Université catholique de Louvain
Faculté des sciences appliquées
Département d'ingénierie informatique

Global Constraint for the Set Covering Problem

Promoteur: Yves Deville
Lecteurs: Laurence Wolsey
Pierre Schaus

Mémoire présenté en vue de l'obtention du grade d'ingénieur civil en informatique par
Sébastien Mouthuy

Louvain-La-Neuve
2006-2007

Acknowledgments

Je voudrais sincèrement remercier mon promoteur le Pr.Yves Deville pour son soutien tout au long de ce travail. Il m'a mis en contact avec des personnes très intéressantes, dans le cadre de ce mémoire ainsi que dans la perspective de collaborations futures. Enfin, je le remercie de la confiance qu'il m'a portée en acceptant que nous publiions un papier aux Journées Francophones de la Programmation par Contraintes (JFPC). Sa persévérance et sa rigueur m'a été d'un très grand apport pour structurer mes idées et qu'elles ne ressemblent pas à ces songes qui nous apparaissent juste avant l'heure du réveil.

Je remercie le Dr.Jean-Charles Régin pour nous avoir suggéré la pertinence d'une contrainte globale pour le problème du recouvrement par ensemble. Enfin, il a stimulé ma curiosité même à la veille de la remise de ce travail en ouvrant des perspectives pertinentes laissant entrevoir l'intérêt de pousser la recherche initiée dans ce mémoire plus avant. Je le remercie très chaleureusement pour cette vision large du sujet car j'y vois une nouvelle vague qui me permettra peut-être de poursuivre mes recherches encore plus loin.

Ma gratitude va aussi au Pr.Laurence Wolsey pour m'avoir accueilli à plusieurs reprises afin de discuter et de réfléchir au problème. Les pistes et réponses qui en sont ressorties ont impacté de manière significative le contenu de travail.

Je me rappelle aussi de discussions très intéressantes que j'ai eues avec Damien Saucez et Pierre Gramme, et qui ont ouvert à un dynamisme intellectuel certain. Je n'aurai jamais de regret par rapport aux projets que j'ai fait en collaboration avec Damien. Nous partageons tous les deux le souci d'allier la proposition d'idées originales, qui ressortent du cadre de l'application simple, et de la concrétisation fiable de ces dernières. Y arriver est une chose d'extrêmement gratifiante. Les échanges que nous avons eu m'ont toujours beaucoup apportés.

Contents

Chapter 1

Introduction

This work has been initiated in September 2006 during the last year of my master in the *Département d'ingénierie informatique* at the University of Louvain-La-Neuve. The conclusions of the first part of this work have been submitted and presented at the conference *Journées Françaises de la Programmation par Contraintes* (JFPC - Paris) in June 2007.

This work has been done in the field of Constraint Programming, a programming paradigm that allows users to specify and solve efficiently a large number of real-world combinatorial problems. The objective of this work was to design a global constraint and implement it in an efficient CP engine.

The Set Covering Problem (SCP) has been chosen as an interesting problem to solve, for which it does not exist any global constraint in the CP community yet. This problem is very attractive for CP in three main points. Firstly, the Set Covering Problem is a generalization of many important problems in combinatorial optimization: vehicle routing , crew assignment and bin packing problems. Optimal electronic circuit' s tests design can also be formulated as a Set Covering Problem. Secondly, many useful problems arising in graph theory can be formulated as SCP's, such as the independent set, the vertex cover and the dominating set problems. Thirdly, in Constraint Programming, the constraint NValue that counts the number of values used by the assignment of a vector of variables is a specialization of the Set Covering Problem.

Thus a global constraint for SCP can be very useful from a user point of view to model and to solve efficiently many real-world problems.

This work has the five following technical contents. Firstly we analyse the structure of the SCP to design a generic SC constraint that needs a

lower bound on the cost of a set cover to do pruning. Secondly, we analyse such lower bounds that we found in the literature and that can be used into our generic constraint. Thirdly we introduce a lower bound that can be computed by relaxing the initial SCP into an edge covering problem. Fourthly we present an efficient algorithm to compute incrementally this edge covering problem based on the Hungarian method. Finally we extend our constraint to handle problems for which the collection of subsets is too big to be enumerated explicitly. Such problems cannot be tackled with the first approach.

The set covering problem has been extensively addressed over more than forty years. The objective of this work is not to give a final solution to this problem, but to present an initial insight in some possible ways of solving set covering problems with CP. This work has been lead in a research way of thinking; aiming at being creative, able to efficiently materialize our ideas and sharing them to other researchers.

There are two main conclusions of this work. First the set covering problem is a very difficult problem for CP, whose aim is to use the intrinsic structure of problems to reduce the search path of solutions, because SCP has not much structure. This should be kept in mind when considering set covering problems in constraint programming. Second, CP users should remember from this work that using a SC constraint to tackle specializations of SCP is not efficient if the size of the problems is too large. Because the SC constraint is only aware of the covering structure of a problem, developing more specialized propagators that take the additional structure into account would be worth the additional implementation cost.

This report is divided into four main parts. The first part gives a background on contraint programming and optimization theory. The objective of the two first chapters is not to give an exhaustive view of these broad fields, but it aims at giving an intuitive but formal view to the reader who is not used to one of these domains. It also allows to emphasize which parts of this work is more innovative. The second part analyses the set covering problem in more depth and identifies the generic SC constraint. The third part of this report describes our main contributions: the 2SC relaxation, the algorithm to solve it and the extension of the SC constraint to handle very large SCP. The last part contains the experimental sections with an explanation of the non-trivial points of our implementation and quantitative results to assess the added value of the work presented in this report.

Part I

Background

Chapter 2

Constraint Programming

2.1 Main Ideas

is a programming paradigm in which programs contains variables whose relations are expressed by (possibly high-order) constraints on their domains. A solution to such a program would be an assignment of all variables such that all constraints of the program hold. Constraint Programming differs from most well-known programming paradigms by the fact that it specifies properties about variables instead of specifying how compute solutions.

With such a programming paradigm at hand, the user can extend his programs easily as it suffices to add or remove constraints between variables to modify it.

The above concepts should be understood in their most generic meaning. Variables could represents integer, sets, graphs, topologies,... Constraints could also be any kind of imaginable properties between variables.

Historically, constraints were first introduced into [JL87] to create the paradigm (CLP). In Logic Programming, variables can be in two states: bound (assigned to a value) or not bound. CLP allows variables to be constrained rather than bound, constraints are active and could remove several values from the domain of a variable as soon as it detects that these values cannot be part of any solution of the problem. This allows to prune the search space and to decrease the time needed to find one or all solutions. A propagator is a piece of code that makes a constraint *active* by implementing a filtering algorithm. Propagators are executed when domains of the variables have changed, and they try to prune the new domain.

The following sections formalize all these concepts.

2.2 Constraint Satisfaction Problem

Problems that aim at finding solutions respecting a set of constraints are called (CSP).

A CSP is defined over a set of variables $\mathcal{X} = \{X_1, \ldots, X_k\}$. Let $\mathcal{U}$ be the set of all possible values. Each variable $X_i \in \mathcal{X}$ has a set D_i of possible values that is called domain of X_i. We define a as the cross product of all these domains: $\mathcal{D} = \prod_{i=1}^{k} D_i, (D_i \in 2^{\mathcal{U}})$. In the following we shall denote the domain of variable X_i by $\mathcal{D}(X_i) = \pi_i(\mathcal{D}) = D_i$.

An is a function $\Lambda : \mathcal{X} \rightarrow \mathcal{U}$ assigning a single value to each variable of the CSP. A constraint $\mathfrak{c}$ is a set of valid assignments: $\mathfrak{c} \subseteq \mathcal{D}$. Thus Λ is a valid assignment respecting constraint $\mathfrak{c}$ iff

$$(\Lambda(X_1), \ldots, \Lambda(X_k)) \in \mathfrak{c} \wedge \Lambda(X_i) \in D_i, forall i : 1 \leq i \leq k$$

In the following we shall write $\Lambda \in \mathfrak{c}$ to state that.

Formally a CSP is a pair $(\mathcal{C}, \mathcal{D})$, where $\mathcal{C}$ is a set of constraints and $\mathcal{D}$ is a store. A solution to this CSP is an assignment Λ respecting all constraints; $\forall \mathfrak{c} \in \mathcal{C}, \Lambda \in \mathfrak{c}$. The set of solutions of a CSP is denoted by $\mathcal{S}\wr\updownarrow(\mathcal{C}, \Delta)$. Two CSP's $(\mathcal{C}, \mathcal{D})$ and $(\mathcal{C}', \mathcal{D}')$ are equivalent if $\mathcal{S}\wr\updownarrow(\mathcal{C}, \mathcal{D}) = \mathcal{S}\wr\updownarrow(\mathcal{C}', \mathcal{D}')$. A CSP $(\mathcal{C}, \mathcal{D})$ is *failed* if $\mathcal{S}\wr\updownarrow(\mathcal{C}, \mathcal{D}) = \emptyset$.

A *linear program* is an example of CSP. In this case, domains are subsets of $\mathbb{R}$, the set of reals. Constraints are linear inequalities that variables should respect.

2.3 Constraint Programming

Constraint Programming aims at offering a nice framework to the user that allows him to solve efficiently CSP's while requiring small development time.

A user should be able to formulate his CSP in a declarative way; by listing all variables, their domains and by specifying all constraints. This is the first of the two levels of the CP programs architecture.

The second is the search component, where the user specify how the CP solver should explore the solutions. We can summarize the architecture of

CP programs by the equation

$$CP\ programs = constraints\ model + search$$

In order to solve a CSP, CP engines alternate propagation steps with branching steps: when all propagators are idle, a variable is fixed and the propagation can continue. When it is idle, it branches, etc. These steps are detailed below.

2.3.1 Propagation

In order to solve CSP's efficiently, Constraint Programming try to reduce the size of the domains by mean of propagation. Propagation aims at finding an equivalent CSP to the current one with smaller domains. It uses constraints from the CSP to detect invalid assignments and remove values from domains accordingly.

In order to reduce to size of the domain, constraints are coupled with propagators that implement filtering algorithms. This allow to consider constraints locally and remove a few values from domains as soon as the filtering algorithm detects they do not belong to any solution.

Example 2.1 (Propagation) *Imagine we should solve the following CSP*

Variables X, Y *with* $\mathcal{D}(X) = \{1, 2, 3, 4, 5\}$ *and* $\mathcal{D}(Y) = \{2, 3, 4, 5, 6\}$

Constraints $X \geq Y + 2$

A propagator implementing constraint $X \geq Y + 2$ *could remove values* $1, 2$ *and* 3 *from the domain of* X*, because* $Y \geq 2 \Rightarrow X \geq Y + 2 \geq 4$*. By the same reasoning, the propagator could remove values* $4, 5$ *and* 6 *from the domain of* Y*.*

An equivalent CSP to the former one is thus

Variables X, Y *with* $\mathcal{D}(X) = \{4, 5\}$ *and* $\mathcal{D}(Y) = \{2, 3\}$

Constraints $X \geq Y + 2$

It should be noted that when there are more than one constraint, each propagator looks for changes in variables domains. When there is a change it applies the filtering algorithms to possibly remove some values from some domains. Thus propagators *communicate* by mean of changes in variable domains.

Filtering Algorithms

Filtering algorithms implemented by propagators can be regarded as function transforming a CSP into an equivalent CSP with smaller domains. A filtering function $f_c : 2^{\mathcal{U}} \rightarrow 2^{\mathcal{U}}$ for the constraint $\mathfrak{c}$ must be

Contractant $f_c(\mathcal{D}_1) \subseteq \mathcal{D}_1$

Monotone $\mathcal{D}_1 \subseteq \mathcal{D}_2 \Rightarrow f_c(\mathcal{D}_1) \subseteq f_c(D_2)$

Correct $\mathcal{D}_1 \cap \mathfrak{c} \subseteq \mathcal{D}_1$

A filtering function f_c is *entailed* if $\forall \mathcal{D}' \subseteq \mathcal{D}, f_c(\mathcal{D}') = f_c(\mathcal{D})$. Filtering algorithms usually low time complexity as they often must be executed.

2.3.2 Search

Because the type of problems tackled by CP are generally hard problem, being NP-Hard in most cases, propagation in itself is not sufficient to solve them.

The framework of CP integrates then the possibility to explore the set of all assignments by mean of search heuristics. When all propagators have reached a fixed point (any of them cannot remove any value from any domain), we should fix a variable in order to go on.

Search is specified by a search tree and a traversal algorithm. The root node of the search tree represent the original CSP. Then descendents of a given node are obtained by splitting the CSP, i.e. by reducing the size of the domains. Often only one variable domain is splitted.

The traversal strategy usually used is DFS, because the space complexity of this algorithm is linear in the number of variables. Thus CP engines also provides support for backtracking. When propagation leads to a failed CSP (by removing values in domain), we should get to an ancestor of the current node and continue to explore the search tree.

A generic search algortihm is sketched in Algorithm 2.1

In the case DFS is used, $Trail$ can be implemented by a queue (First In First Out) and *choose* would select the last added entry in $Trail$. $propagate(\mathcal{C}, \mathcal{D})$ is provided by the CP engines; it calls all propagators until no one of them makes any more pruning to the doamins of the variables. For the search part,

SearchAll($(\mathcal{C}, \mathcal{D})$)

PRE: - $(\mathcal{C}, \mathcal{D})$ is a CSP
POST: - $\mathcal{S}olutions = \mathcal{S}\wr\updownarrow(\mathcal{C}, \mathcal{D})$

```
1: Trail = {(C, D)}
2: Solutions = ∅
3: while Trail ≠ ∅ do
4:    (C', D') = choose(Trail)
5:    propagate(C', D')
6:    if ∄X_i ∈ X : |D'(X_i)| = 0 then
7:       if ∀X_i ∈ X, |D'(X_i)| = 1 then
8:          Solutions := Solutions ∪ {D'}
9:       else
10:         Trail := Trail ∪ branch(C', D')
11:      end if
12:   end if
13: end while
```

Algorithm 2.1: Generic Search Algorithm

the user has only to provide the branch function in most cases. The *branch* function should return a set of CSP's whose domains of the store are subsets of the domains in the sotre of the current CSP.

2.4 Existing CP Engines and Libraries

The first variant of Constraint Logic Programming was developped in Prolog II [JL87]. They added support for disequations. The first real implementations of CLP were PrologIII, CLP(R) [JMSY92] and CHIP [P.91].

Nowadays a dozen fo different implementations of CP engines and libraries exist. Among others we can cite Ilog Solver that is a proprietary C++ library that provide an API to add support of CP in imperative programs. Gecode [gec] also provides such library while being open-source. The implementation Mozart of the language Oz [Smo98] provides a multiparadigm framework in which Constraint Programming is supported.

Chapter 3

Optimisation Theory

This section gives some background about optimization theory. This theory contains interesting theoretical results about optimization problems and their solutions that will be used through this work. It is a strong background for operations research. In this work we only focus on linear optimization problems.

The objective of this chapter is not to give an exhaustive overview of this field. It only aims at presenting results that are useful in this work.

3.1 Formulation of linear optimisation problems

This section presents how one formulate optimization problem in OR. Decision variables are denoted by $x_1, \ldots, x_n$. The objective is to minimize a weighted sum of all these decision variables: $\min \sum_{j=1}^{n} c_j x_j$. All the constraints are linear inequalities wrt the decision variables, i.e. of the form $\sum_{j=1}^{n} a_{ij} x_j \geq b_i$. More conveniently, problems are often formulated as the following

$$\begin{aligned} \min c^t x \quad & \text{subject to} && (3.1)\\ Ax &\geq b && (3.2)\\ x &\geq 0 && (3.3) \end{aligned}$$

where $c, x \in \mathbb{Q}^n$, $b \in \mathbb{Q}^m$ and $A \in Q^{m \times n}$. n is the number of decision variables $(x_1, \ldots, x_n)$. m is the number of constraints.

The set of solutions of an optimization problem is $\mathcal{P} = \{x \in \mathbb{Q}^n \mid Ax \geq b, x \geq 0\}$. The cost of a solution x is given by $cx = \sum_{j=1}^{n} c_j x_j$. The optimal solution will usually be denoted x^*.

All the following results use this specific formulation.

3.2 Polyhedral results

Definition 1 *A is the region defined on one side of a hyperplan.*

Definition 2 *A polyhedral is the intersection of some* half-spaces.

Because each constraint of a linear optimization problem defines a *half-space*, then the set of all solutions $\mathcal{P} = \{x \in \mathbb{Q}^n \mid Ax \geq b, x \geq 0\}$ is a polyhedral.

Definition 3 *An extreme point of a polyhedral $\mathcal{P}$ is a point $x \in \mathcal{P}$ such that there do not exist two points $y, z \in \mathcal{P}$ different from x and a scalar $\lambda : 0 \leq \lambda \leq 1$ such that $x = \lambda y + (1 - \lambda)z$.*

All these geometrical concepts are illustrated in Fig.3.1.

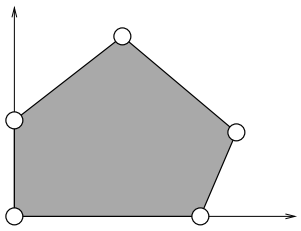

Figure 3.1: A polyhedral being defined by the intersection of five *half-space*. Extreme points are represented by circles.

Theorem 1 (Fundamental theorem) *If an optimization problem P has a finite optimal cost, and if the polyhedral $\mathcal{P}$ of all solutions of P has at least one extreme point, then there exists an extreme point of $\mathcal{P}$ that is an optimal solution of P.* ■

This tells us that it suffices to consider all the extreme points of the polyhedral $\mathcal{P}$ to find an optimal solution, if any exists.

3.3 The Simplex Algorithm

The simplex algorithm aims at finding an optimal solution to an optimization problem. It walks through adjacent extreme points until there is no better adjacent extreme points.

Although it has an exponential theoretical complexity, the simplex algorithm is highly efficient in practice. Moreover this algorithm is well suited to optimize problems incrementally, i.e. reoptimizing them after a few changes in the problem formulation.

