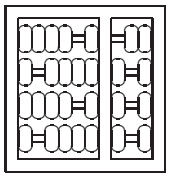

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Integer programming models for the SONET ring assignment problem

Elder M. Macambira *Nelson Maculan*
Cid C. de Souza

Technical Report - IC-05-026 - Relatório Técnico

October - 2005 - Outubro

Integer programming models for the SONET ring assignment problem

Elder M. Macambira [*], Nelson Maculan
Universidade Federal do Rio de Janeiro
Programa de Engenharia de Sistemas e Computação, COPPE
Caixa Postal 68511, 21941-972, Rio de Janeiro – RJ – Brazil
{elder,maculan}@cos.ufrj.br

Cid C. de Souza [†]
Universidade Estadual de Campinas
Instituto de Computação
Caixa Postal 6176, 13084-971, Campinas – SP – Brazil
cid@ic.unicamp.br

Abstract

In this paper we consider the SONET ring assignment problem (SRAP) presented by Goldschmidt et al. in [9]. The authors pointed out to the inadequacy of solving SRAP instances using their integer programming formulation and commercial linear programming solvers. Similar experiences with IP models for SRAP are reported in [1]. In this paper we presented some variants of IP formulations for SRAP and tested the performance of standard branch-and-bound algorithms implemented in a commercial code when computing those models. The results are significantly better than those reported earlier in the literature. Moreover, we also reformulate the problem as a set partitioning model amended with an additional knapsack constraint. This new formulation has an exponential number of columns and, to solve it, we implemented a branch-and-price/column generation algorithm. Extensive computational experiments with our algorithm showed that it is orders of magnitude faster than its competitors. Hundreds of instances taken from [1] and [9] which could not be solved in hours of computation are solved here to optimality in just a few seconds.
Keywords: SONET ring assignment, integer programming models, column generation, branch-and-price algorithm.

1 Introduction

Consider the following graph optimization problem. We are given an undirected graph $G = (V, E)$ and a positive integer B. Associated to each edge (u, v) in E there is a non

[*]Research supported by grants from CNPq (146245/1999-7) and FAPERJ (E-26/152.603/2001).
[†]Corresponding author. Research supported by grants from FAPESP (01/14205–6), CAPES (BEX 04444/02–2), CNPq (302588/02–7 and (Pronex 664107/97–4)

negative integer d_{uv}. A feasible solution of the problem is a partition of the vertices in G into sets $V_1, V_2, \ldots, V_k$ such that: *(i)* $\sum_{(u,v)\in G[V_j]\cup\delta(V_j)} d_{uv} \leq B$ for all $j \in \{1, \ldots, k\}$ and *(ii)* $\sum_{j=1}^{k} \sum_{(u,v)\in\delta(V_j)} d_{uv} \leq B$, where $G[V_j]$ is the graph induced by V_j in G and $\delta(V_j)$ is the set of edges with exactly one extremity in V_j. The goal is to find a feasible solution that minimizes the size of the partition, i.e., the value of k.

This graph problem is also known as the `SONET` *(or* `SDH`*) ring assignment problem*, or `SRAP` for short. The `SRAP` was investigated recently by Goldschmidt et al. in [9]. In their paper the authors motivated the study of the problem by showing its relevance for the design of fiber-optics telecommunication networks using the `SONET` (Synchronous Optical NETwork) or `SDH` (Synchronous Digital Hierarchy) technology. In this context, the vertices of graph G are associated to client sites while the edges are associated to the existence of traffic demand between pairs of clients. The edge weights measure the actual demand of communication among clients. Given a feasible solution to the graph problem stated above, the vertices in each subset of the partition form a *local ring* or simply a *ring*. Due to physical limitations on the equipments used in building the network, the total demand in each ring must not exceed a constant B. Besides, the total demand between clients in different clusters is handled by an additional ring called the *federal ring*. The connection of a local ring to the federal ring is made possible through a device known as a digital cross-connect (`DCS`). Since the capacity of the federal ring is also bounded by B, the sum of the weights of the edges in the multicut corresponding any feasible solution cannot be larger than that amount. Finally, because the `DCS`'s are by far the most expensive equipments needed to implement a network, a basic problem in the design of low-cost `SONET` networks asks for a solution that minimizes the number of local rings. One can easily check that the graph problem discussed at the beginning of this section correctly models the latter problem.

We refer to [9] for a more detailed discussion on the architecture of `SONET` networks and a review of the literature on the `SRAP`. In their work the authors proved that `SRAP` is $\mathcal{NP}$-hard. They also introduce an integer programming formulation for the problem and propose heuristics and exact methods to solve it. Another recent work on heuristics for `SRAP` can be found in [1] where tabu and scatter search meta-heuristics are proposed.

This paper focus on the solution of `SRAP` via integer programming techniques. Our motivation comes from a comment of Goldschmidt and co-authors in [9] about their IP formulation and which is reproduced below:

> "Our results indicate that using this formulation in conjunction with a commercial ILP solver (i.e., `CPLEX`) is not a practical approach for solving `SRAP`."

The authors also reported that the IP approach is particularly unattractive to deal with infeasible instances. In our work we introduced different IP formulations for `SRAP` designed to overcome these drawbacks. Our major result is a set partitioning type model having an extra knapsack constraint which can be efficiently solved by a column generation/branch-and-price type algorithm that we propose. The conclusions of this work are supported by extensive experimentation.

The paper is organized as follows. The next section discusses several different IP formulations for `SRAP` including the original formulation from [9] and the set partitioning model

cited above. Section 3 introduces the main ingredients of our branch-and-price algorithm. In section 4 we describe our computational experiments and analyze the results. Finally, in section 5 we draw some conclusions and discuss future directions. Appendix A describes a simple heuristic based on the GRASP paradigm (see [8]) which computes initial primal bounds in our tests.

2 Integer programming models for SRAP

This section describes alternative IP formulations for SRAP. Experiments with these formulations are reported in the forthcoming sections. From now on, the parameters n and m are used to denote the number of vertices and edges in G, respectively, while the set $\{1, \ldots, n\}$ is represented by N.

The IP model from [9]

In [9] the authors introduce an IP formulation for SRAP that goes as follows. It has four groups of binary variables, namely y, x, p and z. In the first one, there are n variables each representing a possible local ring to be opened[1] in a feasible solution. Thus, the variable y_i is one if and only if the i-th ring is opened. In the second group, there is a variable x_{ui} for each $u \in V$ and $i \in N$. The value of x_{ui} is one if and only if vertex u is assigned to ring i. For every edge (u, v) in E, with $u < v$, and every $i \in N$, p_{uvi} is one if and only if both vertices u and v are assigned to ring i. This third group of variables has mn elements. Finally, in the fourth group, we have $2mn$ z variables. Each z variable is indexed by three elements, the first two taken from $V \times V \setminus \{(u, u) \mid u \in V\}$ and the last one from N. The variable z_{uvi} is then defined to be one if and only if u is in ring i while v is not. A possible formulation for SRAP with these variables is given below.

[1] We say that a ring is *open* if there is at least one vertex assigned to it.

$$(GM) \quad \min \quad \sum_{i=1}^{n} y_i \tag{1}$$

$$s.t. \quad \sum_{(u,v)\in E} d_{uv} p_{uvi} + \sum_{u\in V} \sum_{v|(u,v)\in E} d_{uv} z_{uvi} \leq B, \qquad \forall i \in N, \tag{2}$$

$$\sum_{(u,v)\in E|u<v} \sum_{i=1}^{n} d_{uv} z_{uvi} \leq B, \tag{3}$$

$$\sum_{i=1}^{n} x_{ui} = 1, \qquad \forall u \in V, \tag{4}$$

$$x_{ui} - y_i \leq 0, \qquad \forall u \in V, \forall i \in N, \tag{5}$$

$$x_{ui} + x_{vi} - p_{uvi} \leq 1, \qquad \forall (u,v) \in E,\ \forall i \in N, \tag{6}$$

$$x_{ui} - x_{vi} - z_{uvi} \leq 0, \qquad \forall u \in V, \forall (u,v) \in E,\ \forall i \in N, \tag{7}$$

$$x_{ui} \in \{0,1\}, \qquad \forall u \in V,\ \forall i \in N, \tag{8}$$

$$y_i \in \{0,1\}, \qquad \forall i \in N, \tag{9}$$

$$p_{uvi} \in \{0,1\}, \qquad \forall (u,v) \in E,\ \forall i \in N, \tag{10}$$

$$z_{uvi} \in \{0,1\}, \qquad \forall u \in V, \forall (u,v) \in E,\ \forall i \in N. \tag{11}$$

Let us briefly examine the constraints in formulation (GM). Constraints (2) and (3) limit the capacities of the local and federal rings. Constraints (4) ensure that every vertex is assigned to a ring. Constraints (5) ensure that a ring is open whenever a vertex is assigned to it. For a proper triple (u, v, i), constraint (6) forces p_{uvi} to one if both vertices u and v are assigned to ring i while constraint (7) forces z_{uvi} to one if vertex u is assigned to ring i and v is not assigned to that ring.

The mindful reader has certainly noticed that equations (6) ((7)) are not enough to express the intended relationship between x and p (z) variables. However, as observed by Goldschmidt et al., given a proper triple (u, v, i) and a feasible solution to (GM) if $p_{uvi} = 1$ and either $x_{ui} = 0$ or $x_{vi} = 0$, another feasible solution with the same cost is obtained by simply setting p_{uvi} to zero. An analogous reasoning applies if $z_{uvi} = 1$ and either $x_{ui} = 0$ or $x_{vi} = 1$. Therefore, we can drop from the `SRAP` formulation the constraints that force the p or z variables to zero.

The packing model

Goldschmidt et al. observed in their paper that formulation (GM) is not suitable to identify SRAP instances which are infeasible. A possible way to overcome this disadvantage is to relax the previous model by replacing constraints (4) by constraints of the form

$$\sum_{i=1}^{n} x_{ui} \leq 1. \tag{12}$$

However by doing that we also have to change the objective function so that it favors the feasible solutions that are actual partitions of the vertex set if they exist. This technique is not a novelty in the solution of hard combinatorial problems. Examples of its application can be found for the generalized assignment problem (see [13]) or the traveling salesman problem (see [10]). The new objective function is given by

$$\max\{\sum_{u\in V}\sum_{i=1}^{n} \alpha x_{ui} - \sum_{i=1}^{n} y_i\}. \tag{13}$$

If the constant α representing the vertex assignment costs is suitably chosen, any solution corresponding to a vertex partition has a larger objective value than a pure packing solution where one of the vertex remains unassigned. It is an easy task to check that $\alpha = n + 1$ fulfills this property. This is the value of α used in our computations.

In his doctoral thesis Macambira [12] conducted a polyhedral investigation of the polytope Q given by the convex hull of feasible solutions of the IP model with the replacements discussed above. One of the results presented there that is relevant to our work refers to the lifting of the inequalities (6). In fact, it is not difficult to see that one such constraint, defined for a proper triple (u, v, i), is satisfied at equality only if ring i is open or, in other words, only if $y_i = 1$. A simple 0-1 lifting of the y_i variable (cf., [15] for an explanation on 0-1 liftings) leads to the stronger inequality

$$x_{ui} + x_{vi} - p_{uvi} - y_i \leq 0. \tag{14}$$

Macambira showed that the polytope Q is full-dimensional whenever $n \geq 2$ and $d_e \leq B$ for all $e \in E$. Moreover, under the same assumptions, he also proved that constraints (5), (7), (12) and (14) are all facet defining for Q.

