	1013	0	3.5	0.45	3.6	0.22	0.79	1.5	5.03	1.7	1.88	1.8	3.75	0.1	0.89	10.1	46.4
53pr264	29549	0	0	0	2.4	0	0	1.0	0.36	1.0	5.73	2.2	0.33	0.2	0	2.4	2.4
60pr299	22615	0	0.7	0.05	4.6	0.24	0.02	4.8	2.23	0.1	2.01	4.2	0	0.2	0	5.7	5.7
64lin318	20765	0	0	0	8.1	0.12	0	2.7	4.59	3.1	4.92	4.8	0.36	0.4	0.36	5.9	11.7
80rd400	6361	0	0.3	0.58	14.6	0.87	1.37	2.7	1.23	6.1	3.98	17.1	3.16	0.4	2.97	35.2	49.2
84fl417	9651	0	0	0.04	8.2	0.57	0.07	1.9	0.48	6.4	1.07	20.1	0.13	0.5	0	117.0	117.0
88pr439	60099	0	1.9	0	19.1	0.79	0.23	7.1	3.52	9.2	4.02	18.6	1.42	1.0	0	37.9	38.0
89pcb442	21657	0	0.8	0.01	23.4	0.69	1.31	7.9	5.91	8.5	0.22	12.6	4.22	0.6	0.29	37.5	411.4
Average (36)		0	0.26	0.04	3.00	0.10	0.10	1.27	0.98	1.21	1.48	2.57	0.46	0.13	0.13	7.69	26.75
# Opt (36)		36		29		24	30		19		18		23		31		36
Average (41)		0	0.23	0.03	2.70		0.09	1.14					0.46	0.11	0.11	6.76	23.5
# Opt (41)		41		34			35						27		36		41

Table 2.3: Comparison of the algorithms on the small instances.

In Table 2.4 we present a comparison on a set of new larger instances proposed by Silberholz and Golden in [47]. These instances are again obtained by applying the clustering procedure introduced by Fischetti et al. [13] to instances of the TSPLIB Library (Reinelt [42]). In order to perform a fair comparison with the results obtained by algorithm $mrOX$ [47], which correspond to the averages of the best solutions found in 5 runs, for each instance we have run MSA with 5 different random seeds. The corresponding average results (best solution values and computing times) are reported in Table 2.4. In addition, as done for the smaller instances, we report the results obtained by MSA in a single run, since this is, according to our computational experiments, the most effective way to execute MSA. In Table 2.4 the columns have, for each instance, the following meaning:

1. instance name,
2. the average of the best solution values obtained by algorithm $mrOX$ [47] (computed over 5 runs),
3. the corresponding average computing time expressed in seconds (computed over 5 runs),
4. the best solution value obtained by algorithm MSA in a single run, with the same random seed used for the smaller instances considered in Table 3,
5. the corresponding computing time expressed in seconds,
6. the average of the best solution values obtained by algorithm MSA (computed over 5 runs),
7. the corresponding average computing time expressed in seconds (computed over 5 runs).

For each instance and for each run, MSA was run with a time limit of 600 seconds. We represent in bold the best solution value for each instance. In the last row we present the average values over the 13 instances.

Instance	*mrOX* (5 runs)		*MSA* (single run)		*MSA* (5 runs)	
	avg. value	avg. time	value	time	avg. value	avg. time
99D493	20117.2	35.72	**20023**	226.10	**20023.0**	220.85
107ATT532	13510.8	31.70	**13464**	4.76	**13464.0**	2.81
107SI535	13513.2	26.35	**13502**	0.39	**13502.0**	2.13
113PA561	1053.6	21.08	**1038**	6.51	**1038.0**	2.81
115RAT575	2414.8	48.48	**2388**	27.28	**2388.0**	53.57
131P654	27508.2	32.67	**27428**	6.03	**27428.0**	3.59
132D657	22599.0	132.24	**22498**	367.97	22502.4	296.44
145U724	17370.6	161.82	**17271**	76.97	**17271.0**	133.18
157RAT783	3300.2	152.15	3266	573.37	**3265.4**	311.38
201PR1002	114582.2	464.36	**114311**	70.54	114316.0	86.67
207SI1032	22388.8	242.37	22311	115.70	**22310.4**	311.53
212U1060	108390.4	594.64	**105957**	566.03	106026.2	251.24
217VM1084	131884.6	562.04	**130703**	101.38	**130703.0**	198.50
Average	38356.4	192.74	38012.3	164.85	38018.3	144.21

Table 2.4: Comparison on a new data set of instances proposed in [47]

Table 2.4 shows that algorithm *MSA* outperforms algorithm *mrOX*, since it always obtaines, in comparable computing times, better solutions (by considering both the values found in a single run and the average values found in 5 runs). It is worth to note that the behavior of *MSA* is quite stable with respect to the random seed considered for its execution. Indeed, the average solution values (computed over 5 runs) are very close to the values found in the single runs: for 8 instances (over 13) the values coincide, for 3 instances the single run produces better solutions, and for 2 instances the averages over 5 runs are better (for these 2 instances the best solution values found were 3265 for instance $157RAT783$ and 22306 for instance $207SI1032$); in addition, the global average values are 38018.3 for the averages over 5 runs and 38012.3 for the single runs.

A more detailed experimental analysis on the execution of a single run of *MSA* showed that, for the 13 larger instances considered in Table 2.4, the number of iterations (Steps 1 to 7 of the algorithm described in Section 2.3) was on average 22076 (with a minimum of 39 and a maximum of 101564), while the number of applications of the $FITSP$ procedure (see Step 3) was on average 23 (with a minimum of 0 and a maximum of 102). We observed as well that, as expected, most of the computing time of MSA (about 95%) was spent in the execution of the L-K and swap improvement procedures (see Step 3).

In order to investigate the effects of the probabilistic alteration of the vertices chosen in the initialization phase (see Step 2) and of the rule used to update the value of the probability p (see Step 7.b), we executed a single run of MSA, on the 13 larger instances, by setting $p = 0$ (i.e. with no probabilistic alteration) and $p = 75\%$ (i.e. with a fixed probability equal to its maximum value). The global averages of the solution values and of the corresponding computing times were, respectively, 38150.6 and 213 seconds for $p = 0$, and 38016.9 and 224 seconds for $p = 75\%$. In both cases, the average results are worse than those obtained by executing a single run of MSA with the original parameter setting (38012.3 and 165 seconds for the averages of the solution values and of the computing times, respectively). Indeed, when setting $p = 0$ the algorithm performs a completely random selection of the vertices, and when setting $p = 75\%$ it keeps as fixed several vertices of the reference solution that at the beginning of the process may not be of good quality.

2.5 Conclusions.

We developed a multi-start heuristic for the *E-GTSP*, which starts with a randomly chosen set of vertices, one for each cluster, and applies a Farthest Insertion TSP procedure followed by a Lin-Kernighan and a swap improvement procedures to obtain a good visiting order for the clusters. The algorithm then takes this order as fixed and applies the Layered Network Method to compute the corresponding best cycle. The Lin-Kernighan and swap procedures are applied again, followed by the Layered Network Method, until no further improvement can be obtained. The algorithm is then repeated by considering a new randomly chosen set of vertices, where, with probability p, each vertex is replaced by the vertex belonging to the corresponding cluster in the reference solution.

We compared the proposed method with the best state-of-the-art algorithms on a set of benchmark instances from the literature, and showed the effectiveness of the approach. It turned out that, on the set of smaller instances (with up to 442 vertices), the proposed algorithm finds always the optimal solution in less than four seconds, and clearly outperforms the most effective heuristics. For the set of larger instances (with up to 1084 vertices) the proposed algorithm always improves, in comparable computing times, the solution values obtained by the genetic algorithm recently presented by Silberholz and Golden [47].

The proposed heuristic can be easily extended to the case of the *AGTSP*, by adapting the Farthest Insertion TSP procedure and the Lin-Kernighan and swap improvement procedures. Future work will be devoted to investigate the general case of *GTSP* (where more than one vertex for each cluster can be visited), and the extension to the variant of the problem where clusters may intersect.

Chapter 3

Bin Packing Problem with Conflicts.

3.1 Introduction.

In the *Bin Packing Problem with Conflicts* (BPPC), we are given a set $V = \{1, 2, \ldots, n\}$ of items, each item i having a non-negative weight w_i, and an infinite number of identical bins of weight capacity C. We are also given a *conflict graph* $G = (V, E)$, where E is a set of edges such that $(i, j) \in E$ when items i and j are in conflict. Items in conflict cannot be assigned to the same bin. The aim of the BPPC is to assign all items to the minimum number of bins, while ensuring that the total weight of the items assigned to a bin does not exceed the bin weight capacity and that no bin contains items in conflict.

The BPPC is important because of the high number of real-world applications, and because it generalizes other important problems in combinatorial optimization. Some BPPC real-world applications include examination scheduling (see [26]), the assignment of processes to processors and the load balancing of tasks in parallel computing (see [21]). Other applications concern particular delivery problems, such as food distribution, where some items cannot be placed in the same vehicle (see [6]).

The BPPC is NP-hard, since it generalizes both the *Bin Packing Problem* (BPP) and the *Vertex Coloring Problem* (VCP). The BPP (see, e.g., [33]) is a special case of the BPPC where no item is in conflict with another, i.e., $E = \emptyset$. In the VCP each vertex of the conflict graph has to be assigned a color, such that no two adjacent vertices share the same color and the number of colors used is minimum (see, e.g., [23] and [32]). The VCP is a special case of the BPPC where all items weights take value 0.

The BPPC can be modelled by introducing two sets of binary variables: y_h, taking value 1 if bin h is used, 0 otherwise ($h = 1, 2, \dots, n$); x_{ih}, taking value 1 if item i is assigned to bin h, 0 otherwise ($i = 1, 2, \dots, n$, $h = 1, 2, \dots, n$). We obtain the following Integer Linear Programming model.

$$\min \sum_{h=1}^{n} y_h \tag{3.1}$$

$$\sum_{h=1}^{n} x_{ih} = 1 \qquad i = 1, 2, \dots, n \tag{3.2}$$

$$\sum_{i=1}^{n} w_i x_{ih} \leq C y_h \qquad h = 1, 2, \dots, n \tag{3.3}$$

$$x_{ih} + x_{jh} \leq y_h \qquad (i, j) \in E,\ h = 1, 2, \dots, n \tag{3.4}$$

$$y_h \in \{0, 1\} \qquad h = 1, 2, \dots, n \tag{3.5}$$

$$x_{ih} \in \{0, 1\} \qquad i = 1, 2, \dots, n,\ h = 1, 2, \dots, n. \tag{3.6}$$

Constraints (3.2) require that each item is assigned to a bin. Constraint (3.3) are the classical capacity constraints of BPP and (3.4) are the classical conflict constraints of the VCP. Model (3.1)–(3.6) generalizes the well known descriptive model of the BPP and was introduced by [17], although with a weaker version of (3.4). Note that conflict constraints are here reported as edge constraints, but can be strengthened to clique constraints, as discussed, e.g., in [30]. Other valid inequalities may be introduced to strengthen the model, although its linear relaxation remains weak (see, e.g., [33]).