3.4 Duality theory

3.4.1 The dual problem

Record that all optimization problems can be expressed under the form

$$\begin{array}{rcll} \min\ c^t x & & \text{subject to} & (P) \\ Ax & = & b & \\ x & \geq & 0 & \end{array}$$

This is usually called the primal problem. To each optimization problem we associate another problem, called the dual problem

$$\begin{array}{rcll} \max\ b^t y & & \text{subject to} & (D) \\ A^t y & \leq & c & \\ y & \geq & 0 & \end{array}$$

These two problems share a lot of interesting properties.

3.4.2 Main results

Proposition 2 (Weak duality) *If x is a solution of (P) and y a solution of (D), then*

$$c^t x \geq b^t y$$

■

This tells us that if we find a solution of the dual, then we have a lower bound on any solution of the primal (and on the optimal value of (P) in particular).

Proposition 3 (Strong duality) *If one of (P) or (D) has an finite optimal solution, then also does the other problem.*
If x^* (resp. y^*) is an optimal solution of (P) (resp. (D)), then

$$c^t x^* = b^t y^*$$

■

3.4.3 The Karüsch-Kuhn-Tucker conditions

The Karüsch-Kuhn-Tucker conditions are powerful results about optimal solutions of (P) and (D). They were initially introduced in [Kar39, KT50]. Note that we only present the KKT conditions for the linear case. These conditions also exist for more complex optimization problems (quadratic,...).

If we must solve the following problem

$$\min \sum_{j=1}^{n} c_j x_j \qquad \text{subject to} \tag{3.4}$$

$$\sum_{j=1}^{n} g_{ij} x_j \geq b_i \qquad \forall i \in I \tag{3.5}$$

$$\sum_{j=1}^{n} g_{ij} x_j = b_i \qquad \forall i \in E \tag{3.6}$$

The KKT conditions state that a solution x^* to the previous problem is optimal if and only if there exists λ^* such that

$$c_j - \sum_{i=1}^{m} g_{ij} \lambda_i^* = 0 \qquad \forall j : 0 \leq j \leq n \tag{3.7}$$

$$\sum_{j=1}^{n} g_{ij} x_j \geq b_i \qquad \forall i \in I \tag{3.8}$$

$$\sum_{j=1}^{n} g_{ij} x_j = b_i \qquad \forall i \in E \tag{3.9}$$

$$\lambda_i^* \geq 0 \qquad \forall i \in I \tag{3.10}$$

$$\lambda_i^* (\sum_{j=1}^{n} g_{ij} x_j) = 0 \qquad \forall i \in I \tag{3.11}$$

It can be proven that λ^* is an optimal solution of the dual (3.4)-(3.6).

3.5 Integer programming

Integer programming deals about solving optimization problems for which decision variables must take integer value. The formulation of IP is the following

$$\min c^t x \quad \text{subject to} \tag{3.12}$$

$$Ax \geq b \tag{3.13}$$

$$x \geq 0 \tag{3.14}$$

$$x_i \; is \; integral \qquad \forall i \in I \tag{3.15}$$

3.5.1 Challenges

The introduction of integer variables into problems leads to an increasing complexity. Although optimization problems without integrality constraints can be solved in polynomial time by interior points methods, most of the interesting problems with integrality constraints are NP-Hard to solve to optimally.

Integer programming aims at using tools from the continuous optimization theory to efficiently solve integer problems.

The main difficulty is that integrality constraints modify the polyhedral of feasible solutions in such a way that it is very difficult to compute it. The following definition is useful.

Definition 4 (Convex Hull) *The Convex Hull of a set of points $\mathcal{X}$ in $\mathbb{R}^n$ is the smallest convex set $\mathcal{C} \subseteq \mathbb{R}^n$ containing $\mathcal{X}$, i.e. the smallest convex set $\mathcal{C}$ such that $\mathcal{X} \subseteq \mathcal{C}$.*

It is straightforward that all extreme points of the Convex Hull will be integral points. Thus from Theorem 1, if we can compute the Convex Hull

of the set of solutions of an IP, then we can use a LP solver to find an optimal integral solution to the IP. However computing this Convex Hull can be NP-Hard. Fig.3.2 illustrates the convex hull of a subset of $\mathbb{R}^n$.

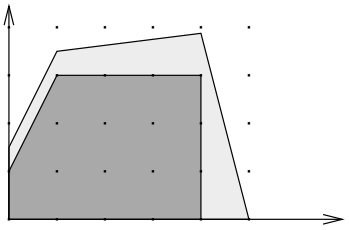

Figure 3.2: IP polyhedral. The bright shaded polyhedral is the polyhedral of the linear relaxation of the IP. The dark shaded polyhedral is the convex hull, i.e. the smallest polyhedral containing all integral solutions of the IP and whose extreme points are integral

3.5.2 Problems for which $LP \equiv IP$

For some problems, the integrality constraints does not lead to any increasing complexity. For these problems, it can be shown that relaxing the integrality constraint (3.15) does not change the polyhedral of solutions, i.e. the polyhedral of the solution of the linear relaxation is the same as the Convex Hull. Solving the linear relaxation with LP solvers provide an integral solution.

There exists a necessary and sufficient condition for an IP to be solvable via the linear relaxation.

Definition 5 (Totally Unimodular Matrix) *A matrix M is said to be totally unimodular (TU) if the determinant of all submatrices of M is equal to $0, 1$ or -1.*

Theorem 4 *The convex hull of the IP (3.12)-(3.15) is exactly the polyhedral $\mathcal{P}_{LP}$ of all solutions of the linear relaxation of IP if and only if A is TU.*

Several facts about TU matrices are very important and will be helpful

in the following.

Theorem 5 *A $m \times n$ matrix A is TU if*

1. $a_{ij} \in \{0, 1, -1\} \qquad \forall i, j : 1 \leq i \leq m, 1 \leq j \leq n$
2. *There are at most two non-zero entries in each row:*

$$\sum_{i=1}^{m} |a_{ij}| \leq 2, \quad \forall j = 1, \ldots, n$$

3. *There exists a partition (M_1, M_2) of the m rows such that for any columns with two non-seros in rows i and j, there is one row in M_1 and one row in M_2.*

$$\left| \sum_{i \in M_1} |a_{ij}| - \sum_{i \in M_2} |a_{ij}| \right| \leq 1 \qquad \forall i : 1 \leq i \leq n$$

■

3.5.3 Branch and bound

One of the most intuitive method to solve an IP would be to completely enumerate all possible assignments and to check whether they respect all constraints and then to take the assignment having the smallest objective value. However the number of such assignments is exponential (in the number of variables), making this enumeration infeasible in practice.

A solution not to fully enumerate the set of all asignments is to use bounds provided by the linear relaxation LP_{IP} of the IP. If we denote by $\mathcal{P}_{LP}$ the set of solutions of the linear relaxation and by $\mathcal{X}$ the set of all solutions of the IP, then we have $\mathcal{X} \subseteq \mathcal{P}_{LP}$. If we denote the optimal fractional solution of LP_{IP} by x^*_{LP}, its objective value by z^*_{LP} and the optimal objective value of IP by z^* then

1. If x^*_{LP} is integral, i.e. $x^*_{LP} \in \mathcal{X}$, then x^*_{LP} is the optimal solution of IP
2. $z_{LP} \leq z^*$
3. If LP_{IP} is infeasible, i.e. $\mathcal{P}_{LP} = \emptyset$, then $\mathcal{X} = \emptyset$ and IP is also infeasible.

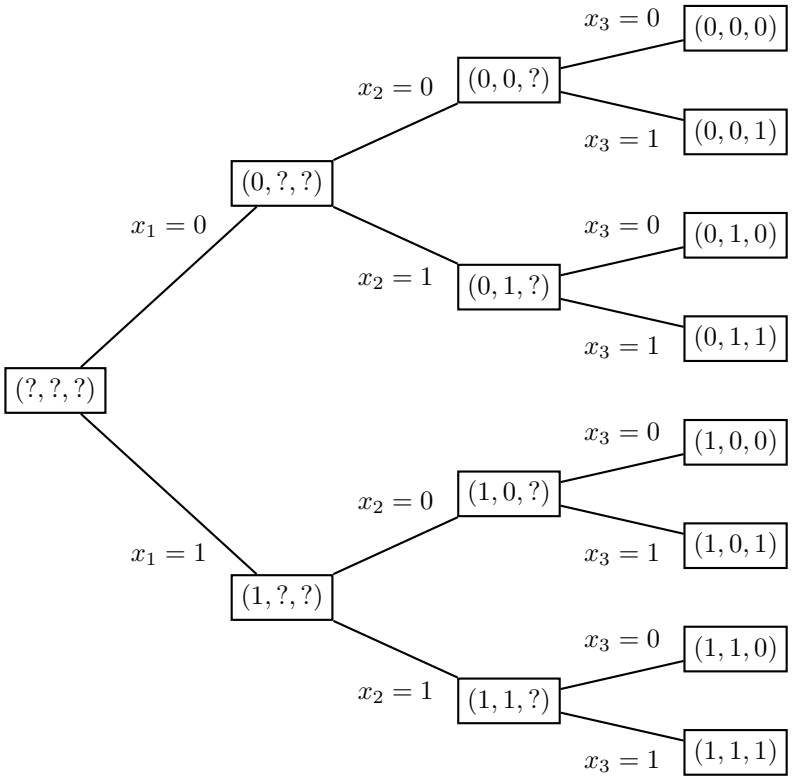

Figure 3.3: Full enumeration tree of a 0-1 IP with three variables. No pruning is done. At each arc, we fix a variable, until all variables are fixed. If the assignment respect all constraints, the current node represents a solution to the IP

This leads to consider solving IP by exploring a binary enumeration tree, solving the linear relaxation at each node and pruning branches of this tree accordingly to x^*_{LP}, z^*_{LP}. If we have a IP solution x_{IP} with cost z, then

Pruning by optimality If the solution x^*_{LP} at the current node is integral, then we can prune the current branch as we know there are no strictly better solution down in the current branch.

Pruning by bound If the optimal linear cost z^*_{LP} in the current node is such that $z^*_{LP} \geq z$ then we can prune the current branch as we know we cannot find a strictly better solution than x_{IP} in the current branch.

Pruning by Infeasability If the linear relaxation of the current node is infeasible, then we can prune the current branch as we know there is no (integral) solution.

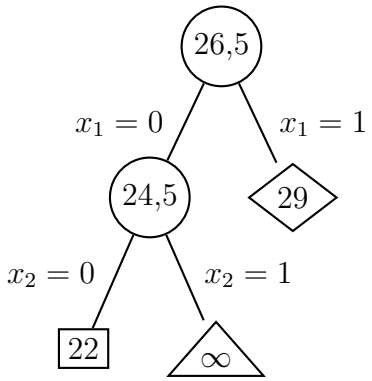

Figure 3.4: Branch-and-Bound: the three types of pruning are illustrated. Rectangular nodes represents nodes whose linear relaxation gives an integral optimal solution. Triangle nodes are pruned because the linear relaxation has no feasible solution. Diamond nodes are pruned because the linear relaxation gives lower bound greater than the best integral solution found so far. We made the hypothesis the search tree was explored by DFS.

3.6 Lagrangean relaxation

Lagrangean Relaxation aims at solving IP problems by removing difficult constraints. Imagine we need to solve the following problem

$$\begin{aligned} z &= \min cx \qquad (IP) && (3.16)\\ Ax &\geq b && (3.17)\\ Dx &\geq d && (3.18)\\ x &\in \mathbb{Z}^n_+ && (3.19) \end{aligned}$$

and that this problems without constraints (3.18) is easy to solve (for instance by a polynomial-time algorithm). One way to find a lower bound on the optimal value z is just to forget constraints (3.18) and to solve this easier relaxation. However this lower bound may be weak because a lot of constraints are not considered. Lagrangean Relaxation aims at handling these complicating constraints (3.18) in the objective function. Define the following problem:

$$z(\lambda) = \min cx + \lambda(d - Dx) \qquad (IP(\lambda)) \qquad (3.20)$$

$$Ax \geq b \tag{3.21}$$
$$x \in \mathbb{Z}^n_+ \tag{3.22}$$

.

It can be proven that $IP(\lambda)$ is a relaxation of IP; we have $\mathcal{X}_{IP} \subseteq \mathcal{X}_{IP(\lambda)} and z \geq z(\lambda)$ [Wol98, Proposition 10.1]. The vector λ can be viewed as a penalty term for constraints (3.18) that are not respected. The quality of the lower bound obtained by solving $IP(\lambda)$ depends strongly on the vector λ. Thus what we want is to solve the following problem called the Lagrangian Dual

$$w_{LD} = \max_{\lambda \in \mathbb{R}^n_+} z(\lambda) \qquad (LD) \tag{3.23}$$

The structure of the Lagrangian Dual is problem depended. If we define the feasible region of problem (3.16)-(3.19) by $X = \{x \in \mathbb{Z}^n_+ \mid Ax \geq b\}$, then it can be shown that [Wol98, Theorem 10.3]

Theorem 6 (Strength of the Lagrangean Dual)

$$w_{LD} = \min\{cx \mid Dx \geq d, x \in conv(X)\}$$ ■

This implies that if $conv(X) = \{x \in \mathbb{R}^n \mid Ax \geq b\}$ then the Lagrangean Dual is not stronger than the LP relaxation.

The remaining problem is to find the Lagrangean multipliers λ that maximises $z(\lambda)$. It is shown in many references (see [Wol98] for instance) that the Lagrangean Dual is equivalent to maximize a piecewise linear function.

A well-known technique to solve such problems is the *subgradient algorithm*. We denote by $x^*(\lambda_k)$ the optimal solution of $IP(\lambda_k)$ of weight $z(\lambda_k)^*$ and by $Dx = d$ the matrix representation of dualized constraints (3.18). The subgradient algorithm is presented in Algorithm 3.1.

One of the most used definition of m_k is the following

$$\mu_k = e_k \frac{z(\lambda_k) - \underline{w}}{||d - Dy(\lambda_k)||^2} \tag{3.24}$$

with $0 < e_k \leq 2$ and $\underline{w}$ is an lower bound of $z(\lambda_k)$. Sometimes we don't have such a lower bound, then we compute a good upper bound by mean a

SubGradient()

1: $\lambda \leftarrow \lambda_1, k = 1$
2: **repeat**
3: Solve $IP(\lambda_k)$ with optimal solution $x^*(\lambda_k)$
4: $\lambda_{k+1} \leftarrow \lambda_k - \mu_k(d - Dy(\lambda_k))$
5: $k \leftarrow k + 1$
6: **until** $|x^*(\lambda_{k-1}) - x^*(\lambda_k)| > e$
7: **return** $x^*(\lambda_k)$

Algorithm 3.1: Subgradient algorithm

heuristic on the non-relaxed problem IP. Properties and justification of this formula can be found in [HWC74].

To summarize, in order to compute a Lagrangean lower bound of our original IP problem, we use the subgradient algorithm to slightly modify the weights in the objective function and we use a specialized algorithm to solve the new $IP(\lambda)$ problem. We loop between these two steps until the $z(\lambda_k)$ does not increase enough.

3.7 Column generation

Column Generation is used when the problem to solve is too large to be feeded into a solver at once. In the previous section we presented the Lagrangean relaxation principle that allows to decrease the number of rows in an optimization problem. Column Generation aims at doing the same thing with columns.

Column generation is also used when the structure of the constraints are too complex to achieve efficient formulation. In this case it can be better to express the solutions of the problem explicitely. This leads to problem with a huge number of columns, that cannot be tackled directly by LP solvers. Then column generation can determine whether we achieved optimality or not by using pricing procedures.

It begins to solve the initial problem restricted to a subset of the columns. Then it computes columns that are good candidates to be part to the optimal solution. It loops on this process until it can determine that all columns not generated cannot be part of the optimal solution.

Column generation is also called branch-and-price. This is different from

branch-and-cut, where we add valid inequalities (thus inserting rows) and branch whenever we cannot find such inequalities anymore. There exist one more link between cutting and generating columns; the pricing problem in column generation is in fact a cutting problem for the dual problem and vice-versa.

There exist two main important points when designing a branch-and-price algorithm:

- Branching on variables - as it is usually done - can be inefficient because it breaks the structure of the pricing problem.
- Solving LP relaxation and subproblem to optimality could be too time-consuming. Special actions should be taken in order to fix this problem, depending on the problem.

A very good review on Column Generation can be found in [BJN$^+$98].

3.7.1 Principles

Suppose that we want to solve the following linear problem, called the Master Problem

$$
\begin{aligned}
(MP) \qquad \min \sum_{\mathfrak{c} \in \mathcal{C}} c_{\mathfrak{c}}.x_{\mathfrak{c}} \qquad & \text{subject to} && (3.25)\\
\sum_{\mathfrak{c} \in \mathcal{C}} \mathfrak{c}.x_{\mathfrak{c}} \quad & \geq \quad b && (3.26)\\
x \quad & \geq \quad 0 && (3.27)
\end{aligned}
$$

where $x \in \mathbb{Q}^n$, $\mathcal{C}$ is a set of columns, i.e. a subset of $\mathbb{Q}^m$ and $c_{\mathfrak{c}}$ is the cost of column $\mathfrak{c}$.

Then the dual of (MP) is

$$
\begin{aligned}
(DMP) \qquad \max \sum_{i=0}^{m} b_i.\pi_i \qquad & \text{subject to} && (3.28)\\
\bar{\mathfrak{c}} = \sum_{i=0}^{m} \mathfrak{c}_i \pi_i \quad & \leq \quad c_{\mathfrak{c}} \qquad \forall \mathfrak{c} \in \mathcal{C} && (3.29)\\
\pi \quad & \geq \quad 0 && (3.30)
\end{aligned}
$$

The challenge is that the set $\mathcal{C}$ is usually too big to be solved at once. Thus we define a Restricted Master Problem

$$(RP) \qquad \min \sum_{\mathfrak{c} \in \mathcal{C}'} c_{\mathfrak{c}}.x_{\mathfrak{c}} \qquad \text{subject to} \tag{3.31}$$

$$\sum_{\mathfrak{c} \in \mathcal{C}'} \mathfrak{c}.x_{\mathfrak{c}} \geq b \tag{3.32}$$

$$x \geq 0 \tag{3.33}$$

where $\mathcal{C}' \subseteq \mathcal{C}$. Because $\mathcal{C}'$ is smaller than $\mathcal{C}$, (RP) can be much more easily solved than (MP). Observe that any solution x of (RP) is also a solution of (MP).