In the remaining of the text, we shall denote by (PM) the model composed of constraints (2), (3), (12), (5), (14), (7), the integrality constraints (8)–(11) and the objective function (13).

The packing model with less symmetry

Models (GM) and (PM) seen before present a high level of symmetry in the sense that a feasible solution with r rings has $\left(\begin{array}{c} n \\ r \end{array} \right) r!$ integer representations in these formulations since the rings are indistinguishable. It is well-known that the presence of symmetry in the model spoils the performance of enumeration algorithms available for integer programming.

Other studies about symmetry on IP models for combinatorial optimization problems can be found in [11] and [18].

A simple way to reduce the symmetry in model (PM) is to add the constraints

$$y_i \leq y_{i-1}, \qquad \forall i \in N. \tag{15}$$

Obviously, these inequalities are not enough to completely eliminate the symmetry. Nevertheless, as we shall see in section 4, they are very helpful in practice. Other inequalities that we have tried, like those who force the number of vertices to decrease as ring indexes increase, did not work as well as the inequalities in (15). A possible explanation for the success of these inequalities is that they do not increase the density of the LP constraint matrix, which is known to harm the efficiency of the solvers that compute the linear relaxations.

We call (SM) the new model obtained by adding to (PM) the constraints (15).

The compact model

Next we derive a new model for `SRAP` based on a slightly different set of variables. The purpose is to reduce the number of variables while keeping the same number of constraints. We hope that by doing so, we end up with a model which is easier to compute.

In order to describe this alternative formulation, consider a feasible solution for `SRAP` where the vertices of V are partitioned into sets $V_1, \ldots, V_\ell$. For every $j \in [1, \ldots \ell]$, the internal and external demands of ring j are, respectively, D_j and W_j, where

$$D_j = \sum_{u \in V_j} \sum_{\substack{v \in V_j \\ u<v}} d_{uv}, \tag{16}$$

$$W_j = \sum_{u \in V_j} \sum_{\substack{v \notin V_j \\ u<v}} d_{uv}. \tag{17}$$

Let us denote the total demand by D and the total demand of ring j by d_j, i.e.,

$$D = \sum_{(u,v) \in E} d_{uv}, \tag{18}$$

$$d_j = D_j + W_j. \tag{19}$$

According to those definitions, the demand flowing through the federal ring is computed as

$$\overline{D} = D - \sum_{j=1}^{\ell} D_j, \tag{20}$$

and we have that

$$2 \times \overline{D} = \sum_{j=1}^{\ell} W_j. \tag{21}$$

Finally, the capacity constraint on the federal ring implies that

$$\overline{D} \leq B \Longrightarrow D + \overline{D} \leq D + B \Longrightarrow$$

$$\stackrel{(20)}{\Longrightarrow} \overline{D} + \overline{D} + \sum_{j=1}^{\ell} D_j \leq D + B \Longrightarrow$$

$$\stackrel{(21)}{\Longrightarrow} \sum_{j=1}^{\ell} W_j + \sum_{j=1}^{\ell} D_j \leq D + B. \tag{22}$$

Thus, equation (22) gives us another way to express the capacity constraint for the federal ring. It also suggests that we can define new binary variables for the edges in G. For all (u, v) in E, $u < v$, and every $i \in N$, f_{uvi} is set to zero if and only if both vertices u and v are not assigned to ring i. In other words, for a fixed ring i, the edges with $f_{uvi} = 1$ are those which are internal to ring i or link vertices of ring i to vertices at some other ring.

With the variables x and y defined as before, the new IP formulation for SRAP reads

$$(CM) \quad \max \quad \sum_{u \in V} \sum_{i=1}^{n} \alpha x_{ui} - \sum_{i=1}^{n} y_i \tag{13}$$

$$s.t. \quad \sum_{(u,v) \in E} d_{uv} f_{uvi} \leq B, \qquad \forall i \in N, \tag{23}$$

$$\sum_{(u,v) \in E} \sum_{i=1}^{n} d_{uv} f_{uvi} \leq D + B, \tag{24}$$

$$\sum_{i=1}^{n} x_{ui} \leq 1, \qquad \forall u \in V, \tag{12}$$

$$x_{ui} - y_i \leq 0, \qquad \forall u \in V, i \in N, \tag{5}$$

$$f_{uvi} - x_{ui} \geq 0, \qquad \forall (u, v) \in E, \ \forall i \in N, \tag{25}$$

$$f_{uvi} - x_{vi} \geq 0, \qquad \forall (u, v) \in E, \ \forall i \in N, \tag{26}$$

$$f_{uvi} - x_{ui} - x_{vi} \leq 0, \qquad \forall (u, v) \in E, \tag{27}$$

$$y_{i-1} - y_i \leq 0, \qquad \forall i \in \{2, \ldots, n\}, \tag{15}$$

$$x_{ui} \in \{0, 1\}, \qquad \forall u \in V, \ \forall i \in N, \tag{8}$$

$$y_i \in \{0, 1\}, \qquad \forall i \in N, \tag{9}$$

$$f_{uvi} \in \{0, 1\}, \qquad \forall (u, v) \in E, \ \forall i \in N. \tag{28}$$

Let us briefly discuss the model. The objective function is the same as that for the (PM) and (SM) models and, as discussed earlier, if α is taken to be $n+1$, every partition, if there is one, has a more attractive cost than any packing of the vertex set. Constraints (23) and (24) refer to the capacity limitations of local and federal rings, respectively. Constraints (25)-(27) ensure the correct relationship between f and x variables. Finally, constraint (15) is added to the model since, as for the (PM) model, the reduction on symmetry is beneficial for computational purposes.

The set partitioning/knapsack model

Several important problems in combinatorial optimization can be modeled as a *set partitioning* problem (cf., [2]). Quite often, the number of variables in those set partitioning formulations is huge, typically exponential in the instance size. A very successful technique is then to solve these models via column generation methods and branch-and-price algorithms (see [4]). The use of column generation is particularly attractive in set partitioning problems where the subsets of the partition are indistinguishable, like it is the case for `SRAP`, since the model is not affected by symmetry and frequently produce excellent dual bounds. These features motivated us to use a set partitioning formulation and column generation techniques for `SRAP`. However, as we show below, the model is not a pure set partitioning problem since it has one knapsack-type constraint which is necessary to model the federal ring capacity.

Suppose we define a variable λ_j for each subset V_j of V which is feasible, i.e., $D_j+W_j \leq B$ where D_j and W_j are defined as in (16) and (17), respectively. Moreover, for every $u \in V$, let a_{uj} be one if vertex u is in V_j and zero otherwise. Thus, if Λ is the number of feasible subsets of V, `SRAP` can be formulated by a *master problem* of the form

$$(SP) \quad \min \quad \sum_{j=1}^{\Lambda} \lambda_j \tag{29}$$

$$s.t. \quad \sum_{j=1}^{\Lambda} a_{uj}\lambda_j = 1, \qquad \forall u \in V, \tag{30}$$

$$\sum_{j=1}^{\Lambda} d_j\lambda_j \leq D + B, \tag{31}$$

$$\lambda_j \in \{0,1\}, \qquad \forall j \in \{1,\ldots,\Lambda\}. \tag{32}$$

Constraints (30) guarantee that every vertex belongs to one of the subsets taken in the solution. Constraint (31) bounds the demand on the federal ring and can be deduced from (22).

Since Λ grows exponentially with n, column generation technique is appropriate. The *slave subproblem* is obtained as follows. Suppose that the *reduced master* problem has been solved. Let μ_u be the dual variable associated to constraint (30) written for vertex u and π be the dual variable associated to the knapsack constraint in (31). Therefore, the reduced cost of a feasible set V_j is given by

$$\overline{c}_j = 1 - \sum_{u \in V_j} \mu_u - \pi d_j. \tag{33}$$

Since the master is a minimization problem, we want to minimize the reduced cost among all feasible sets V_j. The linear relaxation is solved if this minimum is non negative.

We now give an integer programming formulation for the subproblem. The solution of the subproblem is a feasible set V' of V. We start by defining two sets of binary variables. A variable x_u is defined for all i in V and it takes the value one if and only if u is in

V'. A variable f_{uv} is defined for every edge (u, v) in E and it is set to one if and only if $u \in V$ or $v \in V$. With those variables, the constraints in the formulation of the subproblem are simply those from model (CM) corresponding to a fixed index i in N. Therefore, the subproblem is given by

$$(SSP) \quad \max \quad \sum_{u \in V} \mu_u x_u - \pi \sum_{(u,v) \in E} d_{uv} f_{uv} \tag{34}$$

$$s.t. \quad \sum_{(u,v) \in E} d_{uv} f_{uv} \leq B, \qquad \forall i \in N, \tag{35}$$

$$f_{uv} - x_u \geq 0, \qquad \forall (u, v) \in E, \tag{36}$$

$$f_{uv} - x_v \geq 0, \qquad \forall (u, v) \in E, \tag{37}$$

$$f_{uv} - x_u - x_v \leq 0, \qquad \forall (u, v) \in E, \tag{38}$$

$$x_u \in \{0, 1\}, \qquad \forall u \in V, \tag{39}$$

$$f_{uv} \in \{0, 1\}, \qquad \forall (u, v) \in E. \tag{40}$$

Though in our computational tests we solved (SSP) by pure branch-and-bound, we have also conducted a polyhedral study for (SSP). In principle, the results we found could be used to implement a branch-and-cut for (SSP). However, preliminary experiments completely discouraged us to pursue this idea. Despite that, the polytope P' associated to the convex hull of feasible solutions of (SSP) shares some nice properties with other polytopes studied earlier in the literature. This relationship is briefly described in the next paragraphs.

The basic assumptions of our polyhedral investigation of P' are that G is complete and that every pair of vertices of V form a feasible set. Initially we discovered the dimension and some facet defining inequalities for P'. However, we soon established an important relationship between P' and two other well-studied polytopes related to combinatorial optimization problems, namely the cut polytope and the boolean quadric polytope. To explain this result, let us introduce the linear systems corresponding to the latter polytopes.