The BPPC was addressed by [22] and [21], who derived polynomial time approximation schemes for special classes of conflict graphs. The most relevant work from a computational point of view was provided by [17], who surveyed previous results in the literature, presented new lower and upper bounds, and introduced benchmark instances. [18] considered a special case of the BPPC, the so-called *Bounded Vertex Coloring*, where all weights are equal to 1, and proposed upper and lower bounds, as well as complexity results. [2] considered the complexity of the previous problem for different conflict graph classes. [1] considered the mutual exclusion scheduling, where unit-time tasks have to be scheduled on m processors, subject to the constraint that conflicting tasks must run in disjoint time intervals. A problem related to the BPPC, where a number of conflicting examinations have to be scheduled in the smallest number of periods, under capacity restrictions, was addressed through greedy heuristic procedures by [5].

In the remaining of the text we will use the definition of *extended conflict graph* $G' = (V, E')$, where $E' = E \bigcup \{(i, j) : i, j \in V \text{ and } w_i + w_j > C\}$ as the set of edges containing the given input conflicts and the conflicts imposed for those items whose weight sum is greater than the bin capacity.

This work provides new lower and upper bounds for the BPPC and to combine them into an exact approach. In Section 3.2 we survey combinatorial lower bounds from the literature and introduce new lower bounding procedures. In Section 3.3 we present several upper bounds, including a metaheuristic approach making use of an inner Tabu Search algorithm. In Section 3.4 we present an exact approach, based on a classical Set Covering formulation and solved through column generation and Branch-and-Price. In Section 3.5 we evaluate all the algorithms by means of extensive computational tests on benchmark instances.

3.2 Combinatorial lower bounds.

We first note that any lower bound for the BPP is a valid lower bound also for the BPPC. Analogously, any lower bound for the VCP is also a lower bound for the BPPC. In Section 3.2.1 we briefly describe lower bounds from the BPP, the VCP and the BPPC literature, while in Section 3.2.2 we present new lower bounds for the BPPC. In the following, we denote by L_x both the procedure used to produce a lower bound, and the lower bound value itself (where $_x$ changes accordingly to the procedure). Similarly, we denote by U_x both the procedure used to produce an upper bound, and the upper bound value itself.

3.2.1 Lower bounds from the literature.

Many lower bounding techniques have been proposed in the BPP literature. The simplest technique corresponds to the so-called continuous lower bound (denoted as L^0_{BPP} in the following), computed as the rounding up of the sum of the weights divided by the bin capacity

$$L^0_{BPP} = \left\lceil \sum_{i=1}^{n} \frac{w_i}{C} \right\rceil . \tag{3.7}$$

Other effective lower bounding strategies were proposed by [10] through *Dual Feasible Functions*. These functions modify the item weights, while ensuring that, if an item subset fits in one bin, then it also fits in one bin after the weight modification. Then a classical BPP algorithm, as L^0_{BPP}, is applied on the modified weights to obtain a valid lower bound. In [10] three parametric functions were proposed. We denote as L^{DFF}_{BPP} the maximum lower bound value produced by these functions. The corresponding time complexity is $O(n)$ for items sorted by size.

Also the VCP literature has produced lower bounding techniques. A simple idea consists of determining a maximum clique, whose cardinality naturally gives a valid lower bound value for the VCP. [17] developed a valid BPPC bound by heuristically computing a maximum clique on the extended conflict graph G' through the greedy algorithm of [24]. This heuristic method initializes the clique with the vertex of maximum cardinality. Then it enlarges the clique by iteratively adding new vertices: at each iteration it considers the remaining vertices according to a non-increasing degree order, and adds to the clique the first vertex which can fit. Such algorithm can be implemented in $O(m + n)$ time for vertices sorted by non-increasing degree. This simple lower bound is denoted as L_{MC} in the following.

One can note that, when applied to the BPPC, any BPP lower bound is arbitrarily bad (it is sufficient to consider an instance with bin capacity

1 and items weights $\varepsilon << 1$), and also any VCP lower bound is arbitrarily bad (by considering an instance with $E = \emptyset$). It is thus more convenient to focus on bounds tailored to the BPPC structure. One of these bounds was proposed by [17], and is denoted as *constrained packing lower bound* (L_{CP} in the following). In L_{CP} a maximal clique K of graph G' is computed by means of Johnson's algorithm as in L_{MC}. Then a bin is initialized for each item in the clique, and the items in $V \setminus K$ are assigned to these bins (possibly in a fractional way), by solving a transportation problem that takes into consideration both the weights and the given conflicts. All the remaining items (or fractions of items) than cannot fit in the $|K|$ initialized bins are stored in a new set V_1, on which L^0_{BPP} is applied. A valid lower bound is finally obtained by computing

$$L_{CP} = |K| + L^0_{BPP}(V_1). \tag{3.8}$$

Note that by construction $L_{CP} \geq L_{MC}$ and $L_{CP} \geq L^0_{BPP}$.

3.2.2 Clique-based lower bounds.

Let us discuss the lower bounds based on the computation of a maximal clique. [17] propose to compute L_{MC} by finding a maximal clique on graph G'. Even if any maximal clique of G (the graph that considers only input conflicts) is by construction included in a maximal clique of G', computing a maximal clique of G' by means of Johnson's algorithm can lead to disappointing results. Indeed, the drawback of this strategy is that vertices of high weight have systematically high degree in G'. Hence, they are usually among the first to be included in the clique, even when they do not belong to large cliques of G'.

We propose the following improvement to the computation of L_{MC}: 1) compute a maximal clique K on G by means of Johnson's algorithm; 2) add to E all edges (i, j) such that $w_i + w_j > C$, thus obtaining graph G'; 3) enlarge K by selecting vertices in $V \setminus K$ according to a non-increasing degree order. The clique K' we obtain in this way is a maximal clique of G' and contains K. The corresponding lower bound is $L^{imp}_{MC} = |K'|$. On average this strategy leads to larger cliques than those obtained by applying the Johnson's algorithm directly on graph G'.

Bound L_{CP} may be improved by considering the way the maximal clique K is computed. We obtain a new bound, denoted as L_{CP}^{imp}, by determining K as in L_{MC}^{imp} instead of L_{MC}. This generally leads to better lower bounds.

3.2.3 A surrogate relaxation.

Let us study a possible surrogate relaxation of constraints (3.4). We sum the original $n|E|$ constraints over the edges, with all multipliers equal to 1, and we obtain the n constraints

$$\sum_{i=1}^{n} \delta_i x_{ih} \leq |E| y_h \quad h = 1, 2, \ldots, n, \tag{3.9}$$

where δ_i denotes the degree of vertex i in G'. Model (3.1) – (3.3), (3.5), (3.6) and (3.9) may be seen as an instance of the *Two-Vector Packing Problem* (TVPP). The TVPP is a generalization of the BPP in which items have two weight dimensions (e.g., weight and length) and bins have two corresponding capacities.

Clearly any lower bound for this TVPP is a valid lower bound for the BPPC. In particular, we use the lower bounding procedure L_H by [4], which runs in $O(n^4 \log n)$ time. In L_H the items are first partitioned into three subsets $N1$, $N2$ and $N3$. Items in $N1$ and $N2$ are pair-wise in conflict, while $N3$ contains all the remaining items. Lower bounds are computed separately for $N1$ and $N2$ through different techniques and then summed to obtain the overall lower bound. Two partitions are attempted and the highest bound is finally returned. (Note that the only TVPP lower bound able to computationally outperform L_H is the column generation procedure proposed in [4].)

We define L_{TVPP} the lower bound obtained by: 1) relaxing in a surrogate way the BPPC constraints, as described above, so as to obtain a TVPP instance; 2) applying the TVPP procedure L_H to the new instance to get a valid BPPC bound. The corresponding time complexity is $O(n^4 \log n)$.

3.2.4 A matching-based lower bound.

Consider a subset $S \subseteq V$ of items with the property that at most two items in S can fit in a single bin. The minimum number of bins required to pack all the items in S can be computed, in polynomial time, by matching as many items as possible, and then assigning one bin to each matched pair and one bin to each unmatched item. In other words, the optimal solution value corresponds to the cardinality of a maximum cardinality matching M on the complement graph $\overline{G}_S$ of G_S, induced by S on G, plus the number of unmatched items. Of course this results in a valid lower bound for the complete set of items $V \supseteq S$. In this way we obtain the lower bound $L_{match} = |M| + (|S| - 2|M|) = |S| - |M|$.

However, the choice of subset S is far from trivial. First, we want it to be inclusion maximal with respect to the required property (i.e., at most two items in S can fit in one bin, but there exist at least three items that fit in one bin for any subset $S \bigcup \{j\}$, for $j \in V \setminus S$). We do not require S to be the maximum cardinality subset among those having the required property. Indeed, computational evidence and straightforward examples show that there is no guarantee that a maximum cardinality subset S would lead to the largest value of L_{match} (which depends as well on the cardinality of matching M).

We attempt several computations of S through the greedy procedure defined in Algorithm 2. The procedure tries to combine together in S items of large weight and items which are included in a maximal clique K, computed as for L_{MC}^{imp}. This is done by defining a parameter β such that $0 \leq \beta \leq C/2$ and there exists at least one item having weight equal to β. Then, for each value of β a corresponding subset S_β is first initialized by items in K with weight at least equal to β, and then filled with items from $V \setminus S_\beta$ being in conflict with all pairs of items in S_β. These items are considered according to a non-increasing weight ordering. For each subset, Algorithm 2 computes a valid lower bound value, and finally returns the maximum value found.

At Step 7 of Algorithm 2 we carry out the possible inclusion of a new item in S_β by means of procedure CHECK INCOMPATIBILITY by [4], running in

Algorithm 2 A Matching-based lower bound

```
1: $L_{match} := 0$;
2: Compute a maximal clique $K$ as for $L_{MC}^{imp}$;
3: Order the items by non-increasing weight;
4: for all distinct $\beta$ such that $0 \leq \beta \leq C/2$ and $\exists i \in V : w_i = \beta$ do
5:     $S_\beta := \{i \in K : \ w_i \geq \beta\}$
6:     for all $i \in V \setminus S_\beta$ do
7:         if $S_\beta \cup \{i\}$ contains no subset of 3 items which can fit into one bin then
8:             $S_\beta := S_\beta \cup \{i\}$
9:         end if
10:    end for
11:    if $|S_\beta| > L_{match}$ then
12:        Compute a maximum matching $M_\beta$ of $S_\beta$ through Edmonds' algorithm;
13:        if $L_{match} < |S_\beta| - |M_\beta|$ then
14:            $L_{match} := |S_\beta| - |M_\beta|$
15:        end if
16:    end if
17: end for
18: return $L_{match}$
```

$O(|S_\beta| log S_\beta)$ time. We compute the maximum cardinality matching at Step 12 in $O(n^3)$ time, by means of the implementation of the Edmonds' algorithm [8] proposed by Gabow [15]. Before invoking the matching algorithm, we check that subset S_β is not equal to the previous one. Since the number of distinct β values is bounded by n, the overall complexity of L_{match} is $O(n^4)$.