Now suppose we have an optimal solution for (RP) and its dual, respectively x^r and $\pi^r \geq 0$. We define the following

Definition 6 (Reduced cost) *The reduced cost associated to a column $\mathfrak{c}$ wrt dual values π is*

$$\bar{\mathfrak{c}} = c_{\mathfrak{c}} - \sum_{i=0}^{m} \mathfrak{c}_i \pi_i$$

Then π^r is a solution of (DMP) only if all reduced cost are positive, i.e. $\bar{\mathfrak{c}} \geq 0, \forall \mathfrak{c} \in \mathcal{C}$. If all reduced costs are positive, then we have a solution x^r of (MP) and π^r of (DMP) whose costs are equal: $\sum_{\mathfrak{c} \in \mathcal{C}} c_{\mathfrak{c}}.x^r_{\mathfrak{c}} = \sum_{i=0}^{m} b_i.\pi^r_i$. From the Weak duality theorem (2) x^r is then optimal for (MP).

The main idea behind column generation is to select a small subset $\mathcal{S}$ of columns in $\mathcal{C}$, to solve the associated restricted problem. Then it solves the following Pricing Problem

$$(PP) \qquad \zeta = \min \quad c_{\mathfrak{c}} - \sum_{i=0}^{m} \mathfrak{c}_i \pi^r_i \qquad \text{subject to} \tag{3.34}$$

$$\mathfrak{c} \in \mathcal{C} \tag{3.35}$$

Then if $\zeta \geq 0$, then it means that x^r is optimal and we can stop. Otherwise the algorithm adds the optimal column $\mathfrak{c}^*$ of (PP) to (RP), solves (RP) and so on until the optimal solution to (MP) is found.

Part II

The problem: The set covering problem

Chapter 4

The set covering problem

The set covering problem (SC) can be defined as follows. Let $\mathcal{U} = \{1, \ldots, m\}$ be a set of m elements. Let X be a collection of subsets of $\mathcal{U}$, i.e. $X = \{S_1, \ldots, S_n\}$ where $S_i \subseteq \mathcal{U}, (1 \leq i \leq n)$ and let c_j be a weight associated to each subset $S_j, 1 \leq j \leq n$. SC calls for a subset T of indices of X covering $\mathcal{U}$; $T \subseteq \{1, \ldots, n\} \mid \bigcup_{i \in T} S_i = \mathcal{U}$ and such that $\sum_{j \in T} c_j$ is minimum. An example of such a problem is illustrated in Fig.4.1.

SC problems are very difficult. The results in [ALM+98] imply that it does not have a polynomial time approximation scheme unless P = NP; that is, there exists a constant $\epsilon > 0$ such that the problem cannot be approximated in polynomial time with a ratio smaller than $1 + \epsilon$ unless P = NP. More negative results about the complexity of this problem can be found in [LY94].

The integer programming formulation of SC is

$$SC^* = \min_{x \in \mathcal{B}^n} \sum_{1 \leq j \leq n} c_j x_j \qquad (SC)$$

$$\mathcal{U} = \{1, 2, 3, 4, 5\}$$
$$S_1 = \{1, 3, 5\},\ S_2 = \{1, 2, 4\},\ S_3 = \{5, 2\},\ S_4 = \{1, 3, 2\}$$
$$c_j = 1, \quad (1 \leq j \leq 4)$$

$$Sol = \{1, 2\}$$

Figure 4.1: Example of a set covering problem. It is obvious that there is no subset S_i covering $\mathcal{U}$ in itself. We observe $S_1 \cup S_2$ covers entirely $\mathcal{U}$, thus $\{1, 2\}$ is a set cover of $\mathcal{U}$ of minimum cardinality.

subject to

$$\sum_{1 \leq j \leq n} a_{ij} x_j \geq 1 \quad \forall i \in \mathcal{U} \tag{4.1}$$

$$x_j \in \mathcal{B} \quad \forall j = 1 \ldots, n \tag{4.2}$$

where $a_{ij} = 1$ iff $i \in S_j$. $\mathcal{B} = \{0, 1\}$.

A lot of real-life applications can be formulated in terms of a SC problem: crew assignment and scheduling [SZSF02, HP93, CFT99], construction of optimal logical circuits [Pie68], location of emergency facilities [JM92, TSRB71], routing problems (VRP will more deeply considered later). Additionally, a lot of well-known combinatorial problems can be easily formulated by mean of the SC problem such as vertex cover, dominating set and independent set.

A lot of work has been done to solve this problem either to optimality or to a nearly optimal solution, usually using integer programming. Two good reviews are [CFT98, CNS97].

Chapter 5

A general CP approach to SCP

5.1 The global constraint SC

The aim of this CP approach is to design a global constraint for SC having the form $SC(N, T, \mathcal{U}, X = \{S_1, \ldots, S_n\}, Cost)$ where

- $\mathcal{U} = \{1, \ldots, m\}$ is a set of elements
- N is an integer
- X is a collection of subsets $S_i \subseteq \mathcal{U}, (1 \leq i \leq n)$
- T is a subset of the indices of X: $T \subseteq \{1, \ldots, n\}$
- $Cost$ is a function giving a weight for each subset S_i; $Cost : X \rightarrow \mathbb{Z}$. For sake of simplicity, we will denote $Cost(S_i)$ by c_i.

This constraint holds if and only if

$$\mathcal{U} = \bigcup_{i \in T} S_i \tag{5.1}$$

$$\sum_{i \in T} c_i = N \tag{5.2}$$

In the following, N and T will be considered as variables and $\mathcal{U}$, $S_i (i = 1, \ldots, n)$, $Cost$ will be considered constant. N is a finite-domain (FD) variable and T is a set variable [Ger97, Pug96]. $\overline{N}$ denotes the upper bound on

N and $\underline{N}$ denotes its lower bound: $\underline{N} \leq N \leq \overline{N}$. We use the same notations for T: $\underline{T} \subseteq T \subseteq \overline{T}$. We also define $\overline{\underline{N}} = \overline{N} \setminus \underline{N}$ and $\overline{\underline{T}} = \overline{T} \setminus \underline{T}$.

We observe that (5.1) is easy to satisfy, while, for small values of N, finding a T satisfying (5.2) is hard. This is why we need good lower bounds on N in order to prune the domains of N and T.

The objective of the filtering algorithm for this global constraint SC is to prune the domain of N and T by removing values that do not belong to any solution of this constraint.

5.2 Pruning of N

From the structure of SC, it is easy to see that the tightest upper bound on N is $N \leq \sum_{i \in \overline{T}} c_i$. Indeed if there exists a solution to the constraint (i.e. an assignment of variables N and T such that (5.1) and (5.2) hold), then taking all subsets, $N = \sum_{i \in \overline{T}} c_i, T = \overline{T}$, would be a solution to the global constraint SC too. Hence the pruning rule is $\overline{N} = \min(\overline{N}, \sum_{i \in \overline{T}} c_i)$.

However computing a lower bound on N is more difficult. Computing the tightest lower bound on N requires to solve SC to optimality, i.e. to compute SC^*. This problem is NP-Hard [Kar72]. In this paper we use a lower bound of SC^* to prune the domain of the variables. To compute such a lower bound one solves a relaxation of SC. Let denote such a relaxation by $SCRel$ and its optimal value by $SCRel^*$. Then we have

$$lb_{SC} = SCRel^* \leq SC^*$$

The pruning rule is $\underline{N} = \max(\underline{N}, lb_{SC})$. A generic filtering algorithm for pruning the domain of N is sketched in Algorithm 5.1.

$computelbSC(\mathcal{D})$	
$SCRel = buildSCRel(\overline{\mathcal{D}(T)})$	// Build the relaxation $SCRel$ of SC
$SCRel^* = solve(SCRel)$	// compute the optimum value of $SCRel$
return $\{(N, [\ \underline{\mathcal{D}(N)}; SCRel^*[\ )\}$	

Algorithm 5.1: Filtering algorithm for the SC constraint

5.3 Pruning of T

T is a set domain variable. The pruning algorithm should prune the domain of T according to constraints (5.1) and (5.2). From the first constraint we can derive the following two pruning rules:

(P1) The constraint SC does not hold if $\mathcal{U}$ cannot be covered by the available subsets, thus $\exists e \in \mathcal{U}, \forall i \in T \; e \notin S_i \rightarrow fail$

(P2) Because the indices in T should cover $\mathcal{U}$, when there is only one subset S_i covering an element e, S_i should be part of the cover. So we have $\exists e \in \mathcal{U}, \exists! i : \; e \in S_i \rightarrow i \in T$

Without constraint (5.2), these pruning rules would achieve optimal pruning (all partial solution could be extended to a feasible solution). In order to prune constraint (5.2), two other pruning rule should be considered.

(PA) Remove from T the indices of subsets that do not belong to any cover of weight in $[\underline{N}; \overline{N}]$: $\overline{T} \leftarrow \overline{T} \setminus \{i \in \overline{T} \setminus \underline{T} \mid \neg SC(N, T \cup \{i\}, \mathcal{U}, X)\}$

(PB) Put in T all indices such that S_i belongs to all covers of weight in $[\underline{N}; \overline{N}]$: $\underline{T} \leftarrow \underline{T} \cup \{i \in \overline{T} \setminus \underline{T} \mid \neg SC(N, T \setminus \{i\}, \mathcal{U}, X)\}$

Unfortunately, we saw that achieving arc-consistency for the SC constraint is NP-Complete, thus these last two rules can only be approximated. We will use a lower bound to prune according to constraint (5.2):

(P'A) $\overline{T} \leftarrow \overline{T} \setminus \{i \in \overline{T} \setminus \underline{T} \mid lb_{SC}(N, T \cup \{i\}, \mathcal{U}, X) > \overline{N}\}$

(P'B) $\underline{T} \leftarrow \underline{T} \cup \{i \in \overline{T} \setminus \underline{T} \mid lb_{SC}(N, T \setminus \{i\}, \mathcal{U}, X) > \overline{N}\}$

Pruning (P'A) (resp. (P'B)) can be done by mean of a shaving technique: we remove from $\overline{T}$ (resp. add in $\underline{T}$) each value $i \in \overline{T} \setminus \underline{T}$ at turn and recompute a new lower bound lb_i on N. If $lb_i > \overline{N}$, then we must put i in $\underline{T}$ (resp. remove i from $\overline{T}$).

5.4 Existing approaches

5.4.1 Relation between SC and other constraints

As far as we know, SC has never been tackled directly in CP. However some work has been done for the constraint $NValue$ counting the number of distinct values used by a vector of variables, that is close to SC. This constraint was first introduced in [PR99]. This is a generalization of the well-known $AllDiff$ constraint [Rég94, van01].

Bessière et al. [BHH+05] decomposed $NValue$ into two constraints:

$$AtLeastNValue(N, X = \{X_1, \ldots, X_m\})$$

and

$$AtMostNValue(N, X = \{X_1, \ldots, X_m\})$$

where N is an integer and X_i are FD variables having $D(X_i)$ as domain. $AtMostNValue$ holds if the assignment of the m variables X_i uses at most N different values. Formally AtMostNValue holds iff $|\{X_i : 1 \leq i \leq m\}| \leq N$ and AtLeastNValue holds iff $|\{X_i : 1 \leq i \leq m\}| \geq N$. We now show that $AtMostNValue$ is equivalent to SC with uniform weights ($c_j = 1, \forall j$).

Proposition 7 *Unicost SC is strictly equivalent to AtMostNValue (**NV**).*

Proof Define $\mathcal{U} = \{1, \ldots, m\}$ and $S_i = \{j \mid i \in D(X_j)\}, \forall i \in \bigcup_{1 \leq i \leq m} D(X_i)$. Entailment of the original $AtMostNValue$ is equivalent to entailment of $SC(N, T, \mathcal{U}, X)$ and vice versa. set covering problem is NP-Hard, this shows that computing the minimum value for N is also NP-Hard.

This shows that every filtering algorithm developped for one of these two problems can directly be used to prune variables of the other problem. The following of this section applies existing techniques to solve $AtMostNValue$ to SC.

Interesting Facts for Computing a Lower bound on N

Before describing the existing approaches to compute lower bounds on N, we introduce the intersection graph@intersection graph G_X $G_X = (V, E)$ of a collection of subsets $X = \{X_1, \ldots, X_m\}$, where $V = \mathcal{U}$ and $E =$

$\{(i,j) \mid D(X_i) \cap D(X_j) \neq \emptyset\}$. We can observe that, for a given instance of SC, finding the largest independent set (set of vertices being pairwise non-adjacent) of G_X is equivalent as solving the dual problem of SC, that is the set packing problem. In the case of a unicost SC ($c_j = 1, \forall j = 1, \ldots, n$), the formulation of the set packing problem can be formulated as

$$SP^* = \max_{y \in \mathcal{B}^m} \sum_{1 \leq i \leq m} y_j \qquad (SP)$$

such that

$$y_i + y_j \leq 1 \quad \forall (i,j) \in E$$

$$y_i \in \{0,1\} \quad \forall 1 \leq i \leq m$$

The set packing problem is an NP-Hard problem. If we denote SC^*(resp. SP^*) as the optimal value for the set covering problem (resp. set packing problem), SCL^*(resp. $SPLin^*$) as the optimal solution of the linear relaxation of the set covering problem (resp. set packing problem), we have from optimisation theory (see Section 3.4) that

$$SP^* \leq SPLin^* = SCL^* \leq SC^* \tag{5.3}$$

Thus a lower bound on SP^* is also a lower bound lb_{SC} of SC^*. If we denote the independance number of a graph G by $\alpha(G)$, then from (5.3),

$$lb_{SC} = \alpha(G_X) = SP^* \leq SC^* \tag{5.4}$$

5.4.2 Four lower bounds on N

Four approaches exist for computing a lower bound on N.

Linear relaxation of SC (SCL^*) If we relax the integrality constraint of SC, i.e. replacing $x_i \in \{0,1\}$ by $0 \leq x_i \leq 1$ we obtain the linear relaxation of SC, that we denote SCL. From (5.3) we have that $lb_SC = SCL^*$ is a lower bound on SC^*. This bound is better than any bound on SP^*. However it is relatively expensive to compute.

Ordered interval algorithm (OI) In [BCT02], Beldiceanu deals with N-Value where the domain of each variables X_i is an interval. In this special case, he achieves bound-consistency in polynomial time. If

we denote m domain variables whose domains are intervals by $I = \{I_1, \ldots, I_m\}$, then this algorithm computes $\alpha(G_I)$. If we denote the smallest enclosing interval of the domain of each variable X_i by I_i, then this algorithm can be used to compute a lower bound of $\alpha(G_X)$, as G_I contains at least all edges of G_X and eventually some others, thus $lb_{SC} = \alpha(G_I) \leq \alpha(G_X)$.

Greedy algorithm (MD - for Minimum Degree) Consider the graph $G_X = (V, E)$. Let $\Gamma(v), v \in V$ be the set of neighbours of node v and $\Gamma(S) = \bigcup_{v \in S} \Gamma(v)$. Initially, set $S = \emptyset$. Choose the node $v \in V \setminus (S \cup \Gamma(S))$ of minimum degree. Add v to S and loop until $S \cup \Gamma(S) = V$. At the end S will be an independent set of G_X and $lb_{SC} = |S|$ will be a lower bound of $\alpha(G_X) = SP^* \leq SC^*$.

Turán@turan's approximation (TA) Use the lower bound for $\alpha(G_X)$ proposed by Turán in [Tur41]: $lb_{SC} = \left\lceil \frac{n^2}{2m+n} \right\rceil \leq \alpha(G_X) = SP^* \leq SC^*$.

From equation(5.3), we deduce that SCL^* is a tighter bound than the other three. MD is at least as strong as TA. OI is sometimes better, sometimes worse, than MD and TA (see [BHH+05] for details).

Experience will compare these four methods for pruning the domain of N with the method described in the next chapter.

Part III

A Global Constraint for the Set Covering Problem

Chapter 6

A new SC lower bound: *2SC*

In this section we present another relaxation of SC. It can be used in a CP framework to compute a lower bound of SC^* and to prune the domain of N as explained in the previous section. The main advantage of this relaxation is that it can be solved incrementaly, being a good choice in a search-tree scheme as will be pointed out in the experimental section.

In the first section we decribe the principles of this relaxation. Then we describe an incremental algorithm for computing the optimal value of the relaxation 2SC. First we will give six necessary and sufficient conditions for an edge cover R of a bipartite graph to be of minimum weight. Then we will describe an algorithm maintaining the five first conditions as invariant and looping in order to end with the sixth condition holding. Third we will explain how we can update the data structure used by the algorithm to reflect small changes in the bipartite graph (removal/insertion of edges). Finally we will enumerate a few algorithmic enhancements of this algorithm. In the last section pf this chapter we will describe some heuristics for computing the bipartite graph needed in the relaxation 2SC.

6.1 Principles

An interesting relaxation is based on the fact that a set covering problem with subsets S_i containing exactly two elements, that will be denoted as 2SC, can be solved in polynomial time. This relaxation was explored in a strictly MIP context in [EDM92] for the set covering problem and in [LW79] for the set partitionning problem (i.e. the set covering with equality constraints instead of inequalities).

To solve $2SC$, we can build a graph G with one vertex representing one element of $\mathcal{U}$ and one edge representing one 2-set (a set with exactly two elements) S_i from the collection X. A minimum weight edge cover of G would represent a minimum set cover of 2SC. Every set covering problem SC can be decomposed into a 2SC problem in such a way that the graph G is bipartite. This is motivated by efficiency as computing minimum edge covers in general graph is computationaly more expensive (as for the matching problem).

Let us define $E(S_i)$ the decomposition of S_i in 2-sets:

$$E(S_i) = \{S_i^1, \ldots, S_i^k\}$$

.

such that $\forall i : 1 \leq i \leq n$,

$$|S_i^j| = 2 \qquad \forall S_i^j \in E(S_i) \tag{6.1}$$

$$S_i = \bigcup_{S_i^j \in E(S_i)} S_i^j \tag{6.2}$$

If we define weight c_i^j on the 2-subsets S_i^j such that

$$c_i = \sum_{S_i^j \in E(S_i)} c_i^j, \qquad (1 \leq i \leq n) \tag{6.3}$$

where c_i is the cost associated to the subset S_i, then 2SC will be a relaxation of SC because any set cover of SC will be a set cover for 2SC with exactly the same weight. Clearly a solution to the relaxation 2SC will provide a lower bound of the minimum weight of the original SC problem.