For simplicity, we initially discard the knapsack constraint (35) in model (SSP) so that any subset V' of V is feasible. In the CUT problem, we are only interested in representing the edges in the cutset of V' while in the Boolean Quadric (BQ) problem the edges induced by V' are the relevant ones. In the linear system (41)–(47), f' is the set of binary edges corresponding to the cutset and f'' are those corresponding to the induced subgraph both relative to V'. Variables x identify the vertices in V' and, of course, all variables are binary. The convex hull of the integer solutions of the subsystem given by constraints (41)–(44) describe the cut polytope $\mathcal{P}^{\mathcal{C}}$. Similarly, the boolean quadric polytope $\mathcal{P}^{\mathcal{B}}$ can be obtained

from the subsystem (45)–(47).

$$\begin{aligned}
x_u - x_v - f'_{uv} &\leq 0, \quad \forall (i,j) \in E, && (41)\\
x_v - x_u - f'_{uv} &\leq 0, \quad \forall (i,j) \in E, && (42)\\
f'_{uv} + x_u + x_v &\leq 2, \quad \forall (i,j) \in E, && (43)\\
f'_{uv} - x_u - x_v &\leq 0, \quad \forall (i,j) \in E, && (44)\\
f''_{uv} - x_u &\leq 0, \quad \forall (i,j) \in E, && (45)\\
f''_{uv} - x_v &\leq 0, \quad \forall (i,j) \in E, && (46)\\
x_u + x_v - f''_{uv} &\leq 1, \quad \forall (i,j) \in E. && (47)
\end{aligned}$$

Comparing the linear system above with the (SSP) formulation without the knapsack constraint, i.e., the linear system given by (36)–(38), one can notice that the following linear transformations hold

$$\begin{aligned}
f_{uv} + f''_{uv} &= x_u + x_v, && (48)\\
f_{uv} - f''_{uv} &= f'_{uv}, && (49)\\
f'_{uv} &= x_u + x_v - 2f''_{uv}. && (50)
\end{aligned}$$

Equations (48)–(50) put the solutions of `CUT`, `BQ` and (SSP) into a one-to-one correspondence. Moreover, since they are linear, they preserve affine independence and, therefore, every facet defining inequality for one of the polytopes $\mathcal{P}^{\mathcal{C}}$, $\mathcal{P}^{\mathcal{B}}$ or P' is also facet defining for the other two. In fact, equation (49) was introduced in [6] to show that, in some sense, $\mathcal{P}^{\mathcal{C}}$ and $\mathcal{P}^{\mathcal{B}}$ are the same. Therefore, facet defining inequalities for P' can be obtained from facet defining inequalities for $\mathcal{P}^{\mathcal{C}}$or, equivalently, $\mathcal{P}^{\mathcal{B}}$. Results of that type can be found in [3] and [7] for $\mathcal{P}^{\mathcal{C}}$ and [16] for $\mathcal{P}^{\mathcal{B}}$.

Now, if we bring back the knapsack constraint (35) to (SSP) and use (48), we can view the subproblem as a boolean quadric problem with an additional knapsack constraint. The polytope associated to the convex hull of this problem is studied by Mehrotra in [14] in the special case where the knapsack constraint corresponds to a cardinality constraint. Using the linear transformations above, we found out that several facet defining inequalities we discovered for (SSP) could be derived from the results of Mehrotra.

Model sizes

To conclude this section we summarize in table 1 the total number of variables and constraints in each of the models presented so far. Clearly, the all have a number of variables that is polynomial in the size of the input graph.

3 A branch-and-price algorithm for `SRAP`

In this section we discuss the essential ingredients of the branch-and-price algorithm that we implemented for `SRAP`. The design of our code was deeply driven by [4] and [19]. We restrict our discussion to the strategical level of the algorithm, namely we describe our choices for

Model	# variables	# constraints
(GM)	$n^2+n+3mn$	$n^2+2n+3mn+1$
(PM)	$n^2+n+3mn$	$n^2+2n+3mn+1$
(SM)	$n^2+n+3mn$	$n^2+3n+3mn$
(CM)	n^2+n+mn	$n^2+3n+3mn$
(SP)	exponential	$n+1$

Table 1: Model sizes.

the set of initial columns, the selection of nodes to explore, the branching scheme, the detection of tailing off effect and the inclusion or not of multiple generated columns. These choices are detailed below.

The initial set of columns

In order to apply column generation for model (SP), we must have an initial set of columns which form a basis for the linear relaxation. However, it is not easy in general to find such a set. Instead, we decide to add columns corresponding artificial variables each of which with a high cost aiming that, in the end, they will not appear in an optimal solution. If any of these columns is present in the optimal integer solution then we conclude that the SRAP instance is infeasible. Such a set of initial columns can be constructed in the following manner.

For every $u \in V$, construct a column of the form $\begin{pmatrix} e_u^n \\ 0 \end{pmatrix}$ where e_u^n is the vector of $\mathbb{R}^n$ with one in component u and zero elsewhere. Notice that this column does not actually correspond to an assignment where vertex u is the only vertex in a ring. If it did, the last component of the vector, that is, the one corresponding to the constraint (31), should take value $\sum_{(u,v)\in E} d_{uv}$. Hence, this artificial column receives a cost of $n+1$, which is large enough so that, if it is part of an integer optimal solution, this solution has cost larger than any feasible partition of the vertices. Now, the n columns built in that way, together with the column corresponding to the slack variable of constraint (31) form a basis.

Besides those columns, we also populate the initial matrix of the *reduced master* with randomly generated columns. This is done until $5n+1$ distinct columns have been added, including the artificial ones. The algorithm is given in figure 1. Notice that, in principle, it may run into an endless loop. However, this has never happened in our experiments where, typically, a few seconds of computation proved to be enough to generate the $5n+1$ distinct columns. If this was not the case, the loop could be avoided easily by fixing some limit on the execution time of the algorithm.

Node selection and branching rules

In our implementation the next to node to explore during the enumeration procedure is selected as the one with the best dual bound. The branching rule adopted is the one

```
procedure Generate-initial-columns;
1.  A ← ∅;
2.  while |A| < 4n + 1 do
3.      generate a random number t in N;
4.      randomly select t different vertices in V;
        Let S be the set formed by those vertices;
5.      D(S) ← Σ_{(u,v)∈G[S]} d_uv;
6.      W(S) ← Σ_{(u,v)∈δ(S)} d_uv;
7.      while D(S) + W(S) > B do
8.          select randomly a vertex u in S;
9.          remove u from S;
10.     end while
11.     Let (a, D(S) + W(S)) be the column corresponding to S;
12.     if a ∉ A, A ← A ∪ {a};
13. end while
end Generate-initial-columns.
```

Figure 1: Algorithm for generating a set of initial columns.

proposed by Ryan and Foster [17] for set partitioning problems. This rule is based on the following proposition.

Proposition 3.1 *[17] If A is a 0–1 matrix and λ^* is a fractional solution of the system $A\lambda = 1$, then there exist two equations u and v in that system such that*

$$0 < \sum_{k|a_{uk}=a_{vk}=1} \lambda_k < 1. \tag{51}$$

From Proposition 3.1, Ryan and Foster suggested to add the following branching constraints

$$\sum_{k|a_{uk}=a_{vk}=1} \lambda_k = 0 \tag{52}$$

and

$$\sum_{k|a_{uk}=a_{vk}=1} \lambda_k = 1. \tag{53}$$

In the case of `SRAP`, the constraints above give rise to two auxiliary subproblems. In the first one vertices u and v must necessarily be in different rings while in the second one they must be assigned to the same ring. It is well-know that this branching rule lead to more balanced enumeration trees which, in turn, tend to improve the performance of branch-bound algorithms.

Primal bounding

Our branch-and-price algorithm always starts by running the `GRASP` heuristic described in the appendix. If the heuristic is successful, the incumbent value is the cost of the feasible

solution produced by GRASP. Otherwise, the incumbent is initialized as $2\lceil\frac{D}{B}\rceil$, where D is defined as in (18). Clearly, in the latter case, the incumbent value is a guess. If no solution is found with cost better than that value, the algorithm is executed again with the incumbent value set to $n+1$. No such rerun of the code was necessary in any of the feasible instances in our dataset.

Tailing off detection and multiple column generation

The solution of the linear relaxation in a node of the enumeration tree is done via column generation. It may occur that the value of the linear relaxation remains practically unchanged even after several columns have been brought to the reduced master. This phenomenon is known as *tailing off* and impairs the efficiency of the algorithm. To avoid spending too much time generating useless columns and solving LPs, we adopt the following policy to detect tailing off. If after 50 consecutive computations of the reduced master problem the value of the linear relaxation improved by less than 0.1%, a branching is done at the current node.

Another important aspect of column generation is how one solve the subproblem. We tested two ways to generate the columns, one heuristic and the other exact. The heuristic generation was outperformed by the exact one and so we abandon this alternative. The exact method consists in solving the subproblem using a standard LP-branch-and-bound code. In this case, the IP problem to be solved is composed of the constraints in the model (SSP) from the previous section and the constraints required by the branchings that led to the current node. Later in subsection 4.3 we report the results we obtained when comparing the strategy where we add just the column corresponding to the optimal solution against that where we add all columns with negative reduced cost that are encountered by the branch-and-bound code during the solution of the subproblem.

4 Computational experiments

In this section we describe our computational experiments with the different IP models proposed in section 2. All tests were run on a PC desktop equipped with a 450 MHz AMD K6 processor and 256 Mbytes of RAM under Linux operating system. Linear programming relaxations were solved using XPRESS version 12.05 ([5]) and the branch-and-price algorithm of section 3 was implemented via the XOSL callable library of XPRESS. All runs were limited to one hour of CPU time.

4.1 Instances

The set of instances is divided into three categories. Instances in category C1 are taken from [9] and correspond to instances type 1 and 2 in that paper. Those where the instances that the authors proved to be feasible. We denote by C1a the subset of the instances of C1 whose graphs have no more than 25 vertices. Category C2 is composed of all instances tested in [1] excluding those already in C1. Finally, category C3, includes the instances in table 14 of

[9] whose feasibility status could not be proved with the algorithms proposed in that paper. Actually, in [9], the authors conjectured that the instances in `C3` were infeasible.

In total, there are 111 instances in `C1`, 56 in `C1a`, 230 in `C2` and 12 in `C3`. Within each category, instances are divided into *geometric* and *random* ones and, then, further divided into those having *low* and *high* ring capacities, respectively, 155 and 622 `Mbs`. These characteristics of an instance can be deduced from its name. Instance names always start with two letters. The first letter is either "g" or "r" meaning that the instance belongs to the geometric or the random subdivision, respectively. The second letter is either "l" or "h", depending if the ring capacity is 155 or 622 `Mbs`, respectively. For more details on how those instances were generated and their properties, we refer the interested reader to the original papers where they were introduced.

4.2 Comparison between models (GM), (PM), (SM) and (CM)

To compare the performance of the models (GM), (PM), (SM) and (CM) discussed in section 2 we use the instances in class `C1a`. Graphs in those instances have no more than 25 vertices and our experiments shown that running `XPRESS` for larger graphs with the (GM) model was too time consuming. The results are given in tables 2 and 3. In these tables, the first column contains the instance names and column z^* displays the optimal value. For each model there are four additional columns: "#nodes" with the number of nodes explored in the enumeration tree, "z" with the best dual bound reached during the computation of the model, "Δ" representing the relative gap between the dual bound and the optimal value and, finally, "time" with the `CPU` time in seconds necessary to solve the model or to halt the execution after one hour of computation.

Inspecting first the instances solved by all four models, one sees that model (CM) ran faster than all its opponents in 24 out of the 25 cases. The exception was instance `rl_15_10` that was computed in less time by (SM). Now, if we look at the number of unsolved instances, we obtain that models (GM), (PM), (SM) and (CM) failed to compute 29, 23, 22 and 22 instances, respectively. Moreover, the duality gap for unsolved instances is typically smaller for (CM). These numbers illustrate quite well the advantages of (CM) over the other candidate formulations. However, the comparison between pairs of models is also useful and some cases are discussed below. When comparing models M and M', we denote by *successful* those instances which are solved with both models. The instances solved only by M (M') are named M (M') *successful*.