To see that bound L_{match} is not dominated by any of the previous lower bounds, consider the simple instance given in Figure 3.1. Each edge represents a conflict, the values reported inside the vertices denote the corresponding items weights, and suppose $C = 10$. If we relax the conflict constraints (3.4) we obtain a BPP instance having optimal solution of value 2. If instead we relax the capacity constraints (3.3) we obtain a VCP instance having optimal solution of value 3. Similarly one can check that $L_{TVPP} = L_{CP} = L_{CP}^{imp} = 3$. However, the minimum number of bins needed to pack all the items of this example is 4. When $\beta = 5$, Algorithm 2 defines $S_\beta = V$, $|M_\beta| = 2$ and $L_{match} = 6 - 2 = 4$.

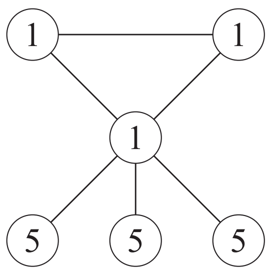


Figure 3.1: A simple BPPC example where L_{match} outperforms the other lower bounds. Let $C = 10$. Edges and numbers represent, respectively, conflicts and weights of the items.

3.3 Upper bounds.

We first adapt to the BPPC classical heuristic algorithms proposed for the BPP ([24]). Let us consider the items ordered according to non-increasing weight. We obtain:

- First-Fit Decreasing with Conflicts (U_{FF}, proposed by [17]), adaptation of the BPP First-Fit Decreasing (FFD) algorithm: consider each item in turn and assign it to the first bin with sufficient residual capacity and for which there is no conflict with the already assigned items;

- Best-Fit Decreasing with Conflicts (U_{BF}): as in U_{FF}, but the items are assigned to the feasible bin for which the resulting weight sum is maximum;

- Worst-Fit Decreasing with Conflicts (U_{WF}): as in U_{BF}, but the resulting weight sum must be minimum.

We obtain an improved class of heuristic algorithms by considering the *surrogate weights*

$$w_i^s = \alpha \frac{w_i}{\overline{w}} + (1-\alpha)\frac{\delta_i}{\overline{\delta}} \tag{3.10}$$

for $i = 1, 2, \dots, n$, where α is a given parameter (with $0 \leq \alpha \leq 1$), δ_i is the degree of i in G', and $\overline{w}$ and $\overline{\delta}$ are, respectively, the average weight and degree of the vertices. Now we can find new heuristic solutions by ordering the items according to non-increasing w_i^s values and executing the three previous heuristic techniques. In our computational experiments, we set $\alpha = 0, 0.1, \ldots, 1$, running 11 times each algorithm. We define $U_{FF(\alpha)}$, $U_{BF(\alpha)}$ and $U_{WF(\alpha)}$ the new heuristic procedures. Both the adaptations of the classical BPP heuristic algorithms and the generalizations obtained by using the surrogate weights run in $O(n^2)$ time.

We finally include in our initial set of heuristic algorithms procedure $H6$, which is the most performing technique among the ones proposed by [17]. This algorithm initially partitions the items in the set C of conflicting items (with at least one conflict in G') and the set $\overline{C}$ of non-conflicting items. Then it computes a maximal clique K among the items in C on the extended conflict graph G'. For each item i of the clique K, the algorithm computes a maximal stable set H_i containing i in the subgraph of G' induced by $(C \setminus K) \cup \{i\}$ (H_i can be computed as a maximal clique in the complement of the induced subgraph). The Bin Packing instance corresponding to the items in H_i is solved through FFD, and the bin containing item i is fixed in the solution and the corresponding items removed from C. The process is reiterated until all items in C are assigned. Finally, all the items in $\overline{C}$ are assigned to the open (and possibly new) bins through FFD.

We propose the following improvements to the above heuristic idea, and call the resulting algorithm $H6^{imp}$: i) compute a clique K' in C through the improved procedure described in Section 3.2.2; ii) assign each item i of clique K' to a new bin and, instead of using FFD on the maximal set H_i, fill the bin heuristically by considering the items in C according to the non-decreasing weight ordering, thus reducing the overall computing time of the algorithm; iii) assign the items in $\overline{C}$ to the current (and possibly new) bins through the Best-Fit Decreasing algorithm. Both algorithms $H6$ and $H6^{imp}$ run in $O(n^2 \log n)$ time (see [17] for further details).

If the best solution value found by the above heuristic algorithms is not equal to the lower bound of the previous section, then we invoke the metaheuristic procedure to be described in the following section.

3.3.1 Population heuristic.

To find high-quality solutions on difficult instances, we propose a population based metaheuristic algorithm, called Population Heuristic in the following. The Population Heuristic is initialized with a pool of solutions obtained by the heuristic procedures described in Section 3.3. New solutions are then produced through a tailored crossover operator. Each solution is given as input to a Tabu Search algorithm. The Tabu Search technique (described in Section 3.3.1) produces an intensive local search on each input solution, and is the core of the algorithm. Whereas the crossover operator (described in Section 3.3.1) is aimed at diversifying the search among different portions of the solution space. This kind of approach is also known in the literature as *memetic algorithm* (see, e.g., [37]).

Tabu Search algorithm.

Our Tabu Search procedure moves among *partial feasible solutions*, i.e., solutions in which each bin satisfies capacity and conflict constraints, but not all the items are assigned to a bin. This idea is based on the *impasse class neighborhood*, proposed by [36] for the VCP, to improve a partial feasible solution of value k to a complete feasible solution of the same value. An effective Tabu Search procedure, based on this neighborhood, was proposed by [31] for the Vertex Coloring Problem. In our case the input parameter k represents the target number of bins to be used to pack all the items.

More formally, a partial feasible solution $\mathcal{S}$ is defined as a partition of the items set V in $k+1$ bins $\{V_1, \ldots, V_k, V_{k+1}\}$, in which all bins, but possibly the last one, are feasible. The aim of the neighborhood structure, and of our Tabu Search procedure, is to move all the items currently in bin $k+1$ to the k previous bins, so as to produce a feasible solution of value k.

In our search process, we evaluate each solution $\mathcal{S}$ through a *score function*

$f(\mathcal{S})$, to be minimized. Such score is defined as

$$f(\mathcal{S}) = \sum_{i \in V_{k+1}} \max \{w_i^s, \epsilon\} \tag{3.11}$$

and is based on the sum of the surrogate weights (introduced in Equation (3.10)) of the items not assigned to feasible bins. This score depends on parameter α and reduces to the cumulate weight or cumulate degree of the unassigned items, when $\alpha = 1$ or $\alpha = 0$, respectively. The parameter $\epsilon << 1$ is used in case some items have surrogate weight equal to 0, to avoid solutions $\mathcal{S}$ with $f(\mathcal{S}) = 0$ and some items still unassigned to feasible bins. According to our computational experience, the best results are obtained by setting $\alpha = 0$ for graphs having density $\delta > 0.4$, and $\alpha = 1$ otherwise.

To minimize the current score, we search the solution space by using a sequence of moves. A *move* from a solution $\mathcal{S}$ to a new solution $\mathcal{S}'$ is obtained by performing the following steps:

1. randomly choose an item $i \in V_{k+1}$;
2. assign i to one of the k feasible bins, say h;
3. move to bin $k+1$ all items $j \in V_h$ that are in conflict with i;
4. if the total weight of the items in V_h exceeds C, then move extra items from h to $k+1$ until the capacity constraint is satisfied.

In Step 1, the item i to be moved is randomly chosen with equal probability among all items in bin $k+1$. To perform Step 2, we first evaluate all possible bins $1, 2, \ldots, k$ for the insertion of item i. The one that, after Steps 3 and 4, will produce the new solution $\mathcal{S}'$ of minimum score is chosen, and the move is then performed.

The removal of items from bin h in Step 3 is straightforward, but the execution of Step 4 may be performed in different ways. We considered four possibilities to satisfy the capacity constraint:

- remove items one at a time according to a non-increasing surrogate weight ordering;

- remove items one at a time according to a non-decreasing surrogate weight ordering;

- remove items one at a time according to a random ordering;

- remove the subset of items that would determine the minimum score of the new solution $\mathcal{S}'$. We obtain this by solving the minimization version of a *0-1 Knapsack Problem* (KP01), in which each item j assigned to h (after Steps 1–3) is given a cost equal to $\max\left\{w_j^s, \epsilon\right\}$ and a weight w_j. Let $\widetilde{C}_h$ be the actual weight of the set of items assigned to h. The aim of the KP01 we address is to determine the subset of items of total weight greater than or equal to $\widetilde{C}_h - C$, and of minimum total cost. To solve this problem we use subroutine MT1 by [33], whose computational requirement is very limited on these small-size instances. Once the optimal subset is found, it is moved from bin h to bin $k+1$.

The removal procedure producing the best results is the one that removes items one at a time according to a random ordering. Although this procedure does not minimize the score of the resulting new solution $\mathcal{S}'$, it introduces a randomness which is beneficial to the algorithm, while the alternative procedures are deterministic and tend to move the same items at every iteration.

The Tabu Search moves from the current solution to the best one in the neighborhood, even if this leads to a worsening of the score. To avoid cycling, a tabu rule is implemented: an item i cannot enter the same bin h it entered during one of the last τ iterations, unless this would improve the incumbent solution. According to our computational experience, the best results are obtained by setting $\tau = 40$.

The Tabu Search is halted when 1) a complete feasible solution of k bins is found, or 2) a maximum number L of iterations has been performed. In case stopping condition 2 is met, the output is the (infeasible) solution of lowest score found during the search.

Diversification.

To improve the Tabu Search performance and allow an effective exploration of the solution space, we use a crossover operator and embed the Tabu Search in a Population Heuristic algorithm. We start with an initial pool of feasible solutions, obtained by means of the fast greedy algorithms of Section 3.3. Each greedy solution is then transformed into a partial feasible solution of value k (in the following simply *solution*), by moving all the items assigned to bins of index larger than $k+1$ to bin $k+1$. Each solution in the pool is then improved by means of the Tabu Search technique of the previous section. We also considered obtaining a solution of value k from a solution of larger value by keeping the best k bins according to (3.11). This choice did not improve the computational behavior of the algorithms. This should not surprise, since the greedy algorithms tend to better fill the first bins.