If we denote the optimal value of the 2SC problem as $2SC^*$, it can be shown that (see Section 6.4)

$$2SC^* \leq SCL^* \leq SC^* \tag{6.4}$$

This lower bound can thus be used in the propagator for the SC constraint. We now show how to compute this lower bound efficiently.

6.2 Incremental algorithm

The relaxation presented in the previous section wouldn't be of any interest if we could not solve it efficiently to optimality. The aim of this section is

to present a new incremental algorithm for the computation of $2SC^*$. We illustrate how we can solve 2SC and how we can update the optimal solution when we restrict the edge cover we want to find, for instance by imposing that all edges from the decomposition of a subset S_i must be part of the edge cover. This is very important in a search-tree scheme in order to prune domains very quickly.

The weighted 2SC problem introduced in the previous section has a lot of similarities with the maximum weighted matching of a graph. We therefore present a reasoning based on the Hungarian method [Kuh55] to solve this last problem. As in the previous section, we will assume the graph G built from 2SC is bipartite.

The aim of the 2SC problem is to find a minimum weight edge cover R of a bipartite graph $G = (V, E)$. The edge cover can be restricted: an edge $(i, j) \in SURE$ if and only if (i, j) *must* be part of the cover, $(i, j) \in REMOVED$ if and only if it *must not* be part of the cover. A node i will be defined as SURE if it is the endpoint of an edge being in SURE. A vertex v from G is critical wrt cover R if exactly one edge from R is incident to v. A vertex v is non-critical otherwise. For now we consider edge weights being constant.

In order to have a strong basis for the following, here is the IP formulation of 2SC:

$$2SC^* = \min \sum_{e \in F} c_i^j y_i^j + \sum_{e \in SURE} c_i^j y_i^j \tag{6.5}$$

$$\sum_{(i,j) \in E} y_i^j \geq 1 \quad \forall \; vertex \; i \notin SURE \tag{6.6}$$

$$0 \leq y_i^j \leq 1 \quad \forall (i, j) \in F \tag{6.7}$$

where $F = E \setminus (SURE \cup REMOVED)$ is the set of edges which we do not know whether they are part of an optimal constrained edge cover. Constraint (6.7) is necessary as we allow negative weights c_i^j. It should be noted that in the previous formulation, we did not constrain y_i^j to be integral. Indeed the matrix defining the constraint (6.6)-(6.7) meets the three conditions of Theorem 5 and is thus totaly unimodular, so the objective value of the optimal linear solution is equal to the objective value of the optimal integral solution.

Results in optimisation theory (commonly called the KKT conditions, see [Kar39, KT50]) give necessary and sufficient conditions for a solution of a LP problem to be optimal. Let $\pi(i)$ be the dual variables of constraints (6.6).

The KKT conditions state that a set of edges (defined by y_i^j's) is a cover and is optimal iff

(i). $\pi(i) \geq 0 \quad \forall node\ i,\ \pi(i) = 0 \quad \forall\ i \in SURE$

(ii). $\bar{c}_i^j \leq 0$ if $(i,j) \in F$ and $y_i^j = 1$

(iii). $\bar{c}_i^j \geq 0$ if $(i,j) \in F$ and $y_i^j = 0$

(iv). $\sum_{(i,j)\in E} y_i^j - 1 \geq 0 \quad \forall\ vertex\ i$

(v). $\pi(v) = 0$ if vertex v is non-critical, $\forall v \in V_2$

(vi). $\pi(v) = 0$ if vertex v is non-critical, $\forall v \in V_1$

where the reduced cost are defined by $\bar{c}_i^j = c_i^j - \pi(i) - \pi(j)$. Algorithm 6.1 solves weighted 2SC optimaly and is based on the conditions (i)-(vi). Beginning with a solution respecting (i)-(v), the algorithm loops in such a way that strictly less vertices i don't respect (vi) at each iteration. It ends when no such vertex i exists.

The intuitive idea behind the algorithm (as for matching algorithms) is to select a node v_0 not respecting (vi) and to decrease the number of edges incident to v_0 being in the cover. This is done by finding alternating paths. An alternating path wrt a cover R from v_0 to v_k is a sequence $p = [v_0, v_1, \ldots, v_k]$ of nodes such that edges (v_i, v_{i+1}) are alternatevely from R and from $E \setminus R$. If $v_0 = v_k$ then exactly one edge of (v_0, v_1) and (v_{k-1}, v_k) belongs to R. An admissible alternating path L from v_0 to v_k is defined as an alternating path with

- Either $(v_0, v_1) \in E \setminus R$ or v_0 is non-critical
  AND
- Either $(v_{k-1}, v_k) \in E \setminus R$ or v_k is non-critical

The following result, similar to [Ber57], states that a cover augmented by an admissible alternating path remains a cover.

Proposition 8 *If L is an admissible path wrt a cover R of G, then the set of edges $R' = R \oplus L = (R \setminus L) \cup (L \setminus R)$ is a cover of G.*

Proof By the way we defined R', it is built from R by removing all edges from $R \cap L$ and adding edges from $L \setminus R$. If $L = [v_0, \ldots, v_k]$ then , for $x = v_2, \ldots, v_{k-1}$, we will remove from R an incident edge of x and add another edge from $L \setminus R$ incident to x. Thus $v_2, \ldots, v_{k-1}$ are still covered by R'. If $(v_1, v_2) \in E \setminus R$ then R' will contain one more edge incident to v_1 than R and v_1 will be covered by R'. If v_1 is non-critical then R' will have one less edge incident to v_1 than R. But as v_1 is non-critical, v_1 will still be covered by R'. We can do the same reasoning for the edge $(k-1, v_k)$. Thus all nodes are covered by R'.

The algorithm finds a node i not respecting (vi) and augments the current cover by an admissible path from i (with the first edge being in the cover). This strictly decreases the number of edges incident to i and being in the cover. After some iterations, node i will become critical and will thus respect (vi). The algorithm loops until there is no more such node. The algorithm explores admissible path of increasing weight (shorter admissible path are prefered).

Correctness of the Algorithm The inner loop (7)-(15) of Algorithm 6.1 modifies dual values π in order to find new admissible paths from node i. We can observe that if $\delta = \beta$ or $\delta = -\gamma$ then the tree T strictly grows. If $\delta = \alpha$ then an admissible path is found (i is non-critical and $e_k \in E \setminus R$). As the loop finishes if $\delta = \alpha$ (because $\exists w \in V_1^T \mid \pi(w) = 0$) and T cannot grow forever, the inner loop (7)-(15) will iterate a finite number of times.

The outer loop (5)-(18) iterates until node i respects condition (vi). After we modify the current edge cover $R \leftarrow R \oplus p$, there will be one less edge incident to node i in R because the first edge $(v_0, v_1) \in R$ for all admissible path $[v_0, v_1, \ldots, v_k]$ in T (R is still an edge cover from Proposition 8). Thus after a finite number of iterations, node i will become critical and the outer loop (5)-(18) will end.

Updating of the bipartite graph When we need to restrict our cover to contain or not to contain specific edges, we cannot simply move these edges to a different linked list and mark them as $SURE$ or $REMOVED$. This

computeMin2SC(2SC = (G,C,M))

PRE: - $G = (V_1, V_2, E)$ a bipartite graph, $C = \{c_i^j\}$: weights for all edge $(i,j) \in E$, $M \subseteq E$

POST: - $R \cup M$ is a minimum weight edge cover EC^* of G such that $M \subseteq EC^*$

1: Let $R \leftarrow E$ and define initial values for $\pi(v) \geq 0$ such that (i)-(v) hold
2: Verify R cover all nodes of G. If not then return IMPOSSIBLE
3: Let $[v_1, \ldots, v_n]$ be an enumeration of all nodes of V_1
4: **for** $i \mid \nexists j, (i,j) \in M$ **do**
5: **while** v_i is non-critical and $\pi(v_i) > 0$ **do**
6: $T = (V_1^T, V_2^T, E^T) \leftarrow computeT(G = (V,E), R, v_i)$
7: **while** $\nexists w \in V_1^T : \pi(w) = 0$ and $\nexists w \in V_2^T, w$ non-critical **do**
8: $\alpha \leftarrow \min\left\{\{+\infty\} \cup \{\pi(v) \mid v \in V_1^T\}\right\}$
9: $\beta \leftarrow \min\left\{\{+\infty\} \cup \{\overline{c}_i^j \mid (i,j) \in E \setminus R, i \in V_2^T, j \notin V_1^T\}\right\}$
10: $\gamma \leftarrow \max\left\{\{-\infty\} \cup \{\overline{c}_i^j \mid (i,j) \in R, i \in V_1^T, j \notin V_2^T\}\right\}$
11: $\delta \leftarrow \min(\alpha, \beta, -\gamma)$
12: $\pi(i) \leftarrow \pi(i) - \delta, \forall i \in V_1^T$
13: $\pi(j) \leftarrow \pi(j) + \delta, \forall i \in V_2^T$
14: $T = (V_1^T, V_2^T, E^T) \leftarrow computeT(G = (V,E), R, v_i)$
15: **end while**
16: Let p be the alternating path from v_i to w in T
17: $R \leftarrow R \oplus p$
18: **end while**
19: **end for**
20: **return** $R \cup M$

Algorithm 6.1: Algorithm for weighted 2SC

is due to the fact that we still want the invariant (i)-(v) to hold after this change.

Before presenting how we modify the structure of the graph, we introduce an operation `removeFromCover(i,j)` that allows to make free an edge (i,j) from the cover while satisfying the invariant. When we want to remove an edge (i,j) from the cover, we must ensure that nodes i and j are still covered by other edges (condition (iv)). If not, for the uncovered node i (resp. j), we must add the edge of minimum reduced cost $\bar{c}_{min}$ in the cover and increase the $\pi(i)$ (resp. $\pi(j)$) value in order to make this reduced cost negative.

When we want to put an edge (i,j) in $REMOVED$, we can use the op-

computeT$(G = (V, E), R, v_i)$

PRE: - $v_0 \in V_1 \cup V_2, R \subseteq E$

POST: - Return a tree built with the alternating paths beginning at v_0: $T = (V_A^T, V_B^T, E^T)$ such that

- $V_A^T = \{v_k \in A \mid \exists$ an alternating path wrt R $[v_0, v_1, v_2, \ldots, v_{k-1}, v_k]$ with $(v_0, v_1) \in R, \overline{c}_{v_{i-1}, v_i} = 0, \forall i = 1, \ldots, k$. $V_A^T = \{v_0\}$ if no such path exists.
- $V_B^T = \{v_k \in B \mid \exists$ an alternating path wrt R $[v_0, v_1, v_2, \ldots, v_{k-1}, v_k]$ with $(v_0, v_1) \in R, \overline{c}_{v_{i-1}, v_i} = 0, \forall i = 1, \ldots, k$. $V_B^T = \emptyset$ if no such path exists.
- $E^T = \{(v_i, v_{i+1}) \mid \exists$ an alternating path wrt R $[v_0, v_1, v_2, \ldots, v_i, v_{i+1}, \ldots, v_{k-1}, v_k]$ with $(v_0, v_1) \in R, \overline{c}_{v_{i-1}, v_i} = 0, \forall i = 1, \ldots, k$

Algorithm 6.2: Algorithm to compute an alternating tree

eration `removeFromCover(i,j)` and then just mark (i, j) as being removed. The invariant will hold after removal.

To put (i, j) in $SURE$ we must set $\pi(i) = \pi(j) = 0$ in order to modify all the reduced costs accordingly (this can be seen by applying the KKT conditions on the same problem without the cover constraint (iv) for nodes i and j. Then because we could have decreased $\pi(i)$ or $\pi(j)$, some edge (i, k) or (i, k) incident to these nodes and being in the cover could have a positive reduced cost. If this is the case, we must remove them from the cover by using the operation `removeFromCover`.

When we want to make an unknown edge (i, j) that was previously in $SURE$, we can put it in the cover if $\bar{c}_{ij} \leq 0$. Otherwise we set it free. In this last case, we must check that both i and j are still covered. If not we put in the cover the edge of minimum reduced cost and we increase π values accordingly.

To swap an edge (i, j) from $REMOVED$ to unknown, we mark it as covered or free depending solely on its reduced cost.

After having updated the bipartite graph, applying the algorithm will make (vi) hold. The algorithm should only search for a few new admissible

paths of negative weight. Such a local adjustment can also be performed after the weights are changed (see next section). This is very attractive for us because algorithms for propagators should be highly incremental in order to be fast through all the search tree. This advantage is demonstrated in the experimental results (see Section 10.1) where we can observe a small overall time per call to the propagator.

Algorithmic improvements We can improve this algorithm similarly as in [MN99, p.423-424]. If we consider the inner loop (7)-(15), we define δ_k as the computed value of δ at the k-th iteration of this loop and $\Delta_k = \delta_1 + \ldots + \delta_k$. Δ is the total change during the execution of the inner loop, i.e. if the inner loop iterates K times, then $\Delta = \Delta_K$. Let $\mu(w)$ be the shortest distance from v_i to w with respect to the absolute value of the reduced costs of the edges defined by the dual values π_i computed before going into the inner loop for the i-th time (i.e. the dual values at line (6)). We can observe the following

Lemma 9 *For all nodes w, w is added into T at the iteration j iff $\Delta_j = \mu(w)$.*

Proof The proof is direct as T is defined as a tree with zero-reduced cost edges and that at each iteration, we modify π values of all nodes being not Thus T. Thus w will be added in T as soon as we will have modified $\pi(j)$ by $\mu(w)$.

This leads to the following interesting result

Lemma 10 *Let* $minA = \min\{\mu(a) + \pi(a) \mid a \in V_1\}$ *and* $minB = \min\{\mu(b) \mid b \in V_2\}$. *Then* $\Delta = \min(minA, minB)$ *and the total change in the dual value for a node* w *is equal to* $\max(0, \Delta - \mu(w))$. *Define* z *as a node such that* $\mu(z) = \min(minA, minB)$. *Then let* p *be a path of length* $\mu(z)$ *from* v_i *to* z. *Then for all edges* (i, j) *of* p, $\bar{c}_{ij} = 0$.

Proof We have $\Delta \leq minB$ from the preceding lemma. $\Delta \leq minA$ as when we will have modified π values of an amount of Δ, $pi(a) = 0$ and the inner loop would end. Δ cannot be strictly smaller than $minA$ or $minB$ because this would tell us we have a node a or b defining $minA$ or $minB$.
As we only modify π values of the nodes in T, and that a node w is added into T only if $\Delta \geq \mu(w)$, we have that if $\Delta > \mu(w)$, then we haven't changed the dual value $\pi(w)$. On the other hand, if w were added into T say at the iteration i, then its dual value will have changed by a value of $\Delta - \Delta_i$. From the preceding lemma, we have $\Delta_i = \mu(w)$.
$\bar{c}_{ij} = 0$ for all edges in the path as they belong to T too, and T contains only zero-reduced cost edges.

This allows us to modify the inner loop (7)-(15). We compute a shortest-alternating-paths tree from v_i using a variant of Dijkstra's algorithm. As the nodes in the graph are reached in increasing order by this algorithm, we can stop it as soon as we are sure to have found a node defining $minA$ or $minB$. Let $v_1, v_2, \ldots, v_k$ be the order in which nodes from V_1 are reached by Disjkstra's algorithm. Recording dual values are positive $\pi(v) \geq 0$ we have

(1) If
$$minA_i = \min\{\mu(v_i) + \pi(v_i) \mid i < k, v_i \in V_1\} \leq \mu(v_k)$$
and no v_j with $j < k$ is a non-critical node of V_2, then $\Delta = minA_i$.

(2) If
$$\min\{\mu(v_i) + \pi(v_i) \mid i < k, v_i \in V_1\} \geq \mu(v_k) = minB_i$$
and v_k is a non-critical node of V_2 then $\Delta = minB_i$.

This follow from lemma (9) and (10). Now if k is minimal such that either (1) or (2) hold, then $\Delta \leq \mu(v_j)$ for all nodes $j > k$ and thus their dual values π are not changed.

In the following section we show how to decompose the initial set covering problem into a 2SC problem in such a way that the underlying graph G is bipartite.

6.3 Construction of the underlying graph

In order to compute this new lower bound on SC^*, we must build a bipartite graph G as explained in the previous section. Several graphs G are possible for a given instance of SC. For example a heuristic is proposed in [EDM92]. Let $H_j = \{e \in \mathcal{U} \mid e \in S_j \text{ and } q_j = \lfloor |H_j| \rfloor\}$. In order to relax SC to a 2SC, they create a bipartition (R', R'') of the set $\mathcal{U}$: $\mathcal{U} = R' \cup R'',\ R' \cap R'' = \emptyset$. For a given element i, if $|\{j : i \in S_j\}| < q_j$, then $i \in R'$. Otherwise $i \in R''$. They build the bipartition by looping on all the elements in $\mathcal{U}$ and by putting elements in R' or in R'' accordingly. They add two different dummy elements to R' and R''. Next they decompose each set S_i into several 2-sets S_i^j such that they contain exactly one element in R' and one element in R'' and such that (6.1)-(6.2) hold $\forall i : 1 \leq i \leq n$. In this case the graph G will be bipartite and 2SC can be solved by finding a minimum weight edge covering on this bipartite graph.

6.4 Improving *2SC* by Lagrangean relaxation

In this section we will enhance the lower bound of SC^* obtained from the relaxation 2SC. The enhancement $2SCLag^*$ of $2SC^*$ leads to a lower bound equivalent to the linear relaxation:

$$2SC^* \leq SCLag^* = 2SCL^* \leq SC^* \tag{6.8}$$

Our objective is to compute $SCLag^* = 2SCL^*$ by mean of the algorithm presented in the previous section that take advantage of the structure of 2SC.