Let us first analyze the results for models (PM) and (SM). We see that for the *successful* instances (SM) was faster then (PM) in 20 cases while the inverse only occurred in 8 cases. The number of (SM) *successful* instances is 4 and that of (PM) *successful* instances is 3. This shows that the inequalities reducing the symmetry improve the performance of **B&B**.

The direct comparison between (GM) and (SM), shows that (SM) outperforms (GM) in 15 cases and is beaten by (GM) in 10 cases. The number of (GM) and (SM) *successful* instances is 1 and 7 respectively. Thus, the packing model with reduced symmetry is more effective than the original model of Goldschmidt et al for `SRAP`.

Finally, to conclude the comparison between the polynomial-sized models, we compare (GM) and (CM). This comparison allows us to assess the quality of one of our models with

		Model (GM)				Model (PM)				Model (SM)				Model (CM)			
Instance	z^*	#nodes	z	Δ (%)	time	#nodes	z	Δ (%)	time	#nodes	z	Δ (%)	time	#nodes	z	Δ (%)	time
gl_15_1	3	7208	3	0	424	14143	3	0	1747	5532	3	0	329	1341	3	0	45
gl_15_4	3	7317	3	0	391	1970	3	0	108	2225	3	0	78	241	3	0	7
gl_15_7	3	7450	3	0	411	19094	3	0	3064	7479	3	0	655	10655	3	0	190
gl_15_9	3	310600	2	33	3600	10968	3	0	755	2181	3	0	118	2246	3	0	17
gl_25_1	4	5300	1.32	67	3600	4500	1	75	3600	19400	2	50	3600	17000	2	50	3600
gl_25_3	3	7300	1.29	57	3600	8300	2	33	3600	15600	2	33	3600	33400	2	50	3600
gl_25_4	4	6900	1.33	67	3600	8800	1	75	3600	8600	2	50	3600	12500	2	50	3600
gl_25_7	3	6600	1.28	57	3600	7649	3	0	1929	9800	2	33	3600	33400	2	33	3600
gl_25_8	4	6500	1.35	66	3600	11900	1	75	3600	17000	1.19	70	3600	9100	2	50	3600
gh_15_1	3	46806	3	0	1457	2134	3	0	442	976	3	0	120	3042	3	0	36
gh_15_2	3	19400	2	33	3600	737	3	0	224	370	3	0	75	290	3	0	16
gh_15_3	2	223	2	0	3	300	2	0	10	109	2	0	7	34	2	0	1
gh_15_6	2	96	2	0	2	299	2	0	9	612	2	0	13	45	2	0	1
gh_15_7	2	111	2	0	1	57	2	0	2	89	2	0	1	5	2	0	1
gh_15_8	3	7047	3	0	1867	8100	2	33	3600	5850	3	0	1295	1072	3	0	50
gh_15_9	3	20324	3	0	1658	4493	3	0	778	3025	3	0	405	387	3	0	7
gh_15_10	2	124	2	0	1	258	2	0	3	69	2	0	4	33	2	0	1
gh_25_1	3	5000	1.24	59	3600	2200	1.46	51	3600	2301	3	0	1051	20541	3	0	3455
gh_25_2	2	3261	2	0	679	1810	2	0	858	3900	2	0	3600	298	2	0	14
gh_25_3	2	2162	2	0	362	3000	2	0	3600	4404	2	0	2360	47	2	0	5
gh_25_4	3	4700	1.27	58	3600	1579	3	0	1185	4000	2	33	3600	1996	3	0	564
gh_25_7	4	3100	1.32	67	3600	7200	1	75	3600	2000	2	50	3600	8700	2	50	3600
gh_25_8	3	5100	1.27	58	3600	1900	1	68	3600	5100	2	33	3600	3285	3	0	1024
gh_25_9	3	4600	1.27	58	3600	1400	1	58	3600	6500	2	33	3600	21500	2	33	3600
rl_15_1	3	11671	3	0	275	8810	3	0	701	1446	3	0	36	1694	3	0	14
rl_15_3	2	170	2	0	1	1043	2	0	6	837	2	0	6	116	2	0	1
rl_15_4	3	5449	3	0	230	9440	3	0	1059	3191	3	0	222	74	3	0	3
rl_15_6	3	14729	3	0	324	3890	3	0	314	3349	3	0	153	1005	3	0	22

Table 2: Comparison (GM) (PM), (SM) and (CM) models for `C1a` instances.

		Model (GM)				Model (PM)				Model (SM)				Model (CM)			
Instance	z^*	#nodes	z	Δ (%)	time	#nodes	z	Δ (%)	time	#nodes	z	Δ (%)	time	#nodes	z	Δ (%)	time
rl_15_8	3	19342	3	0	945	9357	3	0	752	1115	3	0	138	141	3	0	3
rl_15_9	3	24793	3	0	638	10474	3	0	779	2884	3	0	106	150	3	0	2
rl_15_10	3	11663	3	0	907	6849	3	0	981	3355	3	0	74	2473	3	0	109
rl_25_2	3	10000	2	33	3600	17100	2	33	3600	10100	2	33	3600	11667	3	0	306
rl_25_4	4	5500	1.32	67	3600	10600	1	75	3600	17500	2	50	3600	9500	2.04	49	3600
rl_25_5	4	10100	1.28	68	3600	8000	1	75	3600	11700	1.16	71	3600	35800	2.04	49	3600
rl_25_6	4	7000	1.31	67	3600	9200	2	50	3600	11100	1.19	70	3600	8500	2.04	49	3600
rl_25_7	4	7300	1.31	67	3600	14300	2	50	3600	13800	1.09	73	3600	13200	2	50	3600
rl_25_8	3	12600	2	33	3600	14300	2	33	3600	17676	3	0	2793	747	3	0	95
rl_25_9	4	10300	1.33	68	3600	7000	1	75	3600	14800	2	50	3600	12400	2	50	3600
rl_25_10	4	9100	1.29	68	3600	11800	2	50	3600	19000	2	50	3600	11000	2.04	49	3600
rh_15_1	3	20188	3	0	1162	1648	3	0	247	1299	3	0	131	2752	3	0	43
rh_15_4	2	108	2	0	1	60	2	0	2	794	2	0	9	21	2	0	1
rh_15_5	3	29960	3	0	1150	351	3	0	57	740	3	0	110	3470	3	0	52
rh_15_6	3	37300	3	0	1460	990	3	0	129	2130	3	0	332	1926	3	0	33
rh_15_7	2	134	2	0	1	224	2	0	5	240	2	0	3	39	2	0	1
rh_15_8	2	1166	2	0	8	774	2	0	68	566	2	0	30	89	2	0	1
rh_15_9	3	32886	2	0	1105	3592	3	0	502	2478	3	0	206	3376	3	0	41
rh_25_1	3	3800	1.26	58	3600	4200	1	67	3600	6600	2	33	3600	2600	2	33	3600
rh_25_2	3	6500	1.22	59	3600	1789	3	0	1049	1859	3	0	743	25300	2	33	3600
rh_25_3	3	7000	1.23	59	3600	1428	3	0	694	3252	3	0	1377	39800	2	33	3600
rh_25_4	3	4700	1.27	58	3600	6400	1	68	3600	12100	2	33	3600	10500	2	33	3600
rh_25_5	4	3100	1.31	67	3600	5100	1	75	3600	2500	2	50	3600	4900	2	50	3600
rh_25_6	4	4700	1.28	68	3600	5000	2	50	3600	5700	2	50	3600	10700	2	50	3600
rh_25_7	3	7000	1.22	59	3600	1012	3	0	581	5285	3	0	2899	28000	2	33	3600
rh_25_8	4	4000	1.30	68	3600	2000	1	75	3600	11300	1.67	58	3600	14500	2.04	49	3600
rh_25_9	2	374	2	0	21	277	2	0	60	748	2	0	140	28	2	0	1
rh_25_10	3	7900	1.21	60	3600	764	3	0	504	478	3	0	125	89800	2	33	3600

Table 3: Comparison (GM) (PM), (SM) and (CM) models for `C1a` instances (cont.).

respect to an earlier formulation presented in the literature. In this case, one sees that (CM) performs better in all 24 *successful* instances and there are no (GM) *successful* instances and 6 (CM) *successful* instances. These results demonstrate that (CM) is the undisputed winner against (GM). The benefits of using (CM) rather than the other formulations become even more apparent when one inspects the solution times required by the different algorithms. In most cases, the computing time for the (CM) model is orders of magnitude faster than those for the other formulations.

4.3 Single $\times$ Multiple column generation in the B&P algorithm

Before comparing the (SP) model with the previous ones, we first report our experiments to verify whether the generation of multiple columns with negative reduced cost improves the performance of the **B&P** algorithm. As mentioned earlier, preliminary results discouraged the heuristic generation of columns. However, an alternative way to generate suboptimal columns is to store all those having negative reduced costs which are found by the pure branch-and-bound that solves the subproblem. Table 4 presents the results obtained by the two strategies for generating columns, namely, the generation of a single or of multiple columns per subproblem. In this table, column "#initial" represents the number of columns in the initial reduced master problem. For each strategy, there are four columns: "#nodes" with the number of nodes explored in the enumeration tree, "#sub" with the number of subproblems solved, "#cols" with the number of columns generated and, finally, "time" with the time (in seconds) necessary to solve that particular instance. The twenty instances that took part in this test were picked randomly in classes `C1` and `C2`. For all of them, the multiple column generation strategy result in a faster **B&P** algorithm in which less subproblems needed to be solved. Thus, we decided to use this strategy in our final runs.

4.4 Comparison between models (CM) and (SP)

We are now ready to compare the performance of the **B&P** algorithm when computing the (SP) model against that of standard branch-and-bound algorithms when applied to the other models. The instances used in these tests are those of `C1a` that were solved to optimality with the (CM) model, the best formulation to compute `SRAP` instances so far. The **B&P** algorithm also computed the (SP) model to optimality for those instances and, therefore, we focus our attention only on the `CPU` times needed by both algorithms. The results are summarized in table 5. The speedups are measured as the ratio between the `CPU` time of the branch-and-bound algorithm and that of the **B&P** algorithm. Disregarding instances that are solved by **B&B** in less than 10 seconds, the average speedup is 95%. This shows that **B&P** is, by far, more efficient than **B&B**. In fact, the results are so good that one may wonder if the limitations pointed in [9] relative to the performance of IP models and algorithms for `SRAP` cannot be overcame with this approach. This possibility is further investigated in the next subsection.