The pool of solutions is evolved through the following steps:

1. randomly choose two solutions (parents) from the pool;

2. generate an offspring by using a tailored crossover operator;

3. improve the offspring by applying the Tabu Search algorithm (according to our computational experience, the best results are obtained by setting the limit of iterations $L = 500$);

4. insert the offspring in the pool, deleting its worst (according to (3.11)) parent.

At Step 2, we use as crossover operator a variation of the specialized *Greedy Partition Crossover*, proposed by [16] for the VCP. Given two parents $\mathcal{S}^1 = \{V_1^1, \ldots, V_k^1, V_{k+1}^1\}$ and $\mathcal{S}^2 = \{V_1^2, \ldots, V_k^2, V_{k+1}^2\}$, the operator outputs an offspring $\mathcal{S}^3 = \{V_1^3, \ldots, V_k^3, V_{k+1}^3\}$ having the same structure, i.e., the items are partitioned into sets $V_1^3, \ldots, V_k^3$ which correspond to feasible bins, and a set V_{k+1}^3 which can be infeasible. The crossover selects alternatively parents $\mathcal{S}^1$ and $\mathcal{S}^2$. The h-th bin of the offspring, $1 \leq h \leq k$, is obtained by copying into the offspring the bin currently maximizing a given score in the selected parent. Then, items included in the new bin are deleted from both parents, and the process is iterated until the k bins of the offspring are built. Finally, all unassigned items are inserted into set V^3_{k+1}.

When picking up the next bin to be copied into the offspring, alternative scores can be considered:

- the cardinality of the bin;
- the total surrogate weight of the bin, corresponding to its weight or global degree, when $\alpha = 1$ and $\alpha = 0$, respectively.

Experimentally, the best choice is to consider as score of a bin its global surrogate weight with $\alpha = 1$, i.e., its global weight.

This generation–improvement–insertion procedure is iterated until 1) a feasible solution of k bins is found (i.e., a partial feasible solution is extended to a complete feasible solution) or 2) a time limit T_{PH} is reached.

Let L be the best lower bound provided so far and U the value of the incumbent solution. The Population Heuristic is first invoked with $k = U - 1$. Suppose a feasible solution is found. Then, if $k = L$ we have proven the optimality of the solution. If instead $k > L$, we set $U = k$ and re-execute the Population Heuristic with $k = U - 1$ and time limit T_{PH}. If no feasible solution is found but the time limit is reached, we continue our approach by executing the exact algorithm of the next section.

3.4 Set Covering model.

An alternative way to model the BPPC is the well known *Set Covering* (SC) formulation, which produced very interesting results for both the BPP (see, e.g., [51] and [35]) and the VCP (see, e.g., [34] and [31]).

We define $\mathcal{B}$ as the family of all the subsets of items $i \in V$ representing feasible bins. Each bin $b \in \mathcal{B}$ is associated with a binary variable ξ_b having value 1 iff the bin is selected. The BPPC can be formulated as

$$\min \sum_{b \in \mathcal{B}} \xi_b \tag{3.12}$$

$$\sum_{b \in \mathcal{B}: i \in b} \xi_b \geq 1 \quad i = 1, 2, \ldots, n \tag{3.13}$$

$$\xi_b \in \{0, 1\} \quad b \in \mathcal{B}. \tag{3.14}$$

The objective function (3.12) minimizes the total number of bins used. Constraints (3.13) state that every item $i \in V$ must belong to at least one bin. Note that if an item i belongs to more than one selected bin, it can be removed from all the bins but one, thus obtaining a feasible solution of the same value. Finally, constraints (3.14) impose variables ξ_b to be binary.

3.4.1 A better lower bound.

By relaxing the integrality constraints (3.14) to

$$\xi_b \geq 0 \quad b \in \mathcal{B}, \tag{3.15}$$

we obtain the Linear Programming (LP) relaxation of the SC model. Note that we are allowed to disregard constraints $\xi_b \leq 1$, since they are implied when the objective function (3.12) is considered with constraints (3.13). Let z^* be the value of the optimal solution of this LP relaxation. We obtain a valid BPPC lower bound, denoted L_{SC} in the following, by rounding up z^* to the next integer:

$$L_{SC} = \lceil z^* \rceil. \tag{3.16}$$

Model (3.12), (3.13) and (3.15) has an exponential number of binary variables (corresponding to the exponentially many subsets of V representing feasible bins), hence, we need *column generation* techniques to generate only the variables we need, among the exponentially many variables representing bins in $\mathcal{B}$.

To solve this model, called *master problem*, we first initialize it with a subfamily $\mathcal{B}'$ of the family of all bins $\mathcal{B}$. We then solve the model to optimality by means of 20, and obtain the optimal values π_i^*, $i \in V$, of the dual variables associated with constraints (3.13). To detect violated dual constraints, corresponding to variables (bins) to be added to the master problem, we solve the following *slave problem*, where the binary variable θ_i, $i \in V$, has value 1 iff item i is inserted into the bin under consideration:

$$\max \sum_{i=1}^{n} \pi_i^* \theta_i \tag{3.17}$$

$$\sum_{i=1}^{n} w_i \theta_i \le C \tag{3.18}$$

$$\theta_i + \theta_j \le 1 \qquad (i, j) \in E \tag{3.19}$$

$$\theta_i \in \{0, 1\} \quad i = 1, 2, \ldots, n. \tag{3.20}$$

Model (3.17)–(3.20) can be interpreted as a 0–1 *Knapsack Problem with Conflicts* (KPC), with profits π^* and conflicts between pairs of items imposed by constraints (3.19). If the optimal solution of the KPC has value greater than one (the cost of a bin in the BPPC), then we have found a column with negative reduced cost. We thus add it to the master problem and iterate. If instead the solution has value not greater than one, then we have reached the optimal solution of model (3.12), (3.13) and (3.15).

We solve each KPC by first invoking a simple greedy algorithm, in which items are sorted according to a given ordering, and inserted into the bin, one at a time, if they can fit. Items are sorted according to non-increasing values of the ratio π_i^*/w_i^s. The greedy algorithm is called 11 times, varying at each iteration the parameter α in the set of values $\{0.0, 0.1, \ldots, 1.0\}$ and thus obtaining different orderings (recall the surrogate weights depend on α in (3.10)). All the bins (columns) having negative reduced cost are added to the family $\mathcal{B}'$.

If no column to be added is found by the greedy approach, we solve (3.17)–(3.20) by means of CPLEX10. In order to reduce the computing time, we find the first bin corresponding to a violated dual constraint, if such a bin exists (i.e., we stop the execution as soon as we find a column with negative reduced cost).

If model (3.17)–(3.20) is solved to optimality, but no column with negative reduced cost is found, then we have obtained the optimal solution of the LP relaxation, and we stop the column generation phase.

If $L_{SC} = U$ we proved the optimality of the incumbent solution. If $L_{SC} < U$ and the solution of (3.12), (3.13) and (3.15) is integer, we set $U = L_{SC}$ and again terminate with a proof of optimality. If instead $L_{SC} < U$ and the solution is fractional, we need to embed the column generation phase within a Branch-and-Bound algorithm, obtaining what is usually defined as a *Branch-and-Price* algorithm (see the following section).

3.4.2 A Branch-and-Price algorithm.

In the Branch-and-Price algorithm, we use a depth first strategy and a binary branching, choosing as branching variable the one with the largest fractional part, and exploring first the subtree obtained by fixing the branching variable to 1. This strategy generally leads to feasible integer solutions within a short computing time.

Each node of the branching tree is solved through the column generation approach previously described. When a variable $\xi_{\widetilde{b}}$ is fixed to 1, all constraints (3.13) associated with items $i \in \widetilde{b}$ can be removed from the descendent subproblems, since automatically satisfied. Conversely, when a variable $\xi_{\widetilde{b}}$ is fixed to 0, we must forbid in the descendent subproblems that the column generation procedure creates again the associated bin $\widetilde{b}$. This can be ensured by adding to the slave problem the constraint

$$\sum_{i \in \widetilde{b}} \theta_i \leq |\widetilde{b}| - 1, \tag{3.21}$$

imposing that at most $|\widetilde{b}| - 1$ items among the ones in bin $\widetilde{b}$ are chosen. The resulting slave problem (3.17)–(3.21) is solved by CPLEX10.

3.5 Computational experiments.

The algorithms were coded in C++ and run on a Pentium IV 3 GHz with 1GB RAM, under a Linux operating system. To test their behavior, we created a test-bed that replicates the one originally proposed by [17] (note that the original test-bed is no longer available). This is done through the following steps.

We first consider the 8 classes proposed by [9] for the BPP. The first four classes consist of items with integer weights uniformly distributed in the range $[20, 100]$, to be packed into bins of capacity 150. The number n of items is the same for each instance in a class, and grows from the first to the fourth class, taking value, respectively, 120, 250, 500 and 1000. The second four classes consists of "triplets" of items of weights uniformly distributed in the range $[25, 50]$ and with one decimal digit, to be packed into bins of capacity 100. Each triplet in this class is generated so as to form an exact packing of weight 100, i.e., each bin may be completely filled by three items. The number of items for the instances in each class is, respectively, 60, 120, 249 and 501. As done in [17] the items sizes in these last four classes are multiplied by 10 to obtain integer data.

As done in [17], we selected the first 10 instances of the 20 originally proposed in each class. From each BPP instance we obtained 10 BPPC ones, by adding 10 random conflict graphs G, characterized by different density values (named δ in the following), varying from 0 to 0.9. This is done by assigning a value p_i to each vertex $i \in V$, according to a continuous uniform distribution in $[0, 1]$. Then, for each couple (i, j) a conflict is created if $(p_i + p_j)/2 \geq \delta$. The resulting test-bed is composed by 800 instances in total.

We made these instances publicly available at `http://www.or.deis.unibo.it`. Apart from the test-bed, this web site also contains the detailed computational results we obtained by running our algorithms on each single instance. In the following, due to the large size of the test-bed, we focus on average results. In Section 3.5.1 we present the results obtained by the combinatorial lower bounds described in Section 3.2 and by the lower bound based on the Set Covering formulation presented in Section 3.4.1. In Section 3.5.2 we illustrate the results of the upper bounds of Section 3.3 and finally, in Section 3.5.3, we present the performance of our overall algorithm (which includes the Branch-and-Price algorithm proposed in Section 3.4.2). In Section 3.5.4

we discuss an alternative exact algorithm based of a computationally faster lower bound and compare it with the Branch-and-Price algorithm.