We can use Lagrangean relaxation (see Section 3.6) in order to compute tighter lower bound of SC^*. The two problems SC and 2SC can be made equivalent by adding the following constraints to 2SC

$$S_i^k \in SC \iff S_i^l \in SC,\ \forall 1 \leq i \leq n, \forall S_i^k, S_i^l \in E(S_i) \tag{6.9}$$

2SC with the additional constraint (6.9) can also be formulated as the following optimisation problem (equivalent to the optimisation SC)

$$
\begin{aligned}
v &= min \sum_{1 \le i \le n} \sum_{j \mid S_i^j \in E(S_i)} y_i^j c_i^j && (6.10)\\
By &\geq 1 && (6.11)\\
y_i^j &= y_i^{j+1} && (6.12)\\
& \quad 1 \le j \le |E(S_i)| - 1, \forall 1 \le i \le n \\
y &\in \{0,1\} && (6.13)\\
& \quad 1 \le j \le |E(S_i)| - 1, \forall 1 \le i \le n
\end{aligned}
$$

where y_i^j tells whether the set S_i^j belongs to the solution and B is the incidence matrix of all 2-sets (there are exactly two 1's in each column). Constraint (6.11) expresses we want to find an edge covering of G and constraints (6.12) are equivalent to (6.9), i.e. for a given subset S_i, either we take all subsets S_i^j from its decomposition or we don't take any of them.

To approximate v, the constraints (6.12) are dualized to obtain a Lagrangean relaxation of this problem [LW79, EDM92]. Define λ_i^j as the Lagrangean multiplier of constraint $y_i^j = y_i^{j+1}$. Let $\lambda_i = \{\lambda_i^1, \ldots, \lambda_i^{|E(S_i)|}\}$ and $\lambda = \{\lambda_1, \ldots, \lambda_n\}$, the Lagrangean multipliers of constraints (6.12). We define $\lambda_{i0} = 0, (1 \le i \le n)$. Then the Lagrangean subproblem is given by

$$
2SC(\lambda) = min \sum_{1 \le i \le n} \sum_{1 \le j \le |E(S_i)|} y_i^j d_i^j \qquad (6.14)
$$

subject to (6.11) and (6.13), where $d_i^j = c_i^j + \lambda_i^j - \lambda_i^{j-1}$. $2SC(\lambda)$ is a minimum weighted edge covering of a bipartite graph problem; it has exactly the same structure as 2SC, only the weights of the edges are modified. It is interesting to consider how the Lagrangean multipliers λ_i^j can be interpreted. If we consider $\lambda_i^j, (1 \le j \le |E(S_i)| - 1)$ we observe it is present with a coefficient $+1$ in d_i^j and with a coefficient -1 in d_i^{j+1}. So any value of λ_i^j doesn't change the sum $d_i^j y_i^j + d_i^{j+1} y_i^{j+1}$. Thus, for any assignment of λ_i^j, $\sum_{j=1}^{|E(S_i)|} d_i^j = \sum_{j=1}^{|E(S_i)|} c_i^j = c_i$. By definition of weights c_i^j (see equation (6.3)), the Lagrangean multipliers λ_i can be interpreted as a repartition of the weight c_i among all subsets S_i^j of the decomposition of S_i. It shows it is not necessary to design two heuristics for computing weight c_i^j and Lagrangean multipliers λ_i. Any assignment for c_i^j such that $\sum_{j=1}^{|E(S_i)|} c_i^j = c_i$ is correct and we only have to design a good heuristic to compute initial Lagrangean multipliers and to modify them. Because there is no restriction in sign for λ_i^j, the new weights d_i^j could be negative.

As $2SC(\lambda)$ is a relaxation of SC for all λ (i.e. $2SC(\lambda) \leq SC^*, \forall\lambda$), we are interested in finding a λ such that $2SC(\lambda)$ is maximum, i.e. solving

$$SCLag^* = \max_{\lambda} 2SC(\lambda) \tag{6.15}$$

We can use the subgradient algorithm presented in Section 3.6 to solve this problem.

The Lagrangean relaxation provides a lower bound at least as tight as the linear relaxation. In our case, because the matrix B in the constraint (6.11) is totally unimodular, we have

$$SCL^* = \max_{\lambda} 2SC(l) \leq SC^* \tag{6.16}$$

Chapter 7

Handling of large-scale Set Covering Problems

In this chapter, we will focus on important problems that can be handled by a set covering formulation with a huge collection X of subsets. The vehicle routing problem (VRP) will be discussed as a concrete example of such problems.

7.1 Needs and challenges

The general CP approach presented in Section 5.1 requires the knowledge of the collection X, containing all the subsets. This collection needs to be explicitly specified. However in many real-life problems, this collection cannot be known explicitly. This usually happens when the collection is defined as the set of solutions of a combinatorial problem. For instance in the vehicle routing problem, X should contain all routes starting and ending at the depot and respecting some constraints. The number of such routes is exponential, and thus we cannot list all subsets to use the SC constraint as presented before.

The solution presented in this section is to allow the user to define intentionally the collection X by mean of a SCP. All solutions to the SCP will be considered as a member of X. However we still need to compute a tight lower bound as in Section 5.1 while we don't have all columns at hand. A technique generally used to solve an optimization problem (either to optimality or not) when we do not know all columns is the column generation presented in Section 3.7. We will present in this chapter how we integrate column

generation in the SC constraint. Besides describing a general framework for such integration, we want to highlight some parts that can be efficiently and generally implemented, i.e. independently from the problem to solve.

Section 7.2 formulates a well-known problem that require column generation to be solved. Section 7.3 overviews what can be found in CP litterature to solve very large-scale problems such as the VRP or the Crew Assignement Problem. Section 7.4 presents the general framework to integrate column generation techniques in the SC constraint and lists some design options found in the litterature. Section 7.5 explains our choices we made in the design of the column generation process embedded in the SC constraint.

7.2 Formulations of a practical problem (VRP)

The efficiency of the set covering formulation of the VRP is described in [BL94] where it is proven that the relative gap between optimal fractional and integral solutions becomes arbitrarly small as the number of customers increases.

7.3 Existing approaches to solve large-scale problems with Constraint Programming

This section introduces some work that we found interesting in the litterature that aims at solving large-scale problems with Constraint Programming. We don't want to present an exhaustive list of all papers we have found. We will focus on the ones we thought to be the most useful to design a global constraint for SC. These general ideas will be used to integrate Column Generation and Constraint Programming in Section 7.4.

7.3.1 Column Generation and CP

A general framework for including constraint programming in a column generation framework is presented in [JKK+]. This framework aims at solving the Pricing Problems arising in the column generation process by mean of constraint programming. No factorization is presented and many idea were already presented in other papers.

This approach is used in [FJK+02] to solve the crew assignment problem. The Pricing Problem is solved via constraint programming by the use of an efficient Path constraint.

In [SZSF02], they present an integration of this framework and a heuristic tree-search CP approach, emphasizing advantages and drawbacks of both methods. The Crew Assignment Problem is considered. They modify the CAP formulation by relaxing the set-partitionning constraints with set-covering constraints (i.e. replacing = by $\geq$). Then they use their CP based column generation framework (as presented in Section 7.3.1 to solve this relaxation and use a search-tree scheme with constraint programming essentially to build a feasible solution (i.e respecting the set-partitionning constraints) from a solution to the set-covering relaxation.

7.3.2 Local Search and Constraint Programming

A local search approach is presented in [Sha98] in order to solve the VRP. The neighborhood considered are very large (see [AÖEOP02] for a survey on such neighborhoods), being impossible to explore it explicitly to determine which neighbour is best. Here, a constraint-based tree search is used to find good neighbours.

The moves are defined as removal and re-insertion of customer visits. The algorithm selects several visits for removal at once and re-inserts them all together. This allows to define the large neighborhoud. A key point to this algorithm is that it chooses the visits for removal so that they are *related*. In order to define this *relatedness*, it uses a function $\mathcal{R} : M \times M \rightarrow \mathbb{R}$, where M is the set of locations we must serve. This function usually is defined by mean of the distance between visits and the fact that visits are served by the same vehicle. The function proposed in [Sha98] is

$$\mathcal{R}(i,j) = \frac{1}{c_{ij} + V_{ij}}$$

where c_{ij} is the cost to go from location i to location j and $V_{ij} = 1$ if i and j are served by the same vehicle, $V_{ij} = 0$ otherwise. c_{ij} are assumed to be in the range $[0; 1]$.

In order to control the size of the neighboorhoud, the algorithms first try to find $r = 1$ visit to remove and to re-insert it. Then if a moves didn't yield an improvement in the solution, it increases r by one.

Re-insertion of the selected visits is done by solving a CSP by branch-and-bound. This allows to prune the set of vehicles that could serve each visits, and thus to construct feasible subsets quickly. In this CSP, the variable V are the visits to be re-inserted and the domains are the set of routes r that could serve a visit. The most constrained variable rule is used to select which variable V (visit) to assign in the search procedure. The value selection rule selects the route r such that the increase in overall cost if route r serves visit V is minimum. Limited Discrepancy Search (LDS) is used in order to quickly explore a large part of the search-tree.

7.4 Integration of column generation in the SC global constraint

We will explain in this section how we can integrate column generation techniques in the CP approach presented in Section 5.1. This section will point out all parts that need to be implemented in order to clearly present the work that can be factorized in the SC constraint and part that is under the responsability of the user.

The general workflow for solving large-scale problem with CP is depicted in Algorithm 7.1. It is an extension of the workflow depicted in Algorithm 5.1.

In the following, we will explain each step in Algorithm 7.1 and list different ways of implementation to solve problems whose SC is a generalization.

7.4.1 Generating the initial set of columns

In order to use the column generation, we need to price feasible columns by using a dual solution. So we need to define an initial master problem, by generating initial columns. The primary goal of this initial set of columns is that there exists a feasible solution for the Master Problem in order to obtain valid dual values. Note that each column in this set needs not to be solution of the Pricing Problem.

In [FJK+02], the Master Problem is a set-partitionning with column generation. Because finding a feasible solution to SPP is NP-Hard, they add dummy columns in order to ensure that a feasible solution always exists.

However, because we price columns using dual variables, a good initial set

FilterSC()

1: $C = \emptyset; R = \emptyset$
2: $H = getInitialColumns(\mathcal{D}(T))$ // Building initial set of columns
3: **while** $H \neq \emptyset \wedge \neg stopping\_criterion$ **do**
4: $C \leftarrow C \setminus R \cup H$ // Adding good columns and removing bad ones
5: $SCRel = buildSCRel(C)$ // Building the relaxation
6: $(SCRel^*, \lambda) = solve(SCRel)$ // Solving the relaxation
7: $(lb, H) = good\_columns(\lambda)$ // Finding good columns with a heuristic
8: $R = bad\_columns(\lambda)$ // Choose columns to remove
9: **if** $H \setminus C = \emptyset$ **then**
10: $(lb, H) = optimal(\lambda)$ // If the heuristic fails in finding good columns, solve the subproblem optimally
11: **end if**
12: **if** $lb > \overline{\mathcal{D}(N)}$ **then**
13: **return** $\{(N, \mathcal{D}(N))\}$ // if lb is greater than the upper bound of N, we prune all the domain of N
14: **end if**
15: **end while**
16: **return** $\{(N, [\ \underline{\mathcal{D}(N)}; SCRel^*[\ )\}$

Algorithm 7.1: Filtering algorithm of the SC constraint with column generation

of columns could be very important. The ideal situation is when the problem of finding a feasible solution is not difficult. In this case we can use a primal heuristic to find several good columns.

A tree-search constraint programming approach is used in [SZSF02] to find initial feasible solutions. No optimization is done at this stage, the main goal being to generate a medium-size set of columns from which we can find a solution to the Master Problem to derive good dual values for the pricing step.

Depending on the problem local search could be a good choice to generate good primal solutions.

Attention should be paid not to generate too similar columns. These should be diversified in order to obtain *fair* first solutions. Local search can be of a great help to generate columns satisfying the diversity condition in a

small amount of time.

7.4.2 Solving current MP

This step tries to solve the current master problem to obtain a dual solution that will provide us interesting information to generate the following set of columns by pricing them.

Theorically, we need the optimal dual solution in order to assert that the current set of columns is optimal, i.e. that all columns not in the current MP with strictly positive cost are not part of the optimal solution. The best solution to solve the Master Problem optimally is to solve it by the simplex algorithm. It can incrementally find the optimal solution when we add or remove a few columns.

However computing the optimal dual solution can be time-consuming and a nearly optimal solution could be found quickly. A useful design is not to solve the current Master Problem optimaly at each iteration of the loop (3)-(15).

In [GSMD99], they use a stopping criterion for the LP solver: they stop the solver when it does not improve the solution enough in a given number of iteration steps. This allows to spend less time solving LP problems to optimality and allows to branch faster and thus to obtain solutions in a smaller amount of time.

Lagrangean relaxation of the Master Problem could also provide us with near-optimal dual values (also called Lagrangean multipliers). This technique is efficiently applied to the set covering problem in [CFT99]. They present an improved step-size and step direction for the subgradient algorithm that leads to a faster convergence.

7.4.3 Removing Bad Columns

With the current solution, we could use a heuristic to determine whether some columns in the current Master Problem has very small probability to be part of an optimal solution. Such columns could be removed from the Master Problem to allow faster optimization.

The main idea behind column generation is to price columns. One chooses columns with highly negative cost to enter the master problem. [BJN$^+$98]

proposes to remove from the Master Problem the columns with a highly positive reduced cost.

Depending on the structure of the problem, we could use another heuristic to remove such columns.

7.4.4 Adding Valid Columns to the Restricted Master Problem

Once we have a primal and a dual solution, we need to create a subset of columns to add in the Master Problem. These columns should be created such that we would strongly believe they are part of an optimal solution. Some heuristic could provides such subsets of columns very fast.

If no column with negative reduced cost can be found using fast heuristic algorithms, we need to solve the Pricing Problem to optimally.

Heuristics to create valid columns

Two heuristics for VRP are presented in [SS]: one is a construction algorithm and the second is an improvement algorithm.

The first heuristic initializes a route being empty then add repeatedly requests with negative reduced costs until no more such request exists.

The second heuristic selects columns with zero-reduced cost (at least all columns in the current solution have such reduced costs), and modify them with local-improvements algorithm to obtain hopefully better columns.

An important point in this step is the diversity of generated columns. It is useless to add many similar columns. However adding very different columns can lead to very good dual information. A well-used strategy in practice is to impose that an element cannot appear in more than K different generated columns, for a fixed constant K. In the case of the crew assignment problem, the number of times that a crew-pairing assignment appears can be bounded in the same manner (see [FJK$^+$02]).

In [SS], they also work with approximate insertion cost, instead of true reduced costs for columns. This allows to use their heuristics without solving MP.

Optimizing the CG subproblem

If heuristics cannot find improving columns, we need to solve the Column Generation subproblem optimally. Solving th subproblem is problem dependent.

For a lot of problems (such as vehicle routing and crew assignment), the subproblem is a constrained short est path in a acyclic graph problem. A dynamic program exists to solve this problem. When there are a lot of side constraints in the subproblem, sometimes constraint programming is used to solve it using an efficient Shortest Path Constraint [FJK+02, SZSF02].

We can speed up the optimization by reducing the size of the pricing problem. For instance when we need to solve Constrained Shortest Path Problems, we could remove arcs that will unlikely be part of the optimal solution. Then some solutions will be found faster (see [DY91]).

7.5 Our integration of Column Generation and Constraint Programming

We present and justify our own design choices for the framework presented in Algorithm 7.1. These choices are valid for the SC constraints and are not problem-specific as was the case in Section 7.4. Some ideas presented here are new and are generalizations of ideas presented in Section 7.4.

7.5.1 Choosing initial columns

In the case of the SC constraint, the Master Problem is a pure Set Covering Problem. We should generate an initial set of columns at each node of the search tree, or more generally at each call to the propagator. However from one node to another, we can expect that the problem didn't change so much so that we can keep the last set of columns used. In order to ensure that there exists a solution to the Master Problem, we add a dummy column covering all elements with a very high cost.

Although it is not strictly necessary, it should be expected that a very good initial set of columns at the root node would increase the computation time. The experimental section 10.2 will point out the usefulness of using a heuristic to generate columns at the root node.

7.5.2 Solving Current MP

In order to obtain dual values to price columns we could use any relaxation presented in Section 5.4.1 and in Section 6 that provides them (i.e. SCL^*, $SCLag$ and $2SC$). However only SCL^* ensures to get the optimal solution.

We use SCL^* in order to solve the MP and to find dual values. We implemented a stopping criterion for the Simplex algorithm in order not to spend too much time optimizing dual values in the first step of the column generation process. As in [GSMD99], the simplex algorithm is stopped if it does not get a substancially better solution after n iterations.

7.5.3 Choosing which columns to add to the pool

Two approaches can be used to generate new columns: heuristics or complete algorithm. All generated columns should respect a set of constraints $\mathcal{C}$ specified by the user. Being in a constraint programming framework, we use a CSP to generate these columns. We post all constraints $c \in \mathcal{C}$. The heuristic and complete algorithms differ from the branching scheme used. The general algorithm (either complete or heuristic) is presented in Algorithm 7.2. Two procedures must be defined: $createCSP$ and $branch$. $createCSP$ is problem dependent and must be defined by the user. $branch$ is a search heuristic as defined in Section 2.3.2. The aim of this section is to present several solutions for designing the $branch$ procedure.