Instance	#initial	One column				Multiple columns			
		#nodes	#sub	#cols	time	#nodes	#sub	#cols	time
gl_15_1	76	1	22	21	0.72	1	13	24	0.48
gl_15_4	76	1	26	25	0.72	1	21	33	0.62
gl_15_7	76	1	34	33	1.24	1	13	27	0.45
gl_15_9	76	1	34	33	0.98	1	18	35	0.63
gl_25_1	126	1	68	67	7.01	1	42	106	4.78
gl_25_3	126	1	87	86	4.93	1	54	127	3.33
gl_25_4	126	1	95	94	6.09	1	53	102	3.56
gl_25_7	126	1	102	101	4.76	1	51	128	3.49
gl_25_8	126	1	107	106	8.79	1	46	127	4.17
rl_50_1	251	1	255	254	27.33	1	153	519	21.02
rl_50_5	251	3	204	203	19.38	3	153	717	17.17
rl_50_6	251	1	204	203	17.54	1	153	453	15.75
rh_30_1	151	1	153	152	12.56	1	102	284	8.41
rh_30_2	151	1	139	138	40.89	1	61	195	23.29
rh_30_4	151	1	146	145	23.46	1	68	200	12.81
rh_30_6	151	1	141	140	33.31	1	68	192	19.32
rh_30_7	151	1	127	126	26.70	1	57	179	13.91
rh_30_8	151	1	136	135	22.58	1	56	191	9.95
rh_30_9	151	1	131	130	20.31	1	58	189	9.41
rh_30_10	151	1	102	101	13.68	1	54	197	10.81

Table 4: Comparison of strategies to generate columns in the **B&P** algorithm.

4.5 Results of the B&P algorithm for C1 and C2 instances

We discuss now the results obtained by the **B&P** algorithm and the (SP) formulation for the instances in classes C1 and C2. They are displayed in tables 6 to 10. The first of these tables contains data from C1a instances, the second contains data from instances in C1\C1a and the remaining three have data from C2 instances. In these tables the symbols "$*$" and "+" are amended to instance names whenever the optimal solution of the linear relaxation at the root node is integral or our GRASP heuristic is unable to produce an initial feasible solution, respectively. The first column contains the instance name and the subsequent columns contain, respectively: the total number of nodes explored, the number of subproblems solved, the number of initial columns, the number of columns generated during the enumeration, the upper bound computed by GRASP, the lower bound at the root node, the optimal value and the CPU time (in seconds). The analysis that follows considers separately the geometric and the random instances.

In two of the 50 geometric instances in class C1 the solution obtained by the linear relaxation corresponds to an integer optimum. In all remaining cases, the optimal value of the linear relaxation when rounded up was equal to the integer optimum. In this situation, because the coefficients in the objective function are integral, the **B&P** algorithm can stop if a primal solution is known whose cost is no more than one unit apart from the dual bound. Also notice that only in 11 instances the algorithm had to resort to branching. Moreover, the computational times are orders of magnitude smaller than those required by

Instance	speedup (**B&B**/ **B&P**)
gl_15_1	93.75
gl_15_4	11.29
gl_15_7	422.22
gl_15_9	26.94
gh_15_1	30.51
gh_15_2	9.30
gh_15_3	1.67
gh_15_6	1.29
gh_15_7	1.75
gh_15_8	23.36
gh_15_9	6.25
gh_15_10	1.37
gh_25_1	621.40
gh_25_2	3.46
gh_25_3	0.78
gh_25_4	74.90
gh_25_8	148.62
rl_15_1	16.67
rl_15_3	1.72
rl_15_4	2.16
rl_15_6	29.73
rl_15_8	2.97
rl_15_9	2.94
rl_15_10	110.10
rl_25_2	105.88
rl_25_8	36.68
rh_15_1	23.12
rh_15_4	0.99
rh_15_5	33.12
rh_15_6	21.02
rh_15_7	0.83
rh_15_8	0.59
rh_15_9	50.00
rh_25_9	0.20

Table 5: Comparison between (SP) and (CM) computing times.

branch-and-bound algorithms and the previous models. All instances were solved in less than a minute and 80% of them were solved in less than 15 seconds.

The scenario observed above for geometric instances is not altered if we consider the 61 random instances in class `C1`. Again we have two instances whose optimal integer solution was computed by simply solving the linear relaxation. In the remaining instances, rounding up the optimal values of the relaxation produces the optimal integer value. Only in 6 instances the algorithm had to resort to branching, however, no more than 3 nodes were explored in any case. The maximum `CPU` time once again remained significantly smaller than that for the **B&B** algorithms and the other models. The maximum time spent to prove optimality only exceeded one minute for instance `rh_50_1` and, even in this case, by only a few seconds. In 90% of the random instances in `C1`, the computation time stayed below 30 seconds.

We now turn our attention now to the instances in class `C2`. According to Aringhieri and Dell'Amicco ([1]) these instances were generated to be "hard" ones to solve.

In 10 out of the 140 geometric instances tested in class `C2` the optimal solution of the (SP) linear relaxation was integer. Once again, by rounding up the optimal bounds from the linear relaxations of the remaining instances, one gets the optimal integer values. Nevertheless, it is interesting to notice that 65 instances required the exploration of multiple nodes of the enumeration tree. This occurred more often for larger graphs. As an example, instance `gl_50_6.3` forced the algorithm to visit 27 nodes to have its optimality proven. The higher degree of difficulty of these instances when compared with those in `C1` can also be assessed from the slight increase in computation times. However, they still remain quite small since 104 instances were solved in no more than a minute and 69 out of them were computed in less than 15 seconds.

The random instances in `C2` seem to be easier to solve than the geometric ones. There are 90 instances in this subclass in total and, for 17 among them, the optimal solution of the linear relaxation of (SP) is integral. Rounding can be used once again to reach the optimal integral values in all remaining instances. The maximum number of nodes explored in the enumeration tree was 5 and in 62 instances only the root node was necessary to conclude that an optimal solution was at hand. Only instance `rl_50_2.2` required more than a minute of computation and 63 other instances required less than 15 seconds of `CPU` to halt the search.

4.6 Dealing with infeasible instances

The inability of IP models and algorithms to detect infeasibility for `SRAP` was pointed earlier in the literature. In [1] the authors mention that 57 hours of `CPU` were necessary on average to prove infeasibility using a PC Pentium III/600 with 512 MB of RAM and running under Linux operating system. We had similar experiences even when using **B&B** with the (CM) formulation. Thus, we decide to investigate further the capabilities of the **B&P** algorithm by testing its performance on possibly infeasible instances. To this end, we ran **B&P** on the 12 instances in class `C3` which were conjectured to be infeasible in [9]. In these tests, even when the instances were later proved to be feasible, the starting lower bound was set to $2\lceil\frac{D}{B}\rceil$. As explained in section 3, this value was used to initialize the

Instance	#nod	#sub	#ini	#gen	z_{ub}	z_r	z^*	time
gl_15_1	1	13	76	24	3	2.83	3	0.48
gl_15_4	1	21	76	33	3	2.67	3	0.62
gl_15_7(∗)	1	13	76	27	3	3.00	3	0.45
gl_15_9	1	18	76	35	3	2.22	3	0.63
gl_25_1	1	42	126	106	4	3.39	4	4.78
gl_25_3	1	54	126	127	3	2.93	3	3.33
gl_25_4	1	53	126	102	4	3.25	4	3.56
gl_25_7	1	51	126	128	3	2.56	3	3.49
gl_25_8	1	46	126	127	4	3.96	4	4.17
gh_15_1	1	23	76	53	3	2.14	3	1.18
gh_15_2	1	20	76	33	3	2.33	3	1.72
gh_15_3	1	18	76	32	2	1.86	2	0.60
gh_15_6	1	26	76	53	2	1.75	2	0.77
gh_15_7	1	30	76	46	2	1.50	2	0.57
gh_15_8	1	21	76	58	3	2.85	3	2.14
gh_15_9	1	18	76	29	3	2.39	3	1.12
gh_15_10	1	31	76	55	2	1.59	2	0.73
gh_25_1	1	55	126	149	3	2.30	3	5.56
gh_25_2	3	77	126	194	3	1.95	2	4.05
gh_25_3	1	92	126	226	2	1.95	2	6.43
gh_25_4	1	63	126	136	3	2.71	3	7.53
gh_25_7	1	47	126	129	4	3.45	4	9.54
gh_25_8	1	61	126	156	3	2.56	3	6.89
gh_25_9	1	60	126	150	3	2.60	3	6.10
rl_15_1	1	21	76	42	3	2.36	3	0.84
rl_15_3	1	24	76	44	2	1.73	2	0.58
rl_15_4	1	23	76	41	3	2.85	3	1.39
rl_15_6	1	24	76	49	3	2.25	3	0.74

Instance	#nod	#sub	#ini	#gen	z_{ub}	z_r	z^*	time
rl_15_8	1	22	76	39	3	2.50	3	1.01
rl_15_9	1	21	76	36	3	2.30	3	0.68
rl_15_10	1	20	76	40	3	2.95	3	0.99
rl_25_2	1	44	126	126	3	2.84	3	2.89
rl_25_4	1	38	126	108	4	3.28	4	4.10
rl_25_5	1	40	126	121	4	3.04	4	2.64
rl_25_6	1	47	126	121	4	3.31	4	5.09
rl_25_7	1	32	126	101	4	3.40	4	3.63
rl_25_8	1	47	126	121	3	2.57	3	2.59
rl_25_9	1	50	126	138	4	3.34	4	5.16
rl_25_10	1	44	126	130	4	3.29	4	4.22
rh_15_1	1	22	76	49	3	2.37	3	1.86
rh_15_4	1	25	76	48	2	1.65	2	1.01
rh_15_5	1	23	76	49	3	2.23	3	1.57
rh_15_6	1	19	76	42	3	2.18	3	1.57
rh_15_7	1	27	76	49	2	1.74	2	1.20
rh_15_8(∗)	1	26	76	47	2	2	2	1.67
rh_15_9	1	15	76	34	3	2.13	3	0.82
rh_25_1	1	44	126	132	3	2.95	3	6.71
rh_25_2	1	50	126	154	3	2.25	3	6.90
rh_25_3	1	53	126	147	3	2.27	3	6.37
rh_25_4	1	40	126	138	3	2.99	3	10.62
rh_25_5	1	45	126	148	4	3.29	4	11.11
rh_25_6	3	42	126	134	4	2.96	4	7.12
rh_25_7	1	70	126	185	3	2.23	3	11.52
rh_25_8	1	45	126	139	4	3.11	4	10.58
rh_25_9	1	89	126	204	2	1.58	2	4.94
rh_25_10	1	64	126	167	3	2.06	3	7.41

Table 6: Results from **B&P** algorithm for `C1a` instances.