3.5.1 Lower bounds results.

In Tables 3.1 and 3.2 we present the results obtained by the lower bounds developed for the BPPC. In Table 3.1 the results are given as averages for class, i.e., each line contains average values over 100 instances. In Table 3.2, instead, the results are given as averages for value of density, and each line contains average values over 80 instances. The first columns in the tables have the following meanings: *cl* gives the number of the original Falkenauer class (Table 3.1), δ gives the value of the density used to create the random conflict graph (Table 3.2) and U gives the average best upper bound value produced by the overall algorithm. Then, for each lower bound, say L_x, $\%g$ is the average percentage gap, computed as $100(U - L_x)/U$, and *time* is the computing time in seconds required to run L_x to completion.

In the tables we do not report the computing times of L_{BPP}^{DFF}, L_{MC} and L_{MC}^{imp}, since these values were always lower than 0.1 second. Moreover, we do not report the results obtained by L_{BPP}^{0}, since they are always dominated by L_{BPP}^{DFF}. Negative results were obtained by the lower bound L_{TVPP} based on the surrogate relaxation of Section 3.2.3. Indeed this bound could never improve the values obtained by the other combinatorial lower bounds, with computing times that could be very large (greater than one minute on average).

The dual feasible functions L_{BPP}^{DFF} and the max clique based bounds L_{MC} and L_{MC}^{imp} obtain very high percentage gap on all the 8 classes of instances (Table 3.1), always larger than 20% and 15%, respectively. This is reasonable, since these techniques consider just one aspect of the double nature of the BPPC. To compute a clique, the simple method L_{MC} seems slightly better than the new method L_{MC}^{imp}. However, when considering Table 3.2, which reports the results as averages for density value, the real performance of the lower bounds is correctly disclosed: L_{BPP}^{DFF} gradually worsens from the very low percentage gaps achieved on the sparse graphs (where the BPP component is more important) to the huge gaps obtained for the dense graphs (where the VCP component is more important). The opposite effect may be noted for L_{MC} and L_{MC}^{imp}. For graphs having densities varying from 0.3 to 0.9, L_{MC}^{imp} always outperforms L_{MC}, while the fact that L_{MC} obtains a better performance for graphs having densities lower than 0.3 is not significative, since both procedures have a very large gap on these instances, and are useless in practice.

Table 3.1: Computational results of the BPPC lower bounds (averages for class).

		L_{BPP}^{DFF}	L_{MC}	L_{MC}^{imp}	L_{CP}		L_{CP}^{imp}		L_{match}		L_{SC}	
cl	*U*	*%g*	*%g*	*%g*	*%g*	*time*	*%g*	*time*	*%g*	*time*	*%g*	*time*
1	68.17	22.05	15.13	17.63	1.38	0.0	0.24	0.0	6.29	0.2	0.01	22.8
2	139.49	20.33	16.21	18.05	1.36	0.1	0.18	0.1	6.99	1.5	0.00	57.4
3	277.82	20.30	17.86	17.54	1.79	0.4	0.17	0.4	7.32	13.1	0.07	151.2
4	555.03	20.54	19.25	17.26	1.90	2.0	0.09	2.3	7.58	122.4	0.04	275.1
5	32.06	28.37	22.93	22.93	0.98	0.0	0.98	0.0	10.97	0.0	0.05	38.3
6	63.54	27.49	22.94	22.94	0.27	0.0	0.27	0.0	11.44	0.4	0.12	44.8
7	130.59	26.79	22.58	22.58	0.11	0.1	0.11	0.1	11.93	5.4	0.05	68.5
8	264.87	27.35	21.93	21.93	0.05	0.5	0.05	0.5	11.90	63.4	0.03	232.6
avg	191.45	24.15	19.85	20.11	0.98	0.4	0.26	0.4	9.30	25.8	0.05	111.3

The constrained packing bounds L_{CP} and L_{CP}^{imp} obtain much better results. In particular L_{CP}^{imp} (which uses L_{MC}^{imp} in the clique computation) achieves a global percentage gap of just 0.26% from the best upper bound available, in a very limited time (0.4 seconds on average and never exceeding the 4 seconds for a single instance). Note that L_{CP}^{imp} on average outperforms L_{CP} on all classes. In particular, when considering instances grouped by density (Table 3.2), the two procedures have the same gap for graphs having densities up to 0.3, while L_{CP}^{imp} clearly outperforms L_{CP} for larger densities. This

Table 3.2: Computational results of the BPPC lower bounds (averages for δ value).

		L^{DFF}_{BPP}	L_{MC}	L^{imp}_{MC}	L_{CP}		L^{imp}_{CP}		L_{match}		L_{SC}	
δ	U	$\%g$	$\%g$	$\%g$	$\%g$	*time*	$\%g$	*time*	$\%g$	*time*	$\%g$	*time*
0.0	133.01	0.00	60.11	60.11	0.00	0.3	0.00	0.3	26.35	4.8	0.00	40.7
0.1	133.05	0.13	49.12	72.07	0.13	0.3	0.13	0.2	26.35	3.3	0.01	127.6
0.2	133.16	0.61	40.01	45.63	0.61	0.3	0.61	0.4	24.66	5.4	0.14	131.4
0.3	133.54	1.41	26.04	17.78	0.69	0.4	0.69	0.6	12.83	8.7	0.16	289.3
0.4	144.43	10.66	10.20	2.34	1.86	0.5	0.34	0.7	1.47	12.8	0.13	262.7
0.5	177.36	27.38	5.50	0.91	2.78	0.6	0.18	0.7	0.47	18.5	0.00	41.1
0.6	213.16	39.63	3.28	0.72	1.78	0.5	0.22	0.6	0.28	28.8	0.00	53.0
0.7	247.38	47.97	2.20	0.64	1.11	0.4	0.20	0.5	0.23	41.5	0.00	60.3
0.8	282.25	54.38	1.26	0.40	0.61	0.3	0.17	0.3	0.21	57.3	0.01	73.9
0.9	317.13	59.36	0.80	0.47	0.23	0.1	0.09	0.1	0.19	76.9	0.00	33.4
avg	191.45	24.15	19.85	20.11	0.98	0.4	0.26	0.4	9.30	25.8	0.05	111.3

confirms the observation of the previous paragraph: when the densities are low, regardless how a maximal clique is computed, the corresponding gap is very large, and using L_{MC} instead of L^{imp}_{MC} does not benefit L_{CP} (in some sense, it is the Vertex Coloring component of the problem that matters). On the contrary, when densities are increasing, the capability to compute a better clique embedded in L^{imp}_{CP} gives it an advantage on L_{CP}.

The behavior of L_{match} is not satisfactory, because of a quite large percentage gap on instances having low density. However, L_{match} outperforms L_{CP} for graphs having densities larger than 0.3 (while it is on average outperformed by L^{imp}_{CP} for all classes and densities). In addition, it is worth pointing out that in 32 cases out of 800 L_{match} improves all the previous lower bounds. This is however achieved with larger computing times: half a minute on average, 10 minutes for the slowest instance.

We first evaluate L_{CP} and L^{imp}_{CP}, and compute the combinatorial upper bounds of Section 3.3, in order to possibly prove the optimality of a large set of instances (more details are given in the next section), or to create an initial pool of solutions when optimality cannot be proved. Then the Population Heuristic of Section 3.3.1 is executed with a time limit of two minutes. For the instances which are not solved to optimality, we finally solve the LP relaxation of the Set Covering formulation described in Section 3.4.1 with a time limit of 1 hour. Clearly, we stop the computation as soon as the LP relaxation of the Set Covering formulation cannot improve on the incumbent lower bound. In two instances of our test-bed the column generation procedure did not converge within the time limit, and thus we have only the corresponding combinatorial lower bound. The results we obtain are very good, since we reduce the global average percentage gap to 0.05%. The average computing time (including the computation of L_{CP}, L_{CP}^{imp} and the time of the Population Heuristic) is less than 2 minutes. Note that the most time consuming class is the fourth one (for which $n = 1000$), followed by the eighth one ($n = 501$) and the third one ($n = 500$), while the most time consuming density is 0.3. Note also that instances with densities between 0.1 and 0.4 are the most difficult to be solved.

3.5.2 Upper bounds results.

In Tables 3.3 and 3.4 we present the results obtained by the upper bounds developed for the BPPC. Columns cl (Table 3.3) and δ (Table 3.4) have the same meanings as in the corresponding tables of the previous section, whereas L gives the average best lower bound value produced (equal to the optimum if a proof of optimality for the instance is given). Now the average percentage gap $\%g$ is computed as $100(U_x - L)/L$, where U_x denotes both the upper bound value obtained by a heuristic procedure, and the procedure itself.

In Table 3.3 we note that on average the best heuristic technique among the adaptations of the classical BPP heuristic procedures is the U_{WF}. This is a bit surprising, but we note that it can be a good idea to leave some empty space in the opened bins during the greedy execution, so that when assigning the last small (but possibly with many conflicts) items, one has more alternative ways to fit them. Among the parametric heuristic procedures,

$U_{BF(\alpha)}$ turns out to be superior. Since 11 values of α are attempted, it seems that in at least one of them the Best-Fit policy finds a good assignment. The average percentage gap of $U_{BF(\alpha)}$ is the smallest among the ones produced by the heuristic procedures. Comparing U_{H6} and U_{H6}^{imp}, we note that the solutions found by U_{H6}^{imp} are slightly worse than those of U_{H6}, but the computing time is eight times shorter, for the reasons discussed in Section 3.3. The computing times required to run the other heuristic algorithms are not reported since they are negligible: on average always smaller than 0.1 seconds.

Before invoking the Population Heuristic algorithm, whose initial pool of solutions is obtained through the previous heuristic procedures (more details on the population initialization are given in the next Section), we compute the lower bounds L_{CP} and L_{CP}^{imp}, and set the number of available bins k equal to the best heuristic solution value minus 1. Then the Population Heuristic iteratively solves the problem for decreasing values of k, until a proven optimal solution is obtained or the time limit T_{PH} of 2 minutes is reached. Not surprisingly, the Population Heuristic consistently outperforms the heuristic algorithms used for its initialization, achieving a 0.30% of average percentage gap, with an average computing time of less than 1 minute. The most time consuming class is the fourth one.

In Table 3.4 we first focus on the line in which $\delta = 0$, corresponding to a pure BPP. We note here that U_{WF} cannot outperform U_{FF} and U_{BF} (this confirms previous results in the literature), and that the values of the parametric heuristic procedures are equivalent to the ones of the greedy procedures (since the surrogate weights are always equal to the input weights and just the call with $\alpha = 0$ is performed). The other lines confirm the difficulty of the sparse graphs with $0.1 \leq \delta \leq 0.3$, for which both $\%g$ and *time* reach the highest values.

Table 3.3: Computational results of the BPPC upper bounds (averages for class).