PRE: - x^* the optimal LP solution of the current Master Problem
- $createSCP$ is a function defined by the user that returns a CSP and a set variable S. The value of S in all solutions of the CSP are feasible columns of the Pricing Problem
- $branch$ is a Branching scheme as defined in Section 2.3.2
- $\mathcal{E}$ the set of elements to cover

1: $(CSP, S) \leftarrow createCSP()$
2: $CSP \leftarrow CSP + \{Cost(S) < 0\}$
3: $Pool \leftarrow search(CSP, branch)$
4: $\mathcal{T}$:= select some columns from $Pool$

Algorithm 7.2: Algorithm to generate new columns with reduced cost

Heuristic

For the heuristic we use an incomplete search-tree traversal in order to quickly obtain columns with negative cost. We took ideas from local search algorithms, because they revealed to be highly efficient though very fast. The idea is similar as the ones used in the two improvement algorithms presented in [SS]; we try to generate new subsets with negative reduced costs from the subsets being part of the optimal solution of the Master Problem. Because we don't want to solve a LP problem to compute the real reduced cost of the newly created subsets, we define the insertion and deletion costs [SS] of an element e in subset S_i as

$$I(e, S) = Cost(S + e) - Cost(S) + \pi_e$$

and

$$D(e, S) = Cost(S - e) - Cost(S) - \pi_e$$

They represent an approximation of the reduced cost of a subset after insertion or deletion of an element. An abstract body for the heuristic is presented in Algorithm 7.4. The main idea is as follows:

- choose a collection $\mathcal{S}$ of already built subsets; line:7
- remove some elements from some subsets from $\mathcal{S}$; line:8-line:11
- re-insert these elements into subsets of $\mathcal{S}$; line:14-line:19

At the end of the heuristic, we will have a collection $\mathcal{S}$ of subsets with some of them being new. These can be added in the Master Problem and the column generation process can go on. Algorithm 7.4 is a generalization of the improvement algorithm presented in [SS] and of the local search algorithm from [Sha98], even though these two last algorithms are very different.

We need to specify the following in order to completely design the heuristic algorithm:

- When we consider the algorithm generated enough columns
- What is the collection of already built columns which we will create other columns from
- Which element from which subset must be removed

We choose to stop the search when either there is no subset with cost less than C_{min} in the last N generated subsets, or when we have generated T subsets.

As in [SS], we decided to generate subsets from subsets having zero-reduced cost wrt the current LP solution of the Master Problem.

removeElements(Subsets Ss, int toRemove, int D)

1: $\mathcal{E} = \bigcup_{S \in Ss} S$
2: $e := chooseRandomElement(\mathcal{E})$
3: $\mathcal{E} := \mathcal{E} \setminus \{e\}$
4: $\mathcal{R} = \{e\}$
5: remove e from all subsets in Ss
6: **while** $|\mathcal{R}| < toRemove$ **do**
7: $e := chooseRandomElement(\mathcal{R})$
8: $lst := rankUsingRelatedness(e, Ss)$ // Sort element with respect to relatedness
9: $rand :=$ a random number in $[0; 1[$ // Choose the r^Dst element in lst
10: $e := lst[Integer(|lst| * r^D)]$
11: $\mathcal{R} := \mathcal{R} + e$
12: remove e from all subsets in Ss
13: **end while**

Algorithm 7.3: Algorithm to select some elements to remove in the local search algortihm

In order to selects how to remove elements, we use the local search algorithm presented in [Sha98] rewritten in Algorithm 7.3. However we must define another *relatedness* function because the one they give is specific to the VRP. We use a idea similar to the simulated annealing to discover it. Let $\mathcal{P}$ the current column pool

$$K(e, f) = \{i \in \mathcal{P} \mid e, f \in S_i\}$$

$$\kappa(e, f) = \frac{\sum_{i \in K(e,f)} Cost(S_i)}{|K(e, f)|}$$

$$\kappa_{tot} = \frac{\sum_{S_i \in \mathcal{P}} Cost(S_i)}{|\mathcal{P}|}$$

We use the following relatedness function

$$\mathcal{R}(e, f) = \frac{(\alpha t)\kappa(e, f) + (1 - \alpha t)\kappa_{tot}}{\kappa(e, f) + \kappa_{tot}} \tag{7.1}$$

where $\alpha = \frac{1}{T}$ and t is equal to the number of solutions already found (T is defined above). We observe that $\kappa(e, f)$is in the range $[0; 1]$. In the beginning of the heuristic, $\kappa(e, f)$ does not depend on e and f. This allows to remove elements in a random fashion. After a while, we will use the current column pool to determine whether a lot of good subsets in the pool does contain both elements. In this case, this would show that e and f are *related*. We don't need any information on the structure of the problem to compute this value. This makes it a useful definition.

branchLS()

PRE: - S is the set-variable defining the subset we are currently building

1: $\mathcal{R} := \emptyset$
2: **if** enough changes **then**
3: **return** $\emptyset$
4: **end if**
5: **if** $\mathcal{E}^+ = \emptyset$ **then**
6: $Pool := Pool + S$
7: $\mathcal{S} := chooseColumns(Pool)$
8: **while** not enough removed elements **do**
9: $(s, e) := chooseSubsetandElement(\mathcal{C})$
10: $\mathcal{S} := \mathcal{S} - s + (s - e)$
11: **end while**
12: define $\mathcal{E}^+$ // define the elements to add
13: **end if**
14: **if** $\exists e \in \mathcal{E}^+, s \in \mathcal{S} \mid I(e, s) = \max_{e' \in \mathcal{E}^+} \min_{s' \in \mathcal{C}} I(e', s')$ **then**
15: $Ds := \{D \oplus (e \rightarrow \{s\}), D \oplus (e \rightarrow D(e) - s)\}$
16: **return** Ds
17: **else**
18: **return** $\emptyset$
19: **end if**

Algorithm 7.4: Local Search algorithm to compute new columns

Complete Algorithm

The complete algorithm solve the same CSP by a branch-and-bound traversal of the search-tree. The main idea in the branching scheme is to add the element with the smallest dual values, doing it repeatedly until no more such

elements exists. The simplest version of this branching scheme would be to begin with an empty subset at the root node. An improved version would be to begin with a subset already containing a few elements at the root node. This allows to build not too similar columns. The complete search algorithm is given in Algorithm 7.5.

branchComplete()

PRE: - x^* an optimal LP solution
POST: - Add an element in S

1: $S \leftarrow \emptyset$
2: **while** $\exists e \in \overline{S} \mid I(e,S) < 0$ **do**
3: Let $e \in \overline{S} \mid I(e,S)$ is minimum
4: $S \leftarrow S + e$
5: propagate
6: **end while**

Algorithm 7.5: Complete branching scheme to generate new columns

7.5.4 Choosing which columns to remove

As in most column generation scheme we chose to remove from the current pool all columns with a reduced cost $rc > D^{<}_{pool}$.

7.5.5 Pool Management

We describe here how columns are added and removed from the Master Problem. As can be observed in Algorithm 7.2, the heuristic and complete pricing algorithms add columns to the pool $\mathcal{P}$. We still need to specify which of the columns in $\mathcal{P}$ are effectively added to the Master Problem.

In order to add not too similar subsets into the Master Problem, we greedily select subsets from the pool such that no elements are covered by more than three added subset.

The subsets having a reduced cost $rc > D^{<}_{MP}$ are removed from the Master Problem, but not from the pool.

Chapter 8

Search Heuristics for Set Covering Problems

The main goal of this work is the design of a global constraint for Set Covering Problems. However, using internal information of most constraints presented in Section 5.4.1, we can design powerful search heuristics in order to quickly find good solutions to the constrained Set Covering Problems.

Search heuristics are mandatory because filtering algorithms implemented in propagators are not sufficient to prune the search space and to find solutions. When all propagators are idle (i.e. they cannot do any more pruning), we need to branch in order to explore the search tree and to build a solution, for instance cutting the search tree in two, with one branch $i \in T$ and the second with $i \notin T$.

This section aims at presenting such heuristics.

8.1 Search Heuristics in the Litterature

When we use the linear relaxation of the Set Covering Problem in the SC constraint, one solution is to choose a fractional component of the last computed solution x^* and to assign it to 0 or to 1 (variable fixing). However this strategy does not preserve the structure of the Pricing Problem; down in the search tree several solutions will be forbidden when we fix $x_i = 0$ and the Pricing Problem will be highly complicated.

An improved branching strategy consists in fixing several fractional values at once. For instance in [GSMD99], they fix at 1 all variables with a

value greater than 0.6 in the optimal solution of the linear relaxation. This branching strategy seems to work when we want a fast solution, exploring branches with $x_i = 1$ instead of branches with $x_i = 0$.

There exists another solution when the Master Problem has a set partitionning structure, there exists an efficient branching rule [FR76]. It is based on the observation that for two given elements e and f, in a solution to MP, either they belong to the same subset or they don't. So we can choose two non-integral variables λ_e and λ_f and impose that $e \in S_i \iff f \in S_i$ or $e \notin S_i \iff f \in S_i$.

Otherwise, the most efficient branching schemes take care of the structure of the problem.

8.1.1 Search Heuristics for the VRP

For the VRP, branching schemes can be roughly subdivided into two categories: branchings based on routing decisions and the ones based on assignment decisions. The firsts modify the way one vehicle serves different requests. The seconds modifies the way requests are handled by which vehicles.

A routing branching scheme is presented in [DY91]. Let x_r be a fractional component of the solution of the LP Master Problem and let $i_1, \ldots, i_n$ the requests served by route r with request i served right before request $i+1$. Let $O_{ij} = 1$ if request j is served right after request i and $O_{ij} = 0$ otherwise. We can cut the search tree into $n+2$ branches. The i^{th} branch would correspond to $O_{01} = O_{12} = \ldots = O_{i-1\ i} = 1$ and $O_{i\ i+1} = 0$.

A branching strategy based on assignment decision is presented in [DY91]. It is based on the following proposition.

Proposition 11 *Let x^* be the optimal solution of the LP relaxation of the VRP. Define Ω_k to be the set of routes served by vehicle k and $\delta_i^r = 1$ if and only if route r serves request i. Define also $z_r^i = \sum_{r\Omega_k} \delta_i^k x_r^k$. Then x^* is integral if and only if z is integral.*

Proof see [SS].

The branching strategy selects one variable such that $0 < x_r^k < 1$ and a request i such that z_k^i is not integral. Two branches are created: one with $z_k^i = 0$ and one with $z_k^i = 1$.

Another branching scheme for the CAP [FJK+02] build a route by adding requests that contributes the lowest value to the path cost.

Iterative search strategy are proposed in [Sha98, BFS+00, TLZO01].

Part IV

Experimental framework

Chapter 9

A Gecode implementation

In order to validate our different ideas, we implemented them in the GEneric COnstraint DEvelopment environment [gec]. We chose this framework because it seems to be the most efficient CP solver that is freely distributed, making it easy to extend.

In this chapter we will briefly describe how we implemented global constraints presented in Section 5.4.1 and in Section 6 and how we implemented the column generation process described in Section 7.5.

9.1 Implementation of the global constraints

Global constraints as $2SC$, SCL and MD were implemented as propagators in Gecode. The aim of a propagator is to consider a small subset of all variables in a CSP and to prune their domain by removing elements that are not part of any solution. All the three propagators $2SC$, SCL and MD have the same structure, they consider an Finitie Domain variable and a Set variable. The former one represents the cost of a cover, and the second one contains all indices of the subsets being part of this cover. As explained in Section 5.1, these propagators compute a lower bound $lb_{SC} \leq SC^*$ and use it to prune the domain on N. A shaving process is used to prune T.

9.1.1 Common Structure of the Propagators

This main structure is implemented in the file SCRel.cpp (see appendices). It has the structure of a Gecode propagator. The philosophy of Gecode is

to copy everything when one creates a computational space. This can be too time-consuming if the propagators contain a lot of information. This is the case for us; we need to store a lot of information in order to solve the $2SC$ or SCL^* lower bounds efficiently. Thus we decided to create a class representing a relaxation of SC that store all this information. Instead of being copied, it is updated when needed to reflect changes in spaces. This class can be accessed through a pointer (`relax`).

When we need to propagate, we need to achieve the following steps: update of the data structures (`update`), compute a lower bound on N (`computeLB`) and prune the domain of T by shaving (`shave`). Algorithms implemented in SCRel are the ones described in Section 5.1. In order to be used with the propagator SCRel, the different relaxations must provide

`build()`: A function that is called at the creation of the propagator that builds up the relaxation (initialization of data structures)

`makesure` **(resp.** `makeremoved`**,** `makeunknown`**) :** functions that update the data structures to reflect that a subset must (resp. cannot, may) be part of the cover.

`computeLB()`: A function that compute the lower bound lb_{SC} and returns a negative result if the relaxation of SC has no solution (i.e. SC would not have any solution too).

`getOptimal()`: returns the lower bound computed by `computeLB`().

In the following sections, we present the implementation of three relaxation of SC: $2SC$, SCL^* and MD. We describe them in terms of the functions enumerated above.

9.1.2 Implementation of *2SC*

We describe some implementation details of the algorithm Algorithm 6.1.

Data Structures

The bipartite graph is stored in a special data-structure in order to allow fast incremental changes in its structure and efficient algorithmic design.

No special data structures are used to store information about the nodes of the graph. One object `edge` represents an edge and contains all the information we may need about it. In particular it stores its cost, its endnodes and some values needed to compute the subgradient algorithm for Lagrangean relaxation. For each nodes i, there exist four linked lists containing edges incident to i. The first contains all free edges (edge being not part of the cover), the second contains edges that are in the cover. The third and fourth linked list contain the edges that must be part of the cover and the ones that cannot be part of it respectively. We store these two last kinds of edges instead of erasing them because as told in Section 9.1.1 we use only one data structure for the entire search tree. Thus the edges that are in $SURE$ or $REMOVED$ could be reinserted as unknown when we explore other branches of the tree.

The four linked lists per node are encoded directly in `edge`: each edge contains a pointer to the previous and the next edge in the linked list. In this way, once the algorithm has an edge *in hand* it can remove it and re-insert it in the different linked list in $\mathcal{O}(1)$. This allows to efficiently augment the current cover with a given admissible path.

The main operation in Algorithm 6.1 that cannot be achieved in optimal time-complexity is the retrieval of the edge incident to a given node i of minimum cost. We tried to store edges in both linked lists and priority queues, that should allow a logarithmic-time retrieval of the edge of minimum cost. However from our experience the additional cost needed to maintain the priority queue is not worth its benefits.

`build()`

This function initializes the data structures used by the algorithm. There are two main objectives here. First we need to build the graph in itself. Second we need to build an initial edge cover in order to make the algorithm invariant (i)-(v) holding (see Section 6.1). The first part is pretty easy. There are two options for the second objective: either we put all edges in the cover or we try to put as least edges as we can in the cover. For the first option, we can set $\pi(i) = \max_{(i,j)\in E} c_{ij}$ and the invariant holds. For the second option, for each node i, we put in the cover one edge of minimal cost and set $\pi(i)$ to its cost. We implemented this second option.

`makesure makeremoved, makeunknown`

The implementation is direct from the algorithm described in Section 6.2.

`computeLB()`

The implementation follows the algorithm and its improvements described in Section 6.2.

`getOptimal()`

We just can compute the following value and return it

$$\sum_{(i,j)\in R} c_{ij}$$

9.1.3 Implementation of *SCL*

In order to provide a lower bound here, we need to compute the linear relaxation of SC i.e.

$$SC^* = \min_{x\in\mathcal{B}^n} \sum_{1\leq j\leq n} c_j x_j \qquad (SC) \tag{9.1}$$

subject to

$$\sum_{1\leq j\leq n} a_{ij} x_j \geq 1 \quad \forall i \in \mathcal{U} \tag{9.2}$$

$$x_j \in \mathcal{B} \quad \forall j = 1\ldots, n \tag{9.3}$$

where $a_{ij} = 1$ iff $i \in S_j$. $\mathcal{B} = \{0, 1\}$.

As explained in Section 3.3, the simplex algorithm is very useful for solving efficiently linear problem while allowing incremental changes.

Many linear programming solver exists that provides libraries implementing the simplex algorithm. We chose to use lp_solve because it is freely distributed, while being robust and providing an easy API to use in C++ programs.

The following describes how our code use lp_solve in order to solve the linear relaxation of SC. There is nothing special, implementation of this relaxation is straightforwrd.

`build()`

lp_solve provides a class `lprec` for describing a linear problem. We use the API provided to encode the problem (9.1)-(9.3). Each column represents a subset, and each row represents an element to cover. Upper and lower bounds of the variables are set to 1 and 0 respectively.

`makesure makeremoved, makeunknown`

In order to put a subset S_j in the cover, we set the bounds on the variable x_j to 1. To disallow a subset S_j to be in the cover we set both bounds to 0. To make unknown we set the lower bound of x_j to 0 and its upper bound to 1.

`computeLB()`

We just call tha API and call the solver: `solve(lp)`.

`getOptimal()`

We return the optimal value of the linear problem found by lp_solve.

Haaa, if implementation could always be as easy...

9.1.4 Implementation of *MD*

`build()`

Let n be the number of subsets and m be the number of elements to cover and $N = \{1, \ldots, n\}$. As in Section 6.2, we denote the set of the indices of the subsets that *must* be in the cover by $SURE$ and the indices of the subsets that *must not* be in the cover by $REMOVED$. Remember that the intersection graph is the graph whose nodes are the elements. There is an edge (i, j) if and only if there is one subset that contains both i and j. We store the intersection graph G_X as an array g of m hashmaps. The keys of these maps are neighbours of a given node and values are the number of subsets that contain both elements:

$$g[i].find[j] = g[i][j] = |\{k \in N \setminus REMOVED | i \in S_k, j \in S_k\}|$$

We also store an array `modified` of m integers that remembers, for each element e, how many subsets that *must* be in the solution contains e:

$$covered[i] = |\{j \in Sure \mid i \in S_j\}|$$

We also store two other arrays of n boolean `oldglb` and `oldlub` that gives the greatest lower bound and the lowest upper bound of the variable T when the current data-structure were built. This allows to update the data-structure more efficiently: if an index j didn't move among $\overline{T}$, $\underline{T}$ or $\underline{\overline{T}}$ during two consecutive calls of the propagator, we don't need to consider it to update the data-structures.

`makesure makeremoved, makeunknown`

These functions are direct implementation that maintain the class invariant described in the above section.

`computeLB()`

In order to compute a independent set S of the intersection graph G_X, we store all nodes in a priority queues that store them in increasing degree. Until there are nodes in the priority queue, we take the node i with minimum degree, put it in S, remove all its neighbours from the priority queues and update the degree of all neighbours of the neighbours of i by mean of a `decreaseKey` function provided by the priority queue. This allows to efficiently find a independent set of the graph

`getOptimal()`

We just need to return the number of elements inserted in S.

9.2 Implementation of Column Generation

9.2.1 Main Structure of the Implementation

This section describes our implementation of the approach presented in Section 7.5 to handle very large-scale SC problems. The big picture of the main concepts is illustrated in Fig.9.1.