Instance	#nod	#sub	#ini	#gen	z_{ub}	z_r	z^*	time
gl_30_1	1	71	151	195	4	3.56	4	6.67
gl_30_2	1	90	151	205	4	3.43	4	7.81
gl_30_3	1	84	151	198	4	3.25	4	8.96
gl_30_4	1	81	151	195	4	3.10	4	6.79
gl_30_5(∗)	1	109	151	242	3	3.00	3	9.92
gl_30_6	1	77	151	182	4	3.13	4	7.09
gl_30_7	1	65	151	161	4	3.26	4	10.13
gl_30_8	1	68	151	167	4	3.44	4	8.96
gl_30_9	1	75	151	186	4	3.15	4	5.87
gl_50_1	3	153	251	440	6	4.61	5	24.29
gl_50_4	1	153	251	473	6	5.06	6	45.57
gl_50_7	1	136	251	414	5	4.42	5	46.29
gl_50_8	3	153	251	465	6	4.84	5	29.43
gl_50_9	1	153	251	373	4	3.57	4	15.04
gh_30_1	1	87	151	229	3	2.61	3	8.58
gh_30_2	1	63	151	161	4	3.25	4	14.19
gh_30_3	1	49	151	147	4	3.40	4	9.42
gh_30_4	1	116	151	267	3	2.75	3	18.51
gh_30_5	1	51	151	143	4	3.30	4	10.01
gh_30_9	1	62	151	170	4	3.14	4	10.46
gh_30_10	1	102	151	270	3	2.19	3	8.26
gh_50_1	3	153	251	459	6	4.53	5	43.31
gh_50_4	3	153	251	458	5	3.94	4	35.04
gh_50_5	1	153	251	464	5	4.21	5	37.03
gh_50_6	1	153	251	459	5	4.71	5	44.85
gh_50_7	1	153	251	495	5	4.38	5	42.59
rl_30_1	1	66	151	174	3	2.74	3	5.73
rl_30_4	1	69	151	180	4	3.79	4	7.34

Instance	#nod	#sub	#ini	#gen	z_{ub}	z_r	z^*	time
rl_30_5	1	73	151	201	4	3.06	4	7.43
rl_30_7	1	70	151	198	3	2.92	3	7.69
rl_30_8	1	62	151	190	4	3.17	4	10.48
rl_30_10	1	76	151	215	4	3.50	4	8.41
rl_50_1	1	153	251	519	5	4.22	5	21.02
rl_50_3	3	153	251	492	5	3.81	4	21.10
rl_50_4	3	153	251	515	5	3.83	4	25.72
rl_50_5	3	153	251	717	5	3.65	4	17.17
rl_50_6	1	153	251	453	4	3.62	4	15.75
rl_50_7	1	130	251	439	5	4.35	5	27.98
rl_50_10	3	153	251	481	5	3.68	4	22.81
rh_30_1	1	102	151	284	3	2.09	3	8.41
rh_30_2	1	61	151	195	4	3.12	4	23.29
rh_30_3	1	88	151	262	3	2.17	3	9.34
rh_30_4	1	68	151	200	3	2.48	3	12.81
rh_30_6	1	68	151	192	4	3.38	4	19.32
rh_30_7	1	57	151	179	4	3.12	4	13.91
rh_30_8	1	56	151	191	3	2.89	3	9.95
rh_30_9	1	58	151	189	3	2.74	3	9.41
rh_30_10(∗)	1	54	151	197	4	3.00	3	10.81
rh_50_1	1	153	251	559	4	3.54	4	69.65
rh_50_2	3	153	251	609	4	2.78	3	32.46
rh_50_5	1	153	251	468	3	2.63	3	25.46
rh_50_6	1	153	251	584	3	2.67	3	32.79
rh_50_8	1	153	251	576	4	3.43	4	41.05
rh_50_9	1	153	251	569	4	3.17	4	37.15
rh_50_10	1	134	251	476	5	3.89	4	57.94

Table 7: Results from **B&P** algorithm for `C1` \ `C1a` instances.

Instance	#nod	#sub	#ini	#gen	z_{ub}	z_r	z^*	time
gl_15_3.1	1	14	76	33	4	3.28	4	0.41
gl_15_3.2	1	12	76	32	4	3.18	4	0.48
gl_15_3.3	1	13	76	30	4	3.25	4	0.39
gl_15_3.4	1	29	76	49	4	3.19	4	1.16
gl_15_3.5	1	15	76	27	4	3.20	4	0.45
gl_15_3.6	1	12	76	32	4	3.18	4	0.48
gl_15_3.7	1	29	76	49	4	3.19	4	1.18
gl_15_3.8	1	13	76	30	4	3.25	4	0.42
gl_15_3.9	1	15	76	27	4	3.20	4	0.45
gl_15_3.10	1	14	76	33	4	3.29	4	0.40
gl_15_6.1(∗)	1	20	76	39	3	3.00	3	1.14
gl_15_6.2(∗)	1	23	76	50	3	3.00	3	1.15
gl_15_6.3(∗)	1	19	76	38	3	3.00	3	0.92
gl_15_6.4(∗)	1	20	76	51	3	3.00	3	0.89
gl_15_6.5	1	21	76	42	3	2.98	3	0.90
gl_15_6.6	1	21	76	49	3	2.94	3	1.04
gl_15_6.7(∗)	1	25	76	52	3	3.00	3	1.22
gl_15_6.8(∗)	1	11	76	29	3	3.00	3	0.44
gl_15_6.9(∗)	1	12	76	32	3	3.00	3	0.39
gl_15_6.10(∗)	1	12	76	24	3	3.00	3	0.55
gl_25_9.1	1	40	126	120	4	3.61	4	2.11
gl_25_9.2	1	43	126	107	4	3.38	4	2.76
gl_25_9.3	1	54	126	154	4	3.42	4	4.24
gl_25_9.4	1	28	126	92	4	3.42	4	1.82
gl_25_9.5	1	45	126	116	4	3.33	4	3.32
gl_25_9.6	1	35	126	120	4	3.36	4	2.73
gl_25_9.7	1	42	126	130	4	3.97	4	1.79
gl_25_9.8	1	48	126	133	4	3.37	4	3.04
gl_25_9.9	1	49	126	133	4	3.33	4	3.14
gl_25_9.10	1	40	126	115	4	3.33	4	2.25
gl_25_10.1	1	60	126	118	5	4.29	5	6.39
gl_25_10.2	1	63	126	126	5	4.28	5	5.64
gl_25_10.3	1	65	126	141	5	4.31	5	6.72
gl_25_10.4	1	63	126	126	5	4.29	5	5.61
gl_25_10.5	1	59	126	144	5	4.32	5	5.22
gl_25_10.6	1	39	126	90	5	4.27	5	3.51
gl_25_10.7	1	52	126	119	5	4.90	5	2.85
gl_25_10.8	1	46	126	96	5	4.29	5	3.57
gl_25_10.9	1	62	126	143	5	4.29	5	5.36
gl_25_10.10	1	65	126	141	5	4.31	5	6.64
gl_30_10.1	1	53	151	129	5	4.74	5	5.06
gl_30_10.2	3	82	151	217	6	4.59	5	7.22
gl_30_10.3	1	69	151	178	5	4.49	5	7.39
gl_30_10.4	1	68	151	159	5	4.61	5	3.99
gl_30_10.5	3	68	151	189	6	4.47	5	6.80
gl_30_10.6	1	59	151	172	5	4.49	5	6.02
gl_30_10.7	1	60	151	153	5	4.50	5	4.63
gl_30_10.8	1	81	151	200	5	4.48	5	10.53
gl_30_10.9	1	80	151	194	5	4.48	5	15.04
gl_30_10.10	1	55	151	137	5	4.56	5	4.31
gl_50_6.1	11	284	251	769	8	5.69	6	67.35
gl_50_6.2(+)	3	180	251	571	10	5.89	6	24.56
gl_50_6.3	27	489	251	1307	7	5.59	6	235.51
gl_50_6.4(+)	3	271	251	818	10	5.61	6	58.59
gl_50_6.5(+)	3	190	251	545	10	5.69	6	26.53
gl_50_6.6	3	236	251	691	7	5.49	6	179.49
gl_50_6.7(+)	3	203	251	618	10	5.69	6	34.46
gl_50_6.8(+)	3	215	251	597	10	5.60	6	89.20
gl_50_6.9(+)	3	204	251	623	10	5.50	6	52.97
gl_50_6.10(+)	3	194	251	563	10	5.54	6	53.67
gl_50_10.1	3	188	251	561	6	4.99	5	31.87
gl_50_10.2(∗)	1	313	251	919	6	5.00	5	50.69
gl_50_10.3	3	355	251	968	7	5.21	6	30.93
gl_50_10.4	3	309	251	846	6	4.99	5	51.88
gl_50_10.5	3	204	251	599	7	4.98	5	31.69
gl_50_10.6(∗)	1	271	251	690	7	5.00	5	36.26
gl_50_10.7(+)	3	280	251	801	8	5.09	6	31.04
gl_50_10.8	3	180	251	545	6	4.99	5	46.66
gl_50_10.9	3	321	251	878	7	5.06	6	63.54
gl_50_10.10(+)	3	214	251	642	8	5.25	6	20.73

Table 8: Results from **B&P** algorithm for C2 instances.

Instance	#nod	#sub	#ini	#gen	z_{ub}	z_r	z^*	time
gh_25_5.1	1	50	126	127	4	3.42	4	7.63
gh_25_5.2	1	61	126	206	4	3.47	4	8.14
gh_25_5.3	1	55	126	158	4	3.39	4	7.61
gh_25_5.4	1	54	126	177	4	3.43	4	6.89
gh_25_5.5	1	42	126	148	4	3.44	4	6.56
gh_25_5.6	1	38	126	99	4	3.36	4	4.78
gh_25_5.7	1	63	126	175	4	3.38	4	8.61
gh_25_5.8	1	52	126	156	4	3.40	4	7.32
gh_25_5.9	1	46	126	143	4	3.47	4	4.55
gh_25_5.10	1	50	126	160	4	3.39	4	9.13
gh_30_8.1	1	86	151	233	4	3.61	4	8.02
gh_30_8.2	3	76	151	240	5	3.54	4	7.80
gh_30_8.3	1	81	151	264	4	3.58	4	7.55
gh_30_8.4	1	68	151	240	4	3.55	4	10.61
gh_30_8.5	1	82	151	254	4	3.57	4	9.38
gh_30_8.6	1	61	151	199	4	3.52	4	8.77
gh_30_8.7	1	52	151	188	4	3.54	4	8.10
gh_30_8.8	1	72	151	249	4	3.48	4	9.66
gh_30_8.9	1	81	151	254	4	3.50	4	12.05
gh_30_8.10	1	64	151	216	4	3.43	4	11.99
gh_50_2.1(+)	3	306	251	879	10	5.43	6	75.92
gh_50_2.2	1	245	251	748	6	5.53	6	64.61
gh_50_2.3(+)	5	265	251	812	10	5.49	6	97.96
gh_50_2.4(+)	3	355	251	1008	10	5.38	6	96.88
gh_50_2.5(+)	3	356	251	1035	10	5.39	6	78.69
gh_50_2.6	3	299	251	838	8	5.67	6	56.67
gh_50_2.7(+)	3	312	251	875	10	5.55	6	65.84
gh_50_2.8	3	357	251	1025	7	5.40	6	78.06
gh_50_2.9	1	255	251	771	6	5.37	6	104.11
gh_50_2.10(+)	3	379	251	1061	10	5.43	6	147.40
gh_50_3.1	5	139	251	511	6	4.85	5	87.16
gh_50_3.2	3	207	251	774	8	4.88	5	63.66
gh_50_3.3	1	160	251	559	5	4.83	5	73.31
gh_50_3.4	3	122	251	473	6	4.92	5	41.67
gh_50_3.5	3	132	251	489	6	4.91	5	37.88
gh_50_3.6(+)	3	145	251	588	8	4.91	5	43.52
gh_50_3.7(+)	3	210	251	810	8	4.96	5	61.17
gh_50_3.8	3	104	251	399	6	4.82	5	58.31
gh_50_3.9	3	204	251	770	6	4.78	5	69.74
gh_50_3.10(+)	11	211	251	766	8	5.48	6	95.96
gh_50_8.1(+)	3	330	251	970	10	5.91	6	61.48
gh_50_8.2(+)	3	448	251	1275	10	5.71	6	71.14
gh_50_8.3	1	309	251	890	6	5.46	6	56.61
gh_50_8.4	1	277	251	890	6	5.42	6	56.57
gh_50_8.5(+)	3	302	251	951	10	5.96	6	58.38
gh_50_8.6(+)	3	340	251	977	10	5.63	6	61.88
gh_50_8.7(+)	3	314	251	1033	10	5.93	6	62.87
gh_50_8.8(+)	3	372	251	1146	10	5.90	6	63.65
gh_50_8.9(+)	19	486	251	1428	10	5.98	6	171.15
gh_50_8.10(+)	3	267	251	820	10	5.96	6	58.27
gh_50_9.1	7	221	251	753	7	4.70	5	100.27
gh_50_9.2	3	204	251	678	6	4.63	5	53.37
gh_50_9.3	3	172	251	566	6	4.69	5	70.03
gh_50_9.4(+)	3	196	251	702	8	4.95	5	46.93
gh_50_9.5(+)	3	166	251	567	8	4.72	5	58.23
gh_50_9.6(+)	3	232	251	894	8	4.75	5	69.20
gh_50_9.7(+)	3	190	251	679	8	4.95	5	49.39
gh_50_9.8(+)	1	215	251	685	6	4.66	5	67.53
gh_50_9.9(+)	1	160	251	521	5	4.69	5	136.99
gh_50_9.10	3	152	251	517	6	4.74	5	47.14
gh_50_10.1(+)	3	182	251	638	8	4.78	5	57.09
gh_50_10.2(+)	3	208	251	751	8	4.90	5	47.38
gh_50_10.3(+)	3	181	251	618	8	4.80	5	62.04
gh_50_10.4(+)	3	204	251	726	8	4.77	5	69.53
gh_50_10.5(+)	3	204	251	747	8	4.88	5	35.62
gh_50_10.6(+)	3	171	251	593	8	4.79	5	48.34
gh_50_10.7(+)	3	198	251	711	8	4.93	5	53.19
gh_50_10.8(+)	3	210	251	688	8	4.85	5	72.16
gh_50_10.9(+)	3	277	251	941	8	4.93	5	64.30
gh_50_10.10(+)	3	199	251	737	8	4.83	5	63.29