		U_{FF}	U_{BF}	U_{WF}	$U_{FF(\alpha)}$	$U_{BF(\alpha)}$	$U_{WF(\alpha)}$	U_{H6}		U_{H6}^{imp}		U_{PH}	
cl	L	%g	%g	%g	%g	%g	%g	%g	*time*	%g	*time*	%g	*time*
1	68.17	7.86	8.02	5.71	1.03	0.98	1.56	2.67	0.0	3.28	0.0	0.10	22.4
2	139.49	8.37	8.53	5.45	0.96	0.92	1.51	2.50	0.1	3.74	0.0	0.21	52.1
3	277.69	8.17	8.33	5.51	0.90	0.85	1.47	2.27	1.2	3.47	0.2	0.20	69.7
4	554.87	7.59	7.75	5.12	0.87	0.74	1.38	1.96	17.5	2.84	2.0	0.22	107.8
5	32.06	9.47	9.73	9.15	3.92	3.59	3.93	3.05	0.0	2.53	0.0	0.45	37.5
6	63.49	10.30	10.61	10.09	4.30	4.18	4.36	3.15	0.0	2.91	0.0	0.62	40.0
7	130.55	9.83	10.17	10.18	4.50	4.38	4.51	3.16	0.1	3.06	0.0	0.39	51.9
8	264.82	8.52	8.92	9.05	4.22	4.14	4.10	2.84	1.4	2.78	0.3	0.21	58.9
avg	191.39	8.76	9.01	7.53	2.59	2.47	2.85	2.70	2.6	3.07	0.3	0.30	55.0

Table 3.4: Computational results of the BPPC upper bounds (averages for δ value).

		U_{FF}	U_{BF}	U_{WF}	$U_{FF(\alpha)}$	$U_{BF(\alpha)}$	$U_{WF(\alpha)}$	U_{H6}		U_{H6}^{imp}		U_{PH}	
δ	L	%g	%g	%g	%g	%g	%g	%g	*time*	%g	*time*	%g	*time*
0.0	133.01	7.90	7.90	8.07	7.90	7.90	8.07	1.64	1.4	2.84	0.0	0.15	38.0
0.1	133.04	8.00	8.01	8.08	5.26	4.83	5.83	5.97	1.4	6.93	0.0	1.09	95.7
0.2	133.13	9.37	9.43	8.99	5.82	5.43	5.79	6.89	0.7	7.24	0.0	0.65	94.9
0.3	133.41	15.33	15.81	13.78	5.00	5.00	5.25	6.88	1.2	7.15	0.1	0.52	95.6
0.4	144.06	18.37	19.06	14.52	0.99	0.92	2.11	2.51	1.4	3.87	0.2	0.47	58.3
0.5	177.36	11.95	12.35	9.12	0.27	0.17	0.45	1.05	3.3	1.05	0.4	0.05	29.8
0.6	213.16	7.62	7.93	5.66	0.20	0.20	0.34	0.76	5.7	0.68	0.5	0.03	42.0
0.7	247.38	4.90	5.17	3.75	0.16	0.13	0.33	0.62	5.3	0.49	0.6	0.03	40.3
0.8	282.25	2.91	3.08	2.32	0.14	0.07	0.22	0.41	3.4	0.31	0.7	0.01	34.9
0.9	317.13	1.27	1.32	1.04	0.11	0.06	0.13	0.29	1.7	0.21	0.7	0.00	21.0
avg	191.39	8.76	9.01	7.53	2.59	2.47	2.85	2.70	2.6	3.07	0.3	0.30	55.0

3.5.3 Overall algorithm results.

On the basis of the results obtained in the two previous sections, we structured a four-phase algorithm to solve the BPPC:

1. we compute L_{CP} and L_{CP}^{imp}, and set $L = \max\{L_{CP}, L_{CP}^{imp}\}$. We execute all the combinatorial heuristic procedures described in Section 3.3, with the exception of $H6$, and set U equal to the best solution value found;

2. we run the Population Heuristic with a time limit of 2 minutes. The Population Heuristic is initialized with the solutions found by the previously executed heuristic procedures; if more than 20 different solutions are found, only the 20 best ones are considered;

3. we solve the LP relaxation of the set covering formulation, possibly updating L and, when the corresponding solution is integer (and thus optimal), U. The initial pool of columns is set up by storing all the columns corresponding to the solutions found by the combinatorial heuristic procedures and to the feasible solution found for every value of k by the Population Heuristic. Before adding a new column, we check if the column is already contained in the pool, so as to avoid storing a duplicate information. We use hashing techniques to speed up this search step;

4. if the solution provided by the LP relaxation is fractional, we execute the Branch-and-Price algorithm described in Section 3.4.2. The overall time limit of Phases 3 and 4 is set to 10 hours.

We stop the execution of our algorithm as soon as $L = U$.

In Tables 3.5 and 3.6 we give the average values for class and for density value, respectively, produced by our algorithm. Apart from the columns indicating the class (Table 3.5) and the density value (Table 3.6), the tables are divided into five main parts: the first part gives the results obtained by our implementation of the most performing algorithms developed by [17] (L_{CP}, U_{FF} and U_{H6}); the remaining four parts give the results obtained by the four phases of the algorithm we developed. For each part of the tables we report the percentage gap (%g), the number of solutions solved to optimality (*opt*) and the overall computing time (*time*). For Phase 4 we additionally report the average number of nodes (*nodes*) required by our branching scheme (we consider one node if an instance is solved to optimality during one of the first three phases).

Table 3.5: Computational results of the overall algorithm (averages for class).

	Gendreau et al.			Phase 1			Phase 2			Phase 3			Phase 4			
cl	%*g*	*opt*	*time*	%*g*	*opt*	*time*	%*g*	*opt*	*time*	%*g*	*opt*	*time*	%*g*	*opt*	*nodes*	*time*
1	3.65	15	0.0	1.11	44	0.0	0.25	82	22.4	0.11	94	22.6	0.00	100	73.2	29.4
2	3.53	3	0.2	1.02	28	0.1	0.37	60	52.1	0.21	78	62.9	0.00	100	18.6	107.1
3	3.70	0	1.6	0.89	28	0.4	0.31	51	69.7	0.20	67	572.1	0.06	95	47.6	2195.4
4	3.56	0	19.5	0.77	27	2.7	0.27	48	107.8	0.22	56	1032.7	0.04	98	136.0	1911.9
5	4.06	53	0.0	3.02	64	0.0	1.48	69	37.5	0.30	94	38.2	0.00	100	2.8	38.7
6	3.31	54	0.0	2.76	60	0.0	0.78	67	40.0	0.62	75	44.8	0.13	95	64.0	1860.3
7	3.22	48	0.2	2.85	54	0.1	0.46	59	51.9	0.39	68	69.2	0.05	96	222.7	1582.1
8	2.86	49	2.0	2.66	59	0.5	0.23	63	58.9	0.21	70	242.2	0.03	96	73.3	3163.6
avg	3.49	222	2.9	1.88	364	0.5	0.52	499	55.0	0.28	602	260.6	0.04	780	79.8	1361.1

The first phase of the overall algorithm consistently outperforms the results obtained by the most performing algorithms in [17], providing a smaller percentage gap on all classes with shorter computing time (at least, with our implementation). 364 instances are solved to optimality (compared to 222), with an average computing time of 0.5 seconds. The average gap is 1.88%, compared to 3.49%. This gap reduction is obtained by both increasing the lower bound and decreasing the upper bound.

The second phase reduces the percentage gap to 0.52% and solves to optimality 499 instances, with an average computing time of 55 seconds. The third phase is mainly aimed at improving the value of the lower bound. However, for 5 instances the solution found is integer (and thus optimal) and better than the previous upper bound. The third phase reduces the gap to 0.28% and increases the number of optima to 602, with an average computing time of 260.6 seconds. The Branch-and-Price algorithm executed in the fourth phase solves to optimality 780 instances, with an average comput-

Table 3.6: Computational results of the overall algorithm (averages for δ value).

	Gendreau et al.			Phase 1			Phase 2			Phase 3			Phase 4			
cl	*%g*	*opt*	*time*	*%g*	*opt*	*time*	*%g*	*opt*	*time*	*%g*	*opt*	*time*	*%g*	*opt*	*nodes*	*time*
0.0	0.71	42	1.7	0.71	42	0.0	0.15	58	38.0	0.15	58	43.6	0.00	80	76.3	67.9
0.1	5.13	2	1.8	4.74	2	0.1	1.22	31	95.7	0.91	36	121.8	0.02	79	116.4	896.2
0.2	7.15	1	1.1	5.76	1	0.1	1.21	32	94.9	0.65	41	126.2	0.08	77	188.5	2231.6
0.3	7.48	1	1.6	5.35	2	0.2	1.08	31	95.6	0.52	42	329.8	0.16	71	332.4	6080.9
0.4	4.43	26	1.9	1.08	41	0.3	0.69	43	58.3	0.47	50	1754.8	0.13	73	60.2	3951.2
0.5	4.07	28	3.8	0.28	58	0.5	0.19	62	29.8	0.05	74	34.9	0.00	80	6.1	56.1
0.6	2.65	28	6.3	0.35	50	0.7	0.23	55	42.0	0.03	74	46.0	0.00	80	9.2	109.7
0.7	1.77	27	5.7	0.29	51	0.9	0.22	57	40.3	0.03	73	50.9	0.00	80	5.3	71.3
0.8	0.99	31	3.7	0.17	56	0.9	0.13	61	34.9	0.03	75	74.3	0.00	80	2.5	122.3
0.9	0.49	36	1.8	0.12	61	1.0	0.06	69	21.0	0.00	79	23.6	0.00	80	1.0	23.6
avg	3.49	222	2.9	1.88	364	0.5	0.52	499	55.0	0.28	602	260.6	0.04	780	79.8	1361.1

ing time of 1361.1 seconds (including 10 hours for each of the 20 instances that cannot be solved to optimality). The final percentage gap is reduced to 0.04%. It is worth noting that the lower bound produced by the column generation phase is very accurate and generally coincides with the optimal solution value (the Branch-and-Price algorithm is capable of improving its value for two instances only). This also justifies the choice of our branching scheme, which, under a depth-first strategy, has the advantage of quickly leading to optimal or near optimal integer solutions. This allows Phase 4 to improve 181 times the upper bound value found by the first three phases.

Tables 3.5 shows that classes 1, 2 and 5 are easy, as all their instances are solved to optimality in less than two minutes on average. Table 3.6 confirms the difficulty of the sparse graphs. Pure BPP instances ($\delta = 0$) are all solved to optimality in around one minute. Also instances with high density ($\delta \geq 0.5$) are all closed to optimality, in an average time of less than two minutes. All the unsolved instances are characterized by $0.1 \leq \delta \leq 0.4$. For these instances the computing times and the number of nodes of the search tree are high.