`SCRel` is presented above in Section 9.1.1. It uses `SCColGen` which contains the main definition of the new relaxation (e.g. how to compute the lower bound on SC, how update the internal structure). `SetsCollection` is a class that defines a collection of subsets. This collection is the collection X used in Section 5.1. The difference is that X has now a huge size, that makes it impossible to store all the subsets in memory. `LP_solve` is an external library used to solve the reduced problem by the simplex algorithm. It was already used in Section 9.1.3 to compute the linear relaxation of SC. `ColGen` is a class that aims at making the link between the columns in the LP formulation used by `LP_solve` and the entire collection. Because X can be very huge, all subsets generated are not automatically stored in LP in order to speed up the simplex algorithm. `ColGen` aims at enumerating all subsets generated and at providing the index of a given subset as well at retrieving a subset of a given index. `SubsetBuilder` is a user-class that define the collection X, i.e. which subsets belongs to X. `CSPSubset` and CSPHeuristic are two classes that define respectively the CSP of finding the subset of minimum reduced cost and the one used to find subsets of negative reduced costs by a heuristic. `BranchingComplete`, `BranchingHeuristic` and `BranchingClose` defines search strategies used in order to solve these two CSP's.

In the following each class is presented in more details.

9.2.2 Storing the Collection of Subsets

We saw in Section 5.1 that a cover R is defined as the set T of the indices of the subsets being part of the cover. This implies that there is a bijection between the indices and the subsets. Thus we must be able to retrive a subset form its index and vice-versa.

From an implementation point of view, subsets are stored in a `vector` data-structures (i.e. an array) in the order they are found by the column generation process. This allows to easily retrieve a subset from a given index. A hashmap is used in order to implement the function $2^{\mathcal{U}} \rightarrow \mathbb{Z}$ that retrieve the index of a given subset, or to determine that this subset has not been found yet. Two subsets are equal if they contain the same set of elements and their costs are equal. We use the following hash function

$$hash(S) = \sum_{e \in S} 2^{\ e \bmod 32}$$

9.2.3 Storing the Reduced Subproblem

As in section Section 9.1.3 we use the external library `LP_solve` to solve linear optimisation problem by the simplex algorithm. Because the collection X of subsets can be very huge, the LP problem that we solve with `LP_solve` does not contain all columns. `ColGen` aims at making the link between columns in the LP problem and subsets in the collection X. This is not a big class. `LP_solve` allows to give a name at each row and each column of the LP problem. Thus a column that will represents the subset S_i will be named "i". Now making the link between columns in LP and indices of subsets can easily be done by using functions provided by `LP_solve`.

9.2.4 Easy Specification of the Collection of Subsets

Recall that the main objective of the column generation approach is to offer the user a general framework to help him specifying very large-scale SC problems and to offer already implemented parts of this framework. However some parts are specific to the problem to solve and must be implemented by the user. Classes as `SubsetBuilder` contains all the unknowns of the problem and must be written by the user.

`SubsetBuilder` defines the collection X by a CSP. Indeed, as we saw before, X will be the collection of the subsets being solution of the CSP defined by the user. The user must then

1. define the variable of the CSP: functions `init` and `update`.
2. specify all the constraints on the variables of the CSP: functions `constrain`.
3. give some information about the structure of the CSP: `insCost` and `modifyElementWeight`.
4. Explain how to ensure that a set of elements is a solution to the CSP: `postLastBranching`

The following discusses all these functions more in depth. An example of such class is given in Section 9.3.

Defining the Variables of the CSP

All CSP's contains variables whose domains are constrained. These variables will be instance variables of the `SubsetBuilder` class.

The function `init()` creates and initialises all variables. It specifies the domains for each of them.

The function `update(SubsetBuilder)` allows to update the variables. This is needed from the architecture of the CP engine Gecode. It allows to clone variables when we copy computation spaces.

Constraining the variables

A unique function `constrain ( SetVar S, IntVar RC)` allows to post constraints on the variables of the CSP. Furthermore parameters `S` and `RC` represent the subset that will be added to the collection (when a solution to the CSP is found) and the reduced cost of this subset. Thus the user must constrain these two variables in order that `RC` is exactly the reduced cost of `S`. Passing these parameters allow the user to constrain (i.e. to define) these two values.

Structure of the Problem

In order to use our global constraint the user needs to specify additional information about the problem. First he needs to specify how we can compute the insertion cost defined in Section 7.5.3. Given a subset and an element, the function `insCost(S,e)` of the class SubsetBuilder should return $I(e, S)$.

Second, the framework will use the function `modifyElementWeight` to change the problem in order to reflect changes in dual values. Remember that the `RC` parameters of the `constrain` function must be the reduced cost of variable `S`. `modifyElementWeight` is used to pass dual values to the class `SubsetBuilder`.

Third the user should specify how to build a first feasible solution to the problem of finding a cover. The Column Generation process needs a feasible solution to obtain valid dual values. The user must then provide a way of computing one cover of $\mathcal{U}$ such that each subset in this cover belongs to the collection X.

Ensuring that we have a solution

Once we have determined the set variable `S`, we must verify that it is a valid subset for X. Imagine X is defined as the set of nodes of a graph G respecting some constraints. A graph is assigned once we have assigned the set of nodes

and the set of edges. Thus if we have assigned `S` to a given value, we are not sure that `S` is a valid subset, because there could be no set of edges E such that $G = (S, E)$ is a graph repspecting all the constraints. The user must then provide a Branching to assign all variables initialised in the `init` function. Once all variables are assigned, then we are sure that `S` is a valid subset and we can add it to the collection.

The Branching is specified in the function `postLastBranching`. As indicated by its name, this function posts Branchings that will be called after all others, allowing to assign all variables not assigned yet.

9.2.5 The Relaxation Class

At the upper level, the global constraint using column generation is implemented as all the other propagators presented in Section 9.1. `SCColGen` provides functions needed by `SCRel` in order to prune the domains of N and T. Its `build` function initialises the LP problem with columns created by the `buildFeasibleSolution` provided by the user and computes the first primal/dual optimal solution. The `makeSure`, `makeRemoved` and `makeUnknown` are similar to the one of the implementation of the SCL^* relaxation. We fix both the lower and the upper bounds of the variables.

The `getOptimum` function returns the lower bound if the precedent call to `computeLB` found one minimum reduced cost.

The `computeLB` function is a direct implementation of Algorithm 7.1. The heuristic and the optimal search for subsets with negative reduced cost are described below.

9.2.6 Optimal Solution for the Pricing SubProblem

The implementation for the search of the subset of minimum reduced cost is pretty straightforward. We wrote a class `CSPSubset` that define a CSP with a variable `S` and an integer variable `RC`. RC is constrained to be negative. This class also uses the function `constrain` provided to the user to post all constrain on `S` and `RC`. Then we use a Branch and Bound search engine (implemented in Gecode) in order to find the subset of minimum reduced cost.

The generic branching strategy implemented begins with the empty subset and add elements with negative insertion cost wrt the current subset until

no more such elements exists. This branching strategy must be complete because we cannot miss the subset with minimum reduced cost. Thus if there is an element with negative reduced cost, we divide the search tree in two. The first branch puts the element in the lower bound of `S`, and the second removes it from the upper bound of `S`. If the element has positive reduced cost, we remove it from the upper bound in the first branch and put it in the lower bound in the second.

9.2.7 Heuristic

Main structure

As explained in Section 7.5 we selects from the LP problem all subsets with zero-reduced costs. We select elements to remove with a direct implementation of Algorithm 7.3. Values $|K(e,f)|$, $\kappa(e,f)$ and κ_{tot}, needed for this algorithm, are stored as invariants in the class `SetsCollection`.

Let $\mathcal{E}$ be the set of removed elements and $S_1, S_2, \ldots, S_n$ be the selected subsets with zero-reduced cost. In order to compute the heuristic, we create a CSP with n set variables $Sv_1, \ldots, Sv_n$ and n associated finite domain variable $Cv_1, \ldots, Cv_n$ representing the cost of the subset Sv_i. For each variable we set an initial domain $\underline{Sv_i} = S_i \setminus \mathcal{E}$ and $\overline{Sv_i} = \mathcal{U}$. This is done in the class `CSPHeuristic`.

The branching strategy is a direct implementation of Algorithm 7.4; it tries to put elements from $\mathcal{E}$ in the different set variables Sv_i depending on the insertion costs. This is implemented in `BranchingHeuristic`.

`BranchingHeuristic` does not assign set variables; it only puts elements in the lower bound of the set variables but never remove elements from its upper bound. Because propagators may not be arc-consistent, we need to assign set variables in order to verify that the current greatest lower bounds of the set variables represent a solution. To meet this objective, we implemented a special branching, `BranchingClosing`. The most direct implementation to assign a set variable would be to divide the search tree into two until there is an element e that is in $\overline{Sv_i} \setminus \underline{Sv_i}$. In the first branch we have $e \notin Sv_i$ and in the second $e \in Sv_i$. However this is not the most efficient implementation because each time we divide the search tree, Gecode calls all propagators and perform some other tasks. Moreover, for most problems, we expect that we should not fail; if S is a solution then the probability is high that s is a solution too for $s \subseteq S$. Thus we implemented the following strategy in `BranchingClosing`. We divide the search tree in three branches. In the first

we assign Sv_i: $Sv_i \leftarrow \underline{Sv_i}$. If this branch fails then we select the element $e \in \overline{Sv_i}$ that has the minimum insertion cost and in the second we post $e \in Sv_i$ and in the third we post $e \notin Sv_i$. Usually the first branch does not fail. This allows to find a solution very quickly.

In order to constrain the problem of finding a valid subset, the user may have declared other variables. Still because some propagators may be not achieve arc-consistency, we must assign them. After `BranchingClosing`, we branch according to the branchings posted by the `postLastBranching` function provided by the user. When it stops, we know we have a solution that can be added to our set collection.

The solutions of the CSP described above can be explored in different ways. Either we stop exploring when we find the first solution, or we decide to explore the search tree by branch-and-bound. The first choice has the advantage of finding many different subsets quickly. However we could find subsets with better reduced cost with the second choice. When using branch-and-bound we add the constrain

$$Cv_i < Cv_i^{old}$$

for all $i \in \{1, \ldots, n\}$. Cv_i^{old} is the cost of the ith subset of the solution just found. The experimental sction 10.2 will xompare the efficiency of this heuristic with and without branch-and-bound.

Algorithmic improvements

In the previous paragraphs we described the main structure of our implementation of the heuristic. However some algorithmic improvements are needed in order to find subsets more quickly.

No useless branching First, we should observe that when we use a branch-and-bound scheme, we don't expect to find better solution when we insert more elements in a subset that is already a solution. Thus when we have found a solution, we don't want to explore branches defined by `BranchingClosing`. In order to do that, we added two instance variables: one integer `meDone` and one pointer `*done` to an integer that is shared by all instances of `BranchingClosing` (this pointer can be seen as a kind of static variable). We increase the value pointed by `done` by one every time we quit `BranchingClosing` (i.e. when `BranchingClosing` has nothing more to do). When a instance of `BranchingClosing` is created, we set `meDone = *done`.

In the commit function, when we need to post constraint to explore the right branch, we check whether `*done > meDone`. If true, this means that we found a solution in the left branch and instead of exploring the right branch, we fail.

Initialization overhead We observed that the search engine implemented by Gecode is slow during initialization. The reason is that it must initialize data-structures and the propagators and to achieve a first propagation. Thus it is not a good idea to create a new CSP with a bunch of new propagators every time we have selected elements to remove. A more efficient way to do this is to choose which elements to remove in a branching. This would allow to create search engines and CSP's only once. The resulting branching implementation is more complex, but it is worth doing it.

Branching on useful subsets Imagine that we decided to remove 3 elements from a set of 50 subsets with zero-reduced cost. Then, our heuristic will re-insert these 3 elements into some of the 50 subsets obtained by removing the selected elements. Thus at most 3 subsets will be new. It is a waste of time to use `BranchingClosing` and the branchings defined in `postLastBranching` on the 47 other subsets as we already know they are solutions.

We put a boolean array `modified` with n records in `CSPHeuristic`. Initially all these records are set to `false`. When our `BranchingHeuristic` re-inserts one elements into Sv_i, we set `modified[1]` record of this array to `true`. Then `BranchingClosing` and the branchings posted in `postLastBranching` check the values of the `modified` array depending on the variables they are operating on. If the given record is set to `false`, they know it is not worth to branch as we know which solution we will get in the end, and the current branching is skipped.

Delaying propagators Sometimes, propagators posted by the user with the provided `constrain` function are heavy (one such exemple is given with the VRP where we need to post propagators on graphs). With the same argument than in the precedent paragraph, it is not worth posting 50 sets of propagators if only 3 of them will be useful. Thus we decided to call the `constrain` to post propagators on Sv_i in `BranchingHeuristic`, the first time we insert an element into a Sv_i. Thus we post propagators when decision about inserting an element in a given subset is already taken. Of course one could argue that posting all propagators at the beginning is useful because it

prunes the domains of the Sv_i and then there could be subsets in which we know we cannot insert a given element. However, computational experiense shows that this possible pruning is not worth the waste of time we encounter by copying propagators all along the search tree.

9.3 Application: Vehicle Routing Problem

This section gives an example of an application of the Column Generation approach. It also gives an example on how we can use our implementation to solve such large-scale set covering problems.

9.3.1 CP formulation of VRP

We define a route by a sequence containing all the requests it serves: $R = [R_1, R_2, \ldots, R_k]$.

A very useful and efficient way to model routes is to define them as paths in a graph $G = (V, E)$. The set of nodes V would represent all client's location and all depots. The set of edges E would contain an edge (l_i, l_j) if it is possible for a vehicle to go from location l_i to location l_j. Several weights can be assigned to one edge. These could represent costs to travel between two locations (c_{ij}), distances (d_{ij}) or time needed to drive this distance (t_{ij}). Several weights can be assigned to nodes v too representing in particular clients' demand (d_v), time to unload it (u_v) and clients' returns (r_v).

Thus a route is a graph variable P such that

$$\texttt{path}(\texttt{P}, \texttt{0}, \texttt{0})$$

i.e. a path from the depot (node 0) and returning to the depot. These routes should meet some constraints depending on the kind of VRP we need to solve.

Capacitated VRP (CPRV) All vehicles have a maximal capacity that cannot be exceeded by the items it must serve to clients. If we denote by Q the total capacity of a vehicle, then we have the additional constraint

$$\sum_{i \in nodes(R)} d_i \leq Q$$

VRP with Pick-up and Delivering (VRPPD) In this particular case, clients may return some commodities that must be carried by the vehicle.

$$\sum_{i=1}^{j} r_{R_i} + \sum_{i=j+1}^{k} d_{R_i} \leq Q \qquad \forall j = 0, \ldots, k$$

...

A solution to VRP consists in finding which routes allows to serve all requests exactly once, i.e. we should solve a set partitionning problem with collection X. However because $R_i \subseteq R_j \Rightarrow Cost(R_i) \leq Cost(R_j)$, each optimal set covering on X will be a set partitionning. Thus the solution of VRP should repect the constraint

$\texttt{SC}(\texttt{N}, \texttt{T}, \texttt{X}, \texttt{Cost})$

Additional side constraints could be specified on the set of routes chosen to *cover* all requests.

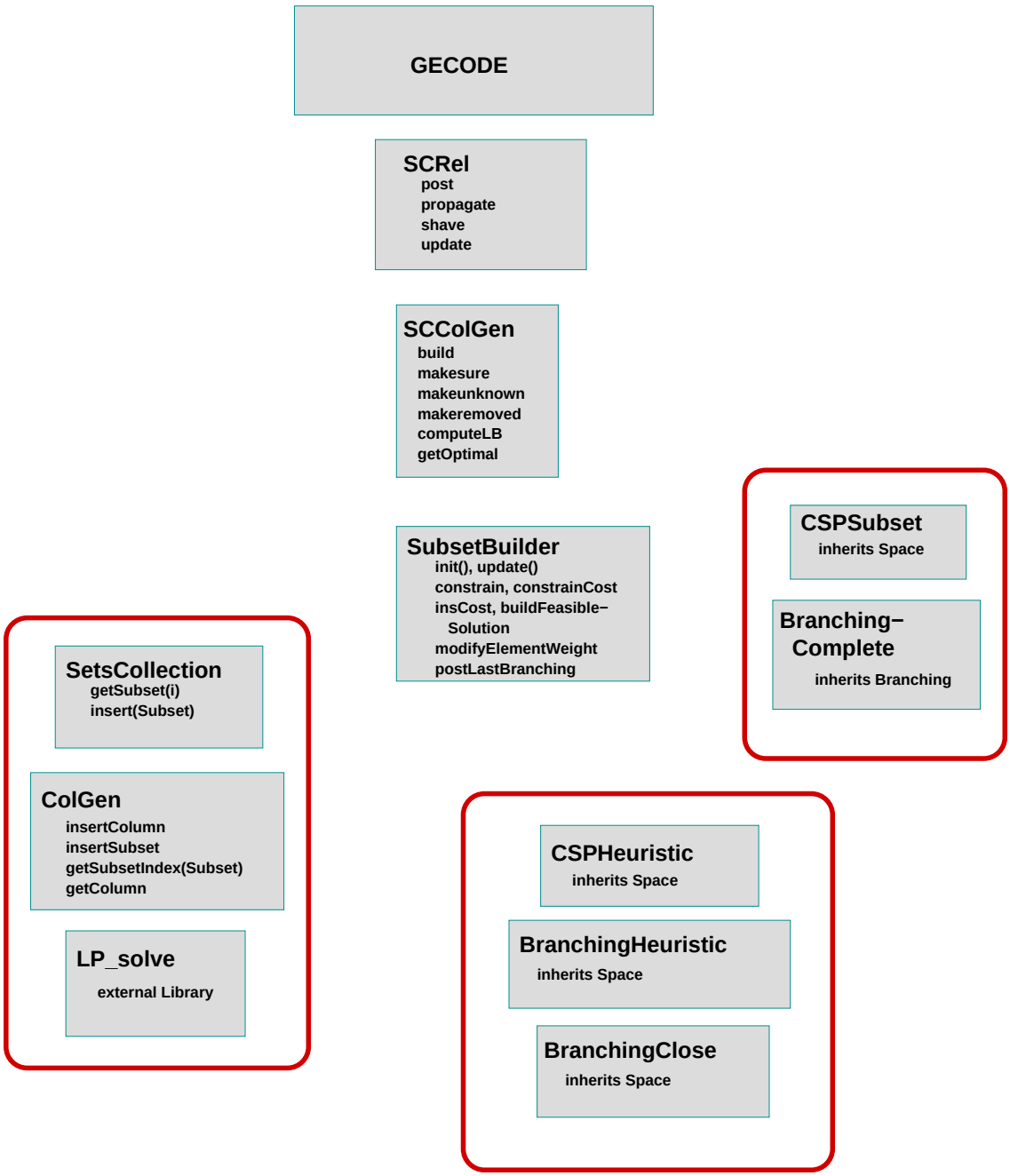

Figure 9.1: Main Concepts behind the Implementation of the Column Generation Approach

Chapter 10

Experiments

10.1 Comparison of the different propagators

From the computational results presented in [BHH+05], we observe that TA and OI achieve far less pruning than the two other methods (MD and SCL^*) for the same computational cost than MD. Thus we decided to compare $2SC$ with SCL^* and MD. Results are presented in Table-10.3.