Table 9: Results from **B&P** algorithm for `C2` instances (cont.).

Instance	#nod	#sub	#ini	#gen	z_{ub}	z_r	z^*	time
rl_15_2.1(∗)	1	19	76	38	3	3.00	3	0.69
rl_15_2.2	1	19	76	50	4	3.13	4	0.65
rl_15_2.3(∗)	1	21	76	44	3	3.00	3	1.09
rl_15_2.4(∗)	1	23	76	39	3	3.00	3	1.05
rl_15_2.5(∗)	1	19	76	30	3	3.00	3	0.79
rl_15_2.6(∗)	1	14	76	34	3	3.00	3	0.59
rl_15_2.7	5	19	76	39	4	2.99	3	0.77
rl_15_2.8	1	19	76	38	3	2.96	3	0.81
rl_15_2.9	1	15	76	33	4	3.06	4	0.43
rl_15_2.10(∗)	1	17	76	35	3	3.00	3	0.70
rl_15_5.1	1	13	76	32	3	2.96	3	0.63
rl_15_5.2(∗)	1	11	76	25	3	3.00	3	0.49
rl_15_5.3	1	15	76	37	3	2.95	3	0.64
rl_15_5.4(∗)	1	11	76	25	3	3.00	3	0.49
rl_15_5.5(∗)	1	19	76	43	3	3.00	3	0.72
rl_15_5.6(∗)	1	11	76	25	3	3.00	3	0.48
rl_15_5.7	1	13	76	32	3	2.95	3	0.63
rl_15_5.8(∗)	1	11	76	25	3	3.00	3	0.48
rl_15_5.9	1	13	76	32	3	2.96	3	0.62
rl_15_5.10(∗)	1	11	76	25	3	3.00	3	0.51
rl_25_3.1	1	53	126	146	4	3.49	4	5.33
rl_25_3.2	1	53	126	126	4	3.53	4	5.09
rl_25_3.3	1	47	126	129	4	3.50	4	3.15
rl_25_3.4	1	49	126	132	4	3.56	4	4.64
rl_25_3.5	1	49	126	134	4	3.50	4	5.01
rl_25_3.6	1	45	126	137	4	3.61	4	4.86
rl_25_3.7	1	45	126	119	4	3.50	4	3.72
rl_25_3.8	1	45	126	122	4	3.50	4	3.75
rl_25_3.9	3	44	126	120	5	3.58	4	4.59
rl_25_3.10	1	45	126	117	4	3.50	4	4.19
rl_30_2.1	3	73	151	180	5	3.86	4	8.62
rl_30_2.2	1	66	151	179	4	3.89	4	7.28
rl_30_2.3(+)	3	61	151	182	6	3.92	4	7.04
rl_30_2.4	1	60	151	174	4	3.95	4	5.22
rl_30_2.5	1	58	151	168	4	3.92	4	5.25
rl_30_2.6	3	68	151	183	5	3.81	4	12.11
rl_30_2.7	1	65	151	183	4	3.99	4	8.59
rl_30_2.8	1	61	151	177	4	3.96	4	5.59
rl_30_2.9	1	60	151	204	4	3.92	4	6.37
rl_30_2.10	3	86	151	216	5	3.91	4	14.83
rl_30_9.1	1	51	151	166	4	3.82	4	4.33
rl_30_9.2	1	66	151	189	4	3.83	4	4.94
rl_30_9.3	1	53	151	166	4	3.86	4	4.44
rl_30_9.4(+)	3	62	151	179	6	3.90	4	3.59
rl_30_9.5	1	56	151	146	4	3.79	4	5.46
rl_30_9.6	1	60	151	165	4	3.85	4	4.28
rl_30_9.7	1	65	151	185	4	3.90	4	3.36
rl_30_9.8	1	62	151	180	4	3.85	4	4.28
rl_30_9.9	1	53	151	147	4	3.81	4	4.06
rl_30_9.10(+)	3	63	151	177	6	3.85	4	4.83
rl_50_2.1(+)	3	196	251	586	8	4.79	5	33.03
rl_50_2.2(+)	3	173	251	513	8	4.71	5	199.39
rl_50_2.3(+)	3	164	251	520	8	4.93	5	20.71
rl_50_2.4(+)	3	201	251	616	8	4.89	5	25.86
rl_50_2.5(+)	3	172	251	555	8	4.83	5	19.85
rl_50_2.6(+)	3	162	251	513	8	4.72	5	26.31
rl_50_2.7(+)	3	177	251	533	8	4.81	5	19.74
rl_50_2.8(+)	3	190	251	572	8	4.87	5	25.91
rl_50_2.9(+)	5	193	251	627	8	4.87	5	32.95
rl_50_2.10(+)	3	193	251	632	8	4.82	5	27.12
rl_50_8.1(+)	3	174	251	552	8	4.98	5	25.85
rl_50_8.2(+)	3	151	251	481	8	4.91	5	20.86
rl_50_8.3(+)	3	173	251	632	8	4.97	5	29.94
rl_50_8.4(+)	5	191	251	842	8	4.99	5	36.46
rl_50_8.5(+)	3	160	251	481	8	5.06	6	18.46
rl_50_8.6(+)	3	206	251	682	8	4.89	5	29.26
rl_50_8.7(+)	3	178	251	514	8	4.97	5	23.39
rl_50_8.8(+)	3	172	251	583	8	4.96	5	24.38
rl_50_8.9(+)	3	169	251	563	8	4.99	5	33.33
rl_50_8.10(+)	3	189	251	584	8	4.95	5	39.94
rh_15_10.1	1	26	76	44	3	2.87	3	2.69
rh_15_10.2	1	19	76	43	3	2.88	3	2.03
rh_15_10.3	1	25	76	50	3	2.99	3	3.40
rh_15_10.4(∗)	1	15	76	49	3	3.00	3	2.33
rh_15_10.5	1	25	76	53	3	2.99	3	3.58
rh_15_10.6(∗)	1	11	76	42	3	3.00	3	1.81
rh_15_10.7(∗)	1	11	76	37	3	3.00	3	1.78
rh_15_10.8(∗)	1	15	76	49	3	3.00	3	2.28
rh_15_10.9	1	16	76	54	3	2.97	3	1.72
rh_15_10.10(∗)	1	10	76	37	3	3.00	3	1.72
rh_30_5.1	1	62	151	206	4	3.32	4	20.67
rh_30_5.2	1	65	151	210	4	3.35	4	19.30
rh_30_5.3	1	61	151	195	4	3.29	4	14.49
rh_30_5.4	1	58	151	183	4	3.36	4	19.36
rh_30_5.5	1	61	151	214	4	3.39	4	22.96
rh_30_5.6	1	58	151	195	4	3.35	4	19.71
rh_30_5.7	1	58	151	195	4	3.36	4	14.88
rh_30_5.8	1	50	151	192	4	3.38	4	20.44
rh_30_5.9	1	58	151	196	4	3.38	4	20.75
rh_30_5.10	1	49	151	186	4	3.37	4	11.54

Table 10: Results from **B&P** algorithm for C2 instances (cont.).

Instance	#nod	#sub	#ini	#gen	z_{ub}	z_r	z^*	time
rl_15_7	1	17	76	114	16	31.9	-	1.12
rl_25_3	1	37	126	232	26	30.6	-	3.09
rl_30_2	1	51	151	308	31	37.3	-	4.87
rl_30_9	1	55	151	309	31	39.5	-	4.60
rl_50_2	1	139	251	696	51	89.7	-	19.65
rl_25_1	3	36	126	228	6	3.4	4	3.17
rl_30_3	3	60	151	320	6	3.7	4	6.08
rl_30_6	3	59	151	349	6	3.9	4	4.32

Table 11: Results of the (SP) model for C3 instances.

incumbent whenever a feasible solution is not available. The results for 8 instances in C3 are summarized in table 11 whose columns have the usual meanings. Notice that, according to the specifications in section 3, if the optimal value of the linear relaxation exceeds n then we can conclude that the instance is infeasible.

Remarkably, the **B&P** algorithm proved the infeasibility of 5 out of the 12 instances in few seconds. Those 5 instances, namely rl_15_7, rl_25_3, rl_30_2, rl_30_9 and rl_50_2 were not known to be infeasible. On the other hand, our algorithm was also able to find the optimum for 3 other instances: rl_25_1, rl_30_3 and rl_30_6. Again, the computation was completed in seconds. However, these 3 instances were already known to be feasible (see [1]). But it is worth noting that among these 3 cases the heuristic GRASP only found a feasible solution for instance rl_30_3.

Unfortunately, for the remaining 4 instances of class C3 (rl_15_2,rl_15_5, rl_50_8 and rl_50_9) the **B&P** algorithm ran into memory problems. The problem occurred in internal structures of the commercial LP solver which we have no access. The origin of the error seems to be related to the number of matrices that are simultaneous kept in memory. We did not pay to much attention in our implementation to memory usage since our experiments with feasible instances shown that very few nodes were explored with **B&P**. Maintaining matrix data on active nodes made the algorithm faster and did not caused memory problems. As we found out later, this is not the case for infeasible instances that require the exploration of larger portions of the state space. We believe that with a deep change in our data structures we could overcame this obstacle and possibly reach a conclusion concerning the feasibility status of these 4 instances. Though theses changes might affect the running time of **B&P**, it will probably remain very competitive.