In Figure 3.2 we graphically represent the evolution of the average lower bound value L and upper bound value U, for the algorithms in [17] and for the four phases of our algorithm. The average lower bound value for our implementation of the algorithms in [17] is 188.97. This value is increased to 191.21 by Phase 1, and further increased to 191.39 by Phases 3. Phase 4 increases the bound for 2 additional instances, and the final average lower bound is 191.39. A similar behavior is noted for the upper bound value, which is 194.47 for the algorithms in [17], 193.18 at the end of Phase 1, and 191.73, 191.72 and 191.45 at the end of Phases 2, 3 and 4, respectively.

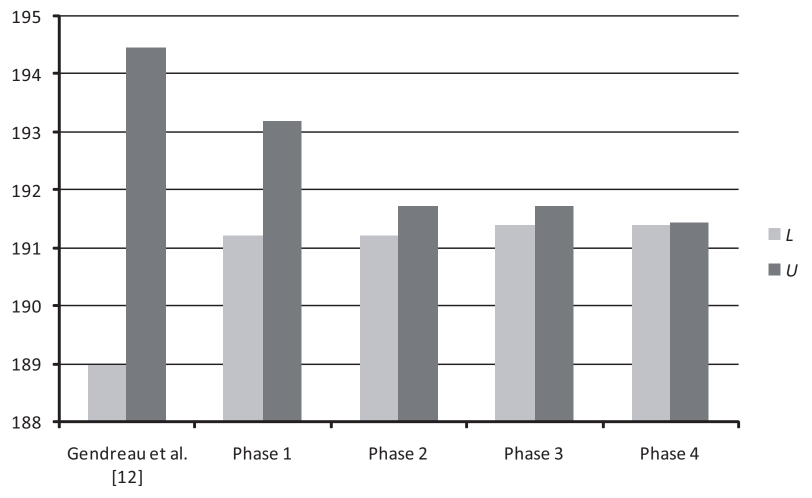


Figure 3.2: Evolution of the lower and upper bound values (averages over the complete set of 800 instances).

In Table 3.7 we focus on the subset of the 20 difficult instances which were not solved to optimality. The first part of the table contains the class, the density and the progressive number ($n°$) of the instance (from 1 to 10, as in our web site) for a given class and density. The three remaining parts of the table present the results of the last three phases of the algorithm. In particular, column *#col* gives the number of columns stored into the pool at a certain step. Column *diff.* gives the difference between U and L, and the last column gives the number of nodes of the search tree.

For these instances, the lower bound computed during the first phase can be improved by the LP relaxation of the Set Covering formulation only in two cases, whereas the Branch-and-Price does not lead to further improvements. Also the upper bound of the second phase is never improved. The heuristic algorithms produce on average 3156 columns, that are given as input to the Set Covering model. Just solving the corresponding LP relaxation raises $\#col$ to 3628 on average. Finally the Branch-and-Price algorithm increases this value to 55030. We note that in two cases our algorithm fails to solve the LP relaxation within the given time limit. For these two cases the difference between L and U is high, whereas for all the other cases is usually between 1 and 4. Finally, we tried to solve these 20 instances by giving a larger time limit to the Population Heuristic (20 minutes instead of 2 minutes) and by performing multiple runs with different seeds for the random number generator used by the Tabu Search algorithm described in Section 3.3.1. The result is that 10 additional instances can be solved to optimality within the second phase. The values of the optimal solutions, which correspond to those of the corresponding initial lower bounds, are marked with a * in the table. In addition, we could improve two upper bounds in the table, namely that of instance (3,0.4,4), for which we find a solution of value 206, and that of instance (3,0.4,7), with a solution of value 211.

3.5.4 An alternative exact algorithm based on a fast lower bound.

As one can observe, the overall algorithm described in the previous section is quite fast in the first two phases, requiring on average 55 seconds of computing time. It is instead slower in the last two phases, where it computes a lower bound, based on column generation, which is very effective but computationally expensive. We thus tested as well a classical Branch-and-Bound algorithm based on a less expensive lower bound computation.

We designed an exact approach inspired to a well known branching rule proposed by [34] for the VCP. The basic idea is as follows: at each node

Table 3.7: Computational results on the difficult instances (* = proved to be optimal).

			Phases 1-2				Phase 3				Phase 4				
cl	δ	n°	*L*	*U*	*#col*	*time*	*L*	*U*	*#col*	*time*	*L*	*U*	*diff.*	*#col*	*nodes*
3	0.4	4	204	207	3763	132	204	207	4535	5543	204	207	3	5555	124
3	0.4	5	206	207	3563	137	206	207	4101	8373	206	207	1	4280	15
3	0.4	7	207	212	3655	122	208	212	4631	27170	208	212	4	4651	20
3	0.4	8	204	206	3918	129	205	206	4405	1171	205	206	1	4656	51
3	0.4	9	196	200	3730	122	196	200	4549	7080	196	200	4	4707	39
4	0.4	8	404	411	7826	139	404	411	9122	*t.lim.*	404	411	7	9122	1
4	0.4	10	397	406	7656	128	397	406	9168	*t.lim.*	397	406	9	9168	1
6	0.2	8	40*	41	547	121	40	41	715	129	40	41	1	105371	30
6	0.2	10	40*	41	493	121	40	41	699	127	40	41	1	111910	2670
6	0.3	5	40*	41	743	121	40	41	927	145	40	41	1	61077	30
6	0.3	6	40	41	842	121	40	41	1043	142	40	41	1	69635	1468
6	0.3	9	40*	41	765	121	40	41	980	141	40	41	1	74791	32
7	0.1	3	83*	84	793	125	83	84	1144	567	83	84	1	160540	3939
7	0.2	7	83*	84	1356	128	83	84	1558	146	83	84	1	292490	7320
7	0.3	2	83*	84	1971	126	83	84	2212	231	83	84	1	3590	35
7	0.3	4	83	84	2092	124	83	84	2420	231	83	84	1	125212	8539
8	0.3	1	167*	168	5015	187	167	168	5219	4001	167	168	1	34139	2394
8	0.3	4	167	169	4566	152	167	169	4884	2513	167	169	2	4952	26
8	0.3	5	167*	168	4879	160	167	168	5052	2172	167	168	1	5504	21
8	0.3	10	167*	168	4948	243	167	168	5200	2244	167	168	1	9256	151
avg			150.90	153.15	3156	138	151.00	153.15	3628	6724	151.00	153.15	2.15	55030	1345

of the search tree we select two non-conflicting items, say i and j, such that $w_i + w_j \leq C$ and $(i, j) \notin E$. We then consider the two subproblems obtained by: 1) adding a conflict between i and j, and 2) collapsing i and j into a single item. The first case is simply obtained by setting $E = E \cup (i, j)$. The second case is obtained by removing i and j from the instance and creating a new item k, having weight $w_k = w_i + w_j$ and conflicts $(k, h) \in E$ with all items h previously being in conflict with i and/or j. In both cases we obtain a BPPC instance with one more edge or one less item, respectively. We can thus apply the fast bounding procedures of Sections 3.2 and 3.3. If these bounds solve the subproblem to optimality or allow us to conclude that the solution of the subproblem cannot improve on the best incumbent solution, then we backtrack. Otherwise we continue exploring the search tree by selecting the next couple of items.

We integrated this Branch-and-Bound algorithm with the first two phases of the overall algorithm, and tested it on all the instances not solved to optimality after the second phase. We explore the search tree according to a depth first strategy. At each node of the search tree, we compute the lower bound L_{CP}^{imp} and the upper bound U_{FF} (we also tested U_{BF} and U_{WF} but this produced worse results). If $L_{CP}^{imp} = U_{FF}$, then the corresponding subproblem is optimally solved. If L_{CP}^{imp} is not smaller than the best incumbent solution value, we can fathom the node. Otherwise we branch by choosing two non-conflicting items i and j and generating the two subproblems as explained above.

The choice of items i and j is not trivial, and should be aimed at improving the lower bound value produced by L_{MC}^{imp} for both descendent nodes. We recall that this procedure initially computes a clique K and then solves a transportation problem. We considered several alternatives, all aimed at enlarging the initial clique and/or increasing the number of conflicts encountered when solving the transportation problem. We always choose first an item $j \notin K$ and then an item $i \in K$ among those not in conflict with j (so as to possibly enlarge K in few iterations). We considered the following alternatives: i) j of largest weight and then i of largest weight; ii) j of largest degree and then i of largest degree; iii) j of largest degree in the induced subgraph $G \setminus K$ and then i of largest degree. Our computational experiments showed that the best configuration is iii), which is much better than i) and slightly better than ii).

Concerning the computation of the maximal clique K, we considered two options: i) computing the clique from scratch at each node; ii) transferring the clique from each father node to its descendent nodes (note that if K is a maximal clique at a given node, then K remains a clique, possibly not maximal, when an item $i \in K$ is collapsed with an item $j \notin K$ or an edge (i, j) is added to E). Our computational experiments showed that the best choice is ii), as one might expect since i and j are selected with respect to the father clique.

We tested the best configuration of this Branch-and-Bound algorithm, with a time limit of 1000 seconds (not including the time required by Phases 1 and 2), and compared it with the results obtained by our original overall algorithm within the same computing time. The original algorithm solved to optimality 233 out of the 301 instances remained unsolved after Phase 2, whereas the most performing configuration of the Branch-and-Bound algorithm described in this section solved only 62 instances. This result confirms the efficacy of the overall algorithm.

3.6 Conclusions.

The Bin Packing Problem with Conflicts is a combinatorial optimization problem of practical interest, since it models many real world situations, occurring in cutting and packing, scheduling, and vehicle routing fields, just to cite some. It is also an interesting problem from the theoretical viewpoint, since it generalizes both the Bin Packing Problem and the Vertex Coloring Problem, both well known and notoriously difficult.

In this work we presented several algorithms devoted to the solution of this problem. By considering the simple lower and upper bounds, we notice how important is to take care of both aspects of the problem (packing and coloring) at the same time, in order to obtain a good performance. Tabu Search algorithms always give a successful scheme to address these combinatorial problems from a heuristic point of view, although in this case they need a diversification phase to explore more efficiently the very large solution space. The Set Covering formulation, solved by column generation and Branch-and-Price, provides very interesting results on the problem. This confirms previous results on the classical Bin Packing Problem.

Extensive experiments on a benchmark test-bed show the good behavior of the algorithms we implemented, that are able to solve to optimality 780 instances out of 800 and to consistently improve previous algorithms in the literature. We made the benchmark instances available on the web, so as to facilitate future research on this problem.

Conclusions.

We have examined a relevant logistic issue arising in the organization of fairs, namely the optimization of the stands allocation in the exhibition space. We proposed two mathematical models for the optimal solution of a basic version of the problem. In real world applications the constraints and requirements (security, ease of access, services) can vary considerably from case to case.

We have examined a number of practical situations, and the corresponding requirements, and we have shown how the two basic mathmatical models, and the corresponding decision support system, can be adapted to handle them. The conclusion of this study is that these mathematical models are flexible enough to be used with satisfactory results in a variety of different real world situations.

We studied the *Generalized Traveling Salesman Problem* which is a generalization of the classical Traveling Salesman Problem (TSP) where the vertices are divided in clusters and the salesman must visit at least one vertex in each cluster.

We have developed a Multi-start Heuristic for the *E-GTSP*, and we compared the proposed method with the best state-of-the-art algorithms on a set of benchmark instances from the literature, and showed the effectiveness of the approach. It turned out that, on the set of smaller instances (with up to 442 vertices), the proposed algorithm finds always the optimal solution in less than four seconds, and clearly outperforms the most effective heuristics. For the set of larger instances (with up to 1084 vertices) the proposed algorithm always improves, in comparable computing times, the solution values obtained by the genetic algorithm recently presented by Silberholz and Golden [47].

We considered a particular *Bin Packing Problem*, noted as the *Bin Packing Problem with Conflicts*, in which some items cannot be assigned to the same bin. The problem combines Vertex Coloring and Bin Packing aspects, and turned out to be particularly challenging.

We have solved the problem through combinatorial lower bounds, parametric greedy heuristics, a population heuristic making use of an effective tabu search inner procedure, and a column generation. Extensive experiments on a benchmark test-bed show the good behavior of the algorithms we implemented, that are able to solve to optimality 780 instances out of 800 and to consistently improve previous algorithms in the literature.Moreover we showed how a clear integration between a metaheuristic and column generation may yield to surprising advances in the performance of the algorithm, both in terms of speed and solution quality.

Acknowledgements.

We thank the GRAPHVS CONS. COM & REP. LTDA and the Universidade Estadual do Ceará, Fortaleza (Brazil), the Ministero dell'Istruzione, dell'Università e della Ricerca (Italy). We also thank the ALFA PROJECT, under grant II-0457-FA-FCD-F1-FC. Their support is gratefully acknowledged.

Thanks are also due to Ana Mércia (Municipality Environmental Manager), Sérgio Pinheiro (Fortaleza Tourism Municipality Secretary) and Francisco Carlos Simões (President of the Beira Mar Fair Association, ASFABEM) for their kind support during the execution of the tests on the Fortaleza fair. We finally thank Valter Franceschini (President of the Fiere di Reggio Emilia), Manuela Maccagnani and Lisa Bianchini, for their help during the execution of the tests on the Reggio Emilia case study.

We thank the professors Martello Silvano (University of Bologna), Mamuel Iori (University of Reggio Emilia) and José Marcos Negreiros Gomes (State University of Ceará - Brazil), likewise the PhD's Enrico Malaguti, Valentina Cacchiani.

Bibliography

[1] Brenda S. Baker and Edward G. Coffman Jr. Mutual exclusion scheduling. *Theor. Comput. Sci.*, 162(2):225–243, 1996.

[2] Hans L. Bodlaender and Klaus Jansen. On the complexity of scheduling incompatible jobs with unit-times. In Andrzej M. Borzyszkowski and Stefan Sokolowski, editors, *MFCS*, volume 711 of *Lecture Notes in Computer Science*, pages 291–300. Springer, 1993.

[3] Valentina Cacchiani, Albert Einstein Fernandez Muritiba, Marcos Negreiros, and Paolo Toth. A multi-start heuristic algorithm for the generalized traveling salesman problem. In *CTW*, pages 136–138. University of Milan, 2008.

[4] Alberto Caprara and Paolo Toth. Lower bounds and algorithms for the 2-dimensional vector packing problem. *Discrete Applied Mathematics*, 111(3):231–262, 2001.

[5] M W Carter, G Laporte, and S T Lee. Examination timetabling: algorithmic strategies and applications. *Journal of the Operational Research Society*, pages 47–373, 1996.

[6] N. Christofides, A. Mingozzi, and P. Toth. The vehicle routing problem. In N. Christofides, A. Mingozzi, P. Toth, and C. Sandi, editors, *Combinatorial Optimization*, pages 315–338. Wiley, Chichester, 1979.

[7] Dongarra and J.J. Performance of various computers using standard

linear equations software. Working paper CS-89-85, Computer Science Department, University of Tennesse, 2004.

[8] J. Edmonds. Paths, trees, and flowers. *Can. J. Math.*, 17:449–467, 1965.

[9] Emanuel Falkenauer. A hybrid grouping genetic algorithm for bin packing, 1996.

[10] Sándor P. Fekete and Jörg Schepers. New classes of fast lower bounds for bin packing problems. 2001.

[11] A. E. Fernandes Muritiba, S. Martello, Marcos Negreiros, and M. Iori. Models and algorithms for fair layout optimization problems. *Annals of Operation Research*, pages 1–14, 2008.

[12] Albert E. Fernandes Muritiba, Manuel Iori, Enrico Malaguti, and Paolo Toth. Algorithms for the Bin Packing Problem with Conflicts. *INFORMS JOURNAL ON COMPUTING*, page ijoc.1090.0355, 2009.

[13] Matteo Fischetti, J.J. Salazar-González, and Paolo Toth. A branch-and-cut algorithm for the symmetric generalized travelling salesman problem, 1997.

[14] Matteo Fischetti, Paolo Toth, and J.J. Salazar-González. The symmetric generalized travelling salesman polytope, 1995.

[15] Harold N. Gabow. An efficient implementation of Edmonds' algorithm for maximum matching on graphs. *J. Assoc. Comput. Mach.*, 23:221–234, 1976.

[16] Philippe Galinier and Jin-Kao Hao. Hybrid evolutionary algorithms for graph coloring. *J. Comb. Optim.*, 3(4):379–397, 1999.

[17] Michel Gendreau, Gilbert Laporte, and Frédéric Semet. Heuristics and lower bounds for the bin packing problem with conflicts. *Computers & OR*, 31(3):347–358, 2004.

[18] Pierre Hansen, Alain Hertz, and Julio Kuplinsky. Bounded vertex colorings of graphs. *Discrete Mathematics*, 111(1-3):305–312, 1993.

[19] A.L. Henry-Labordere. The record balancing problem: A dynamic programming solution of a generalized traveling salesman problem, 1969.

[20] S.A. ILOG. User's manual and reference manual, 2006. `http://www.ilog.com/`.

[21] Klaus Jansen. An approximation scheme for bin packing with conflicts. *J. Comb. Optim.*, 3(4):363–377, 1999.

[22] Klaus Jansen and Sabine R. Öhring. Approximation algorithms for time constrained scheduling. *Inf. Comput.*, 132(2):85–108, 1997.

[23] Tommy R. Jensen and Bjarne Toft. *Graph coloring problems*. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, Chichester-New York-Brisbane-Toronto-Singapore, 1995.

[24] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences, Academic*, 9, 1974.

[25] G. Laporte, A. Asef-Vaziri, and C. Sriskandarajah. Some applications of the generalized traveling salesman problem, 1996.

[26] G. Laporte and S. Desroches. Examination timetabling by computer. *Computers and Operations Research*, 11:351–360, 1984.

[27] G. Laporte, H. Mercure, and Y. Nobert. Generalized traveling salesman problem through n sets of nodes: The asymmetrical case, 1969.

[28] G. Laporte and Y. Nobert. Generalized traveling salesman through n sets of nodes: An integer programming approach, 1983.

[29] A. Lodi, S. Martello, and D. Vigo. Recent advances on two-dimensional bin packing problems. *Discrete Applied Mathematics*, 123:379–396, 2002.

[30] Enrico Malaguti, Michele Monaci, and Paolo Toth. Models and heuristic algorithms for a weighted vertex coloring problem. *Journal of Heuristics*.

[31] Enrico Malaguti, Michele Monaci, and Paolo Toth. A metaheuristic approach for the vertex coloring problem. *INFORMS Journal on Computing*, 20(2):302–316, 2008.

[32] Enrico Malaguti and Paolo Toth. A survey on vertex coloring problems. *Journal of Heuristics*, 2009.

[33] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Chichester, 1990.

[34] A. Mehrotra and M. A. Trick. A column generation approach for graph coloring. *INFORMS Journal on Computing*, 8:344–354, 1996.

[35] Michele Monaci and Paolo Toth. A set-covering-based heuristic approach for bin-packing problems. *INFORMS Journal on Computing*, 18(1):71–85, 2006.

[36] Craig Morgenstern. Distributed coloration neighborhood search. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, 1993*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 335–357. American Mathematical Society, 1996.

[37] P. Moscato and C. Cotta. A gentle introduction to memetic algorithms. In F. Glover and G.A. Kochenberger, editors, *Handbook of Metaheuristics*, pages 105–144. Kluwer Academic Publishers, Boston, 2003.

[38] C. E. Noon. The generalized traveling salesman problem. 1988.

[39] C.E. Noon and J.C. Bean. A lagrangian based approach for the asymmetric generalized traveling salesman problem, 1991.

[40] C.E. Noon and J.C. Bean. An efficient transformation of the generalized traveling salesman problem, 1993.

[41] C. M. Pintea, P. C. Pop, and C. Chira. The generalized traveling salesman problem solved with ant algorithms, 2007.

[42] G. Reinelt. Tsplib-a traveling salesman problem library, 1991.

[43] J. Renaud and F.F. Boctor. An efficient composite heuristic for the symmetric generalized traveling salesman problem, 1998.

[44] J.P. Saksena. Mathematical model of scheduling clients through welfare agencies, 1970.

[45] P. Schneuwly and M. Widmer. Layout modeling and construction procedure for the arrangement of exhibition spaces in a fair. *International Transactions in Operational Research*, 10:311–338, 2003.

[46] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, Chichester, 1986.

[47] J. Silberholz and B. L. Golden. *The Generalized Traveling Salesman Problem: a new Genetic Algorithm approach*, pages 165–181. Springer, 2007.

[48] S.P. Singh and R.R.K. Sharma. A review of different approaches to the facility layout problems. *The International Journal of Advanced Manufacturing Technology*, 30:425–433, 2006.

[49] L. V. Snyder and M. S. Daskin. A random-key genetic algorithm for the generalized traveling salesman problem, 2006.

[50] S. S. Srivastava, S. Kumar, R. C. Garg, and P. Sen. Generalized traveling salesman problem through n sets of nodes, 1969.

[51] F. Vanderbeck. Computational study of a column generation algorithm for bin packing and cutting stock problems. *Mathematical Programming*, 86(3):565–594, 1999.

[52] G. Wäscher, H. Haußner, and H. Schumann. An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183:1109–1130, 2007.

[53] M. Widmer. *Modèles mathématiques pour une gestion efficace des ateliers flexibles*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne, Switzerland, 1990.