5 Conclusions

In this paper we introduced new IP formulations for SRAP. Extensive computational experiments were carried out on benchmark instances taken from the literature. Among the models with polynomial number of variables, our model (CM) proved to be the most efficient. The branch-and-bound (**B&B**) algorithm based on our model solved more instances and faster than that based on models proposed earlier. Moreover, we have also introduced

a set partitioning formulation of SRAP with one extra knapsack constraint. To compute this model, which has an exponential number of variables, we have designed and implemented a branch-and-price (**B&P**) algorithm. Our **B&P** code proved to be largely superior to the **B&B** codes independently on which compact formulation was used in the latter. Not only **B&P** runs almost two orders of magnitude faster on average than the fastest of its competitors but it is also able to demonstrate rapidly the infeasibility of some instances which, otherwise, would require hours, if not days, to be proven by the other methods. Our results contrast with some statements that appeared earlier in the literature about the inadequacy of IP models and algorithms for solving SRAP.

A Appendix: Heuristic for generating upper bounds

In this appendix we explain how we generated the primal bounds for SRAP that are used as an incumbent value by all integer programming models. The procedure we applied belongs to a class of meta-heuristics known as GRASP (Greedy Randomized Adaptive Search Procedure) and proposed in [8]. In a basic GRASP algorithm, a constant number of iterations is executed, each of which is composed of two phases. The first one is a greedy randomized construction phase while the second one is a local search procedure. The randomization of the construction phase is implemented through a change on the standard greedy choice step. Rather than bringing the element with the best value according to a given criterion, a list of candidate elements is built with the β best elements and one of them is randomly chosen to be incorporated to the solution. Here both β and the total number of iterations performed by the algorithm are parameters to be fixed a priori. Below we summarize the GRASP that we implemented for SRAP. However, before we continue, we want to emphasize that our only purpose in implementing this meta-heuristic was to have a good incumbent solution to drive the exact methods. Therefore, we did not run into extensive computations to tune the code and, by no means, we claim that our heuristics outperforms other existing ones.

From now on, the term GRASP is used to denote our specific implementation designed to solve SRAP instances. We first notice that it is possible for GRASP to return infeasible solutions. Of course this must be allowed since, sometimes, we cannot decide in reasonable computing time if the instance has a solution or not. Even when the construction phase discovers that the instance is feasible, infeasible solutions can be visited. The goal here is to allow more flexibility to move along the search space. However, the objective function has been modified to turn feasible solutions more attractive than infeasible ones. To explain the changes in the objective function, we define the following values associated to a feasible solution S with vertex set $V_1, \ldots, V_\ell$:

$$F = \max\{0, \overline{D}\}, \tag{54}$$

$$R = \max\{0, \max\{j \in \{1, \ldots, \ell\} : d_j - B\}\}, \tag{55}$$

where $\overline{D}$ and d_j are defined as in (20) and (19), respectively. Now, the cost of this solution in the modified objective function is computed by the formula

$$c(S) = \ell + \alpha F + \xi R, \tag{56}$$

where α and ξ are penalization parameters for those solutions violating the capacity of any ring, including the federal ring. In our computations both α and ξ were set to one. This means that every unit of demand in excess in the local ring with the most violated capacity constraint and in the federal ring costs as much as the inclusion of an extra ring in the solution.

We now turn our attention to the construction phase of `GRASP`. The algorithm is an adaptation of the *edge-based heuristic* presented in [9] to include randomization. It always starts with a solution containing n rings with only one vertex assigned to it. Those rings are feasible since otherwise the instance itself is infeasible. Therefore, the only constraint that can be violated is the one that imposes a limit on the demand on the federal ring. At each iteration, the union of two rings into a single one is considered. Such an union is accepted only if the resulting ring is feasible. Clearly, this operation reduces the amount of demand on the federal ring. The greedy choice is guided by the demands on edges which are in the multicutset of the rings. The larger the demand on the edge, the better is the edge. As we seen earlier, the randomization will not force us to pick the best edge according to this criteria. Instead, the selected edge is chosen randomly among the β ones with smaller demand. The steps of the construction algorithm are summarized in figure 2. The variable $\pi[u]$ denotes the index of the ring containing vertex u of V.

```
procedure GRASP-construction-phase(S);
1.   for u ← 1 to n do
2.        R_u ← {u};    π[u] ← R_u;
3.   end for
4.   E' ← E;
5.   while E' ≠ ∅ do
6.        Build the set L of edges with the β largest
          demands in E';
7.        Randomly select an edge (u, v) in L;
8.        if π[u] ∪ π[v] is a feasible ring then
9.             E' ← E' \ {(w, w')|w ∈ π[u] and w' ∈ π[v]};
10.            for all vertices w ∈ π[v] do π[w] ← π[u];
11.       else E' ← E' \ {(u, v)};
12.  end while;
13.  return the solution S with the rings constructed above;
end GRASP-construction-phase.
```

Figure 2: Construction phase of `GRASP`.

The local search phase is based on two types of neighborhoods. The first neighborhood considers the possibility of moving every vertex u from any of the existing rings different from $\pi[u]$. The second neighborhood considers all possibilities of exchanging pairs of vertices among different rings. The neighborhoods are inspected in that order as it is suggested in the algorithm in figure 3. The values of c in this algorithm are those defined in (56).

Finally, figure 4 gives an overview of our `GRASP`. Notice that the number iterations we allowed is twice the number of vertices in the graph. Though this number is modest

```
procedure GRASP-local-search-phase(S);
1.  Initialize S as the solution returned by the construction phase;
2.  (* First neighborhood *)
3.  Let {R_1, R_2, ..., R_k} be the rings in S;
4.  for i ← 1 to k − 1 do
5.      for j ← i + 1 to k do
6.          for all u ∈ R_i do
7.              Let S' be the solution obtained from S by
                moving vertex u from R_i to R_j;
8.              if c(S) > c(S') then S ← S';
9.  (* Second neighborhood *)
10. for i ← 1 to k − 1 do
11.     for j ← i + 1 to k do
12.         for all u ∈ R_i and v ∈ R_j do
13.             Let S' be the solution obtained from S by
                exchanging vertices u and v among rings R_i and R_j;
14.             if c(S) > c(S') then S ← S';
end GRASP-local-search-phase.
```

Figure 3: Local search phase of GRASP.

compared to other implementations of this meta-heuristics for other combinatorial problems, this is compensated by the fact that we ran our GRASP for the nine distinct values of the parameter β in $\{2, \ldots, 10\}$.

```
procedure GRASP;
1.  Initialize c* ← +∞ and S*;
2.  for i ← 1 to 2 * n do
3.      GRASP-construction-phase(S);
4.      GRASP-local-search-phase(S);
5.      if c(S) < c* then
6.          Update: c* ← c(S) and S* ← S;
7.  return S*;
end GRASP.
```

Figure 4: The GRASP algorithm.

From table 12 we can assess the results from our GRASP. The statistics shown there refer to the best solutions obtained for β in $\{2, \ldots, 10\}$ for all instances in classes C1 and C2(see subsection 4.1), 341 in total. Columns "% opt" and "% fails" refer to the percentage of instances where at least one of the nine GRASP executions achieved the optimum and to the percentage of instances for which all nine runs were unable to find a feasible solution, respectively. For those instances where a fail did not occurred but no optimal solution was found, column "max" shows the maximum values of the distance between the best GRASP solution and the optimum value computed. Results are presented for each of the instance

classes. As it can be seen from table 12, our simple `GRASP` is still able to produce relatively

class	% opt	% fails	max
`C1`	91	0	34%
`C2`	59	27	29%
Total	70	18	

Table 12: Summary of results for `GRASP`.

good bounds. It is clear however that there is still room for improvements especially to reduce the occurrence of fails. Our experiments showed that fails tend to increase with instance sizes. Also, as expected, instances in class `C2`, concentrate most cases of fails. As mentioned earlier, those were especially conceived to be hard instances to solve heuristically.

References

[1] R. Aringhieri and M. Dell'Amico. Solution of the sonet ring assignment problem with capacity constraints. Technical Report DISMI TR-12, Dipartimento di Scienze e Metodi dell'Ingegneria, Università di Modena e Reggio Emilia, 2001.

[2] E. Balas and M. Padberg. Set partitioning: a survey. *SIAM Review*, 18:710–760, 1976.

[3] F. Barahona and A. R. Mahjoub. On the cut polytope. *Mathematical Programming*, 36:157–173, 1986.

[4] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Salvelsbergh, and P. H. Vance. Branch-and-price: column generation for solving huge integer programs. *Operations Research*, 46:316–329, 1998.

[5] Dash Optimization, Inc. *XPRESS-MP: user manuals*, 1999. Versão 12.05.

[6] C. de Simone. The cut polytope and the boolean quadric polytope. *Discrete Mathematics*, 79:71–75, 1989/1990.

[7] C. de Simone. Lifting facets on the cut polytope. *Operations Research Letters*, 9:341–344, 1990.

[8] T. A. Feo and M. G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[9] O. Goldschmidt, A. Laugier, and E. V. Olinick. SONET/SDH ring assignment with capacity constraints. *Discrete Applied Mathematics*, 129:99–128, 2003.

[10] M. Grötschel and M. Padberg. Polyhedral theory. In Eugene L. Lawler, Jan K. Lenstra, Rinnoy Kan, and D. B. Shmoys, editors, *The Traveling Salesman Problem: A guided tour of combinatorial optimization*, pages 251–305. John Wiley & Sons, 1985.

[11] S. Holm and M. M. Sorensen. The optimal graph partitioning problem: solution method based on reducing symmetric nature and combinatorial cuts. *O.R. Spektrum*, 15:1–8, 1993.

[12] E. M. Macambira. *Modelos e Algoritmos de Programação Inteira no Projeto de Redes de Telecomunicações*. PhD thesis, COPPE, Universidade Federal do Rio de Janeiro, Brazil, 2003. in portuguese.

[13] S. Martello, D. Pisinger, and P. Toth. New trends in exact algorithms for the 0-1 knapsack problem. *European Journal of Operational Research*, 123:325–332, 2000.

[14] A. Mehrotra. Cardinality constrained boolean quadratic polytope. *Discrete Applied Mathematics*, 79:137–154, 1997.

[15] G. L. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization*. Wiley & Sons, 1988.

[16] M. Padberg. The Boolean quadric polytope: some characteristics, facets and relatives. *Mathematical Programming*, B-45:139–172, 1989.

[17] D. M. Ryan and B. A. Foster. An integer programming approach to scheduling. In A. Wren (editor), editor, *Computer scheduling of public transport urban passenger vehicle and crew scheduling*, pages 269–280. Noth-Holland, 1981.

[18] H. D. Sherali and J. C. Smith. Improving discrete model representations via symmetry considerations. *Management Science*, 47(10):1396–1407, 2001.

[19] F. Vanderbeck and L. A. Wolsey. An exact algorithm for IP column generation. *Operations Research Letters*, 19(4):151–159, 1996.