We generated random set covering problems by specifying four different parameters

M The number of elements to cover

N The number of subsets in the collection X

S^- The minimum cardinality of the subsets in the collection X

S^+ The maximum cardinality of the subsets in the collection X

Then we created set covering problems with the following inputs:

- $\mathcal{U} = \{1, \ldots, M\}$
- $X = \{S_1, \ldots, S_N\}, S_i \subseteq \mathcal{U}(1 \leq i \leq N)$ is a collection of randomly generated subsets of $\mathcal{U}$
- $Cost(S_i) = 1(1 \leq i \leq 1)$

We set all costs to 1 in order to be able to use MD and to compare it with the other method. We tested a CSP with only one SC constraint $SC(N, T, \mathcal{U}, X, Cost)$ and try to find the optimal value N (thus computing SC^*) by branch and bound. The three relaxations were used: $2SC$, SCL^* and MD. A naive branching strategy was used; it branches on $S_1, S_2, \ldots$, always in this order. This is not efficient at all to solve set covering problems, but it allows to be sure that the three different propagators explore the search tree in the same manner. This removes any interaction between propagators and the search strategy that could lead to ambiguous results. Problems with at least one element not covered by any subset were deleted from the table, because they are not interesting for the comparison. These three relaxations were implemented as propagators in Gecode [gec]. The SCL^* propagator used the library lp_solve 5.5 [lps] to solve the linear relaxation. This library implements the simplex algorithm which allows incremental propagation along the search tree. Because these problems are pure set covering problems (without any side constraints), all methods perform worse when the shaving process is enabled. So we disabled the shaving in order to be more fair when comparing all methods. However, when there are a lot of side constraints, propagators with small time per call should be advantaged (MD and $2SC$).

Neither method outperforms uniformly the two others. However some general observations can be observed. MD performs usualy better when the value $\frac{M}{N}$ is small. This could be explained by the fact that such problems have a lot of solutions. Then there is less pruning to do and because MD is faster to compute the lower bound (even of worse quality), it can explore more solutions.

On the other hand, SCL^* performs much better when this value is higher ($\frac{M}{N} \geq 1$). In such problems there are less solutions and the good quality of the lower bound computed by SCL^* is worth its computational cost.

$2SC$ seems to be situated between MD and SCL^*. When MD performs best, $2SC$ usually achieves a better performance than SCL^*. In the opposite, when SCL^* performs best, $2SC$ is often better than MD. This is due to the fact that for $2SC$, we relax more the original set covering problem than for SCL^*. But MD is even more relaxed. Its computational cost is also situated between both of them. $2SC$ seems to be more stable in function of the input problem. Because $2SC$ decomposes each subsets in 2-sets, $2SC$ performs better when the sizes of the subsets are small.

The advantage of the incremental behaviour of the algorithm $2SC$ is demonstrated by Table-10.3 where we can observe that the overall time per

call to the propagator $2SC$ is often smaller than the one for the SCL^* propagator which is also incremental.

10.2 Analysis on the Parameters of the Heuristic

The heuristic presented in Section 7.5.3 requires the settings of some parameters. This section analyses the influence of each parameters on the efficiency of the heuristic. We selected a set of real-life instances of Vehicle Routing Problems with Pickup and Deliveries with Time-Windows (VRPPDTW). We implemented this problem being respectful of the framework described in Section 7.5 as presented in Section 9.2.4. Thus this provides experimental results that can validate the concepts of our heuristic.

Problems were selected from the ones provided in [vrp]. They provides VRPPDTW fo a size 200. Their size was decreased to 50 because first experimental experiences show that this size is big enough to show the main important facts about the analysis of the parameters of our heuristic. VRPPDTW was selected because it is a real-life problems, that arises often in practice, perhaps more than the standard Vehicle Routing Problem. The pickup constraints were not implemented.

In order to do experiments on a wise range of values for all the parameters, first we did some experiments with random parameters. We selected the pairs parameter-value (p, v) that seemed to lead to the best results independently of the values of the other parameters. Then for each parameter p_t, we tried the heuristic with different values for p_t and we set all the other parameters p_o to the best value identified in the previous step.

Results are showed in Fig.A.1 to Fig.A.5 in Appendix A. They reveal interesting points for the design of the heuristic that would need more investigation.

Parameter TS Fig.A.2 shows that stopping search early in the heuristic is useful, even if we search for only one solution. This seems to be the case because some choices of the elements to remove could increase the difficulty of finding subsets of negative reduced costs. Perhaps a smaller value of 2000 could be more advantegeous. However, as it is illustrated in the first subfigure of Fig.A.2, a too small value for TS can be unable the finding fo a solution. Adding a rule that increase the TS parameter after some iterations if we

M	**N**	S^-	S^+	$2SC$				SCL^*				MD			
				Sol	Time	failures	TC	Sol	Time	failures	TC	Sol	Time	failures	TC
10	500	2	6	223	¿ 30	38342	0,38	222	¿ 30	38760	0,37	**2**	**12,4**	263731	0,02
10	500	2	10	303	¿ 30	19418	0,73	235	¿ 30	35103	0,41	**1**	**8,46**	127222	0,03
10	200	2	6	**2**	4,09	19507	0,1	**2**	7,46	19507	0,18	**2**	**1,66**	45025	0,02
10	200	2	10	**1**	6,42	19703	0,16	**1**	8,11	19703	0,2	**1**	**1,18**	20875	0,03
20	200	2	4	**5**	**6,83**	55259	0,06	**5**	10,16	18946	0,26	9	¿ 30	771069	0,02
20	200	2	6	**4**	**4,5**	21706	0,1	**4**	10,4	19132	0,26	5	¿ 30	619822	0,02
20	200	2	10	**3**	**8,64**	22412	0,19	**3**	9,64	19317	0,24	**3**	¿ 30	398993	0,04
20	200	4	14	**2**	12,78	19507	0,32	**2**	12,28	19507	0,3	**2**	**8,97**	49310	0,09
20	200	8	10	**3**	15,27	25454	0,29	**3**	**9,57**	19407	0,24	**3**	¿ 30	266804	0,05
20	200	8	14	**2**	16,11	20092	0,39	**2**	**10,33**	19507	0,25	**2**	13,13	102288	0,06
50	500	2	4	141	¿ 30	64416	0,23	228	¿ 30	37053	0,39	**39**	¿ 30	551144	0,03
50	500	2	6	233	¿ 30	35513	0,41	236	¿ 30	34842	0,42	**28**	¿ 30	446423	0,03
50	500	2	10	314	¿ 30	17205	0,82	242	¿ 30	33228	0,44	**19**	¿ 30	318013	0,05
50	500	4	14	370	¿ 30	8405	1,62	260	¿ 30	28770	0,5	**12**	¿ 30	226824	0,07
50	500	8	10	365	¿ 30	9062	1,51	266	¿ 30	27381	0,53	**15**	¿ 30	212295	0,07
50	500	8	14	388	¿ 30	6259	2,12	266	¿ 30	27442	0,52	**11**	¿ 30	174868	0,08
10	50	2	6	**2**	**0,1**	1132	0,04	**2**	0,25	1132	0,1	**2**	**0,1**	3153	0,01
10	50	2	10	**1**	0,14	1178	0,05	**1**	0,26	1178	0,1	**1**	**0,09**	1446	0,03
100	500	2	4	**138**	¿ 30	67100	0,22	236	¿ 30	34938	0,41	199	¿ 30	511838	0,03
100	500	2	6	231	¿ 30	36210	0,4	244	¿ 30	32772	0,44	**104**	¿ 30	382419	0,04
100	500	2	10	312	¿ 30	17646	0,8	254	¿ 30	30312	0,48	**94**	¿ 30	269540	0,05
100	500	4	14	372	¿ 30	8227	1,65	269	¿ 30	26604	0,54	**45**	¿ 30	167095	0,09
100	500	8	10	364	¿ 30	9201	1,49	268	¿ 30	26897	0,53	**62**	¿ 30	176621	0,08
100	500	8	14	388	¿ 30	6257	2,12	277	¿ 30	24780	0,58	**42**	¿ 30	153117	0,1
50	200	2	4	24	¿ 30	291570	0,05	**15**	**13,29**	18405	0,35	36	¿ 30	749486	0,02
50	200	2	6	22	¿ 30	215045	0,07	**11**	**22,75**	20429	0,54	24	¿ 30	500720	0,03
50	200	2	10	11	¿ 30	126201	0,12	**7**	**12,53**	19119	0,32	15	¿ 30	393054	0,04
50	200	4	14	7	¿ 30	58297	0,25	**6**	**25,2**	21042	0,58	10	¿ 30	203992	0,07
50	200	8	10	10	¿ 30	62392	0,23	**7**	¿ 30	20723	0,7	12	¿ 30	243080	0,06
50	200	8	14	7	¿ 30	49796	0,29	**6**	¿ 30	20909	0,69	8	¿ 30	159690	0,09
400	1000	2	4	**791**	¿ 30	21935	0,64	862	¿ 30	9544	1,38	980	¿ 30	117592	0,1
400	1000	2	6	**836**	¿ 30	13515	1	876	¿ 30	7743	1,66	964	¿ 30	128167	0,09
400	1000	2	10	**885**	¿ 30	6651	1,88	**885**	¿ 30	6640	1,89	958	¿ 30	88089	0,14
400	1000	4	14	920	¿ 30	3179	3,41	**897**	¿ 30	5277	2,29	930	¿ 30	61146	0,22
400	1000	8	10	918	¿ 30	3330	3,3	**913**	¿ 30	3769	3	940	¿ 30	64338	0,2
400	1000	8	14	936	¿ 30	2029	4,69	**906**	¿ 30	4393	2,66	934	¿ 30	49467	0,27
400	500	2	10	385	¿ 30	15628	0,82	**318**	¿ 30	16981	0,84	462	¿ 30	121025	0,1
400	500	4	14	368	¿ 30	8755	1,56	**325**	¿ 30	15231	0,93	462	¿ 30	89955	0,14
400	500	8	10	364	¿ 30	9265	1,48	**336**	¿ 30	13385	1,05	463	¿ 30	73104	0,17
400	500	8	14	389	¿ 30	6164	2,15	**340**	¿ 30	12844	1,09	446	¿ 30	55121	0,23

M	**N**	S^-	S^+	$2SC$				SCL^*				MD			
				Sol	Time	failures	TC	Sol	Time	failures	TC	Sol	Time	failures	TC
50	50	2	6	15	¿ 30	321407	0,04	**14**	**0,28**	732	0,17	21	¿ 30	473335	0,02
50	50	2	10	10	¿ 30	218047	0,06	**9**	**0,37**	891	0,18	12	¿ 30	272193	0,04
50	50	4	14	**6**	¿ 30	110772	0,12	**6**	**0,41**	997	0,18	8	¿ 30	183474	0,06
50	50	8	10	**8**	¿ 30	154642	0,09	**8**	**0,59**	965	0,26	10	¿ 30	201884	0,06
50	50	8	14	7	¿ 30	129238	0,11	**6**	**0,61**	1044	0,25	7	¿ 30	143366	0,08
200	200	4	14	90	¿ 30	33609	0,42	**36**	¿ 30	15550	0,93	138	¿ 30	91178	0,15
200	200	8	14	73	¿ 30	30358	0,47	**33**	¿ 30	15642	0,93	84	¿ 30	89203	0,16
100	50	4	14	25	¿ 30	94787	0,11	**17**	**0,41**	686	0,27	23	¿ 30	130224	0,09
100	50	8	14	25	¿ 30	88691	0,12	**16**	**0,53**	757	0,31	29	¿ 30	95007	0,11
400	200	8	14	162	¿ 30	13554	0,75	**74**	¿ 30	10405	1,39	171	¿ 30	65742	0,18
50	20	4	14	**7**	0,08	471	0,06	**7**	**0,04**	98	0,15	**7**	0,14	775	0,06
50	20	8	10	**9**	0,42	2956	0,05	**9**	**0,03**	86	0,13	**9**	1,42	9487	0,06
50	20	8	14	**7**	0,47	2567	0,07	**7**	**0,05**	106	0,18	**7**	2,25	9991	0,08

Table 10.3: Experimental results. M is the number of elements to cover(i.e. the cardinality of $\mathcal{U}$). N is the size of the collection of subsets of $\mathcal{U}$. S^+ (resp. S^-) is the maximum cardinality of subsets S_i (resp. the minimum cardinality of S_i). Sol is the smallest solution found for N. $Time$ is the total search time (search + propagation). A 30 seconds limit was used. $failures$ is the number of times the algorithm failed during search. TC is the time per call of the propagator in milliseconds.

don't find any solution could be very useful not to encounter this problem.

Parameter D For the parameter D specifying the level on randomness in the step of removing elements from the current cover, the best value seems to be 20 as indicated in [Sha98].

Parameter ITR Fig.A.3 shows that beginning by removing four elements from the current solution seems the best choice. One should note that the time after which we stopped each iteration of the heuristic is function of the ITR parameter. This is due to the fact that removing more values enlarge the neighborhood and we some more time to initialize all data-structure in the CP engine. More extensive analysis would analyse these two parameters more separatedly. One more qualitative sign that 4 is a good value for ITR is the fact that the curve is not convex on three figures, although this does not happen with smaller values. I interpret that as the fact that we are at the border between a too small value that explore the search space to slowly (thus giving a smooth convex amelioration curve) and a too big value that explore a too big search space, thus finding good values during the first iterations and not having the time to find a subset of negative cost for the next iterations.

Parameter MA Big values for the maximum number of added subsets before recomputing optimal dual values seems better than assigning small values for this parameter. This could come from the fact that we explore a larger search space before solving the new Master Problem, thus allowing the algorithm to generate more diversified subsets. From the curves illustrated in Fig.A.4 we deduce than we should do more experiences with bigger values than 20.

Parameter MT Experimental results inllustrated in Fig.A.5 show that the number of iterations after which we change the number of removed elements does not seem to impact performance of the heuristic. One interesting observation is that the decrement in the number of elements to remove seems to happen more or less at the same time, independently of the value of MT

With or without branch-and-bound ? We did tests to determine whether using branch-and-boud instead of stopping after the first solution found is useful. Fig.A.6 and Fig.A.7 show the efficiency of the heuristic without and

with branch-and-bound. Not using branch-and-bound leads to better solutions. This is illustrated too in the fact that stopping search early when using branch-and-bound is more efficient than searching more.

Part V

Conclusion

Chapter 11

Conclusion

This work explored two research directions to solve *set covering* problems in a CP framework, focusing mainly on the integration of operational research and constraint programming.

In the first part of this work, we have analysed the structure of the *set covering* problem and have proposed a generic global constraint approach to solve it. The proposed approach requires a lower bound on the optimal cost of a cover. After having presented a number of lower bounds found in the litterature, we have introduced a new lower bound obtained by relaxing a *set covering* problem to an *edge covering* problem, leading to specific benefits in particular SCP's.

The first main contribution of this work, the 2SC relaxation, its incremental algorithm and its implementation were accepted and presented at the conference *Journées Françaises de la Programmation par Contraintes* (JFPC - Paris, June 2007). We consider that we have achieved our objectives for this part: problem analysis, extensive survey of the existing litterature, proposal of a relevant solution with theoretical justification, efficient implementation of the idea and validation by experimental results.

This part of the research work has been finalized by end of February with the preparation and submission of a paper to the JFPC conference. The second part of this work consists in a first step of the extension of the generic SC constraint, allowing to handle very large *set covering* problems whose collection of subsets is too big to be considered explicitely.

In the second part, we started by a litterature survey about *column generation*. After discussions with Pr Deville and Pr Wolsey, we designed our initial solution (described in the chapter 7) and started immediately the basic

implementation of our framework.

We have initiated the validation of the heuristic (initial testing) and develop the final results in this report.

This second part, the integration of column generation to extend our first version of the SC constraint met our initial objectives about this research direction: 1) proposal of a flexible framework to integrate heuristics, exact algorithms and column generation for a CP approach to very large-scale set covering problems and 2) initial analysis to fine tune the proposed heuristic.

At this stage, our preliminary work could be used as a platform to validate the research direction taken by enhancing the optimal search of the subsets with minimum reduced cost. We believe this is the next most difficult and challenging aspect of this approach. Due to specific time constraints (from March to May 2007) and to reach preliminary conclusions, we had to limit our objectives.

For the two parts of the work, we decided on purpose to orient it in a *research* way of thinking, not focusing on specific implementation objectives but more as an exploration work. As a direct consequence, we have decided to take the limited time available to go as far as we could in the implementation and validation of our second approach.

What should the reader remember from this work ?

First the introduction of the *2SC relaxation*, representing a potential in the case the collection of subsets contains subsets with very small cardinality.

Second, the structure of the *set covering problem* is such that the set of solutions of a set covering problem is not strongly constrained, only trivial rules exist to prune according to the fact that we want to find a cover. Thus the pruning can only be done according to the optimization constraint, i.e. the cover we want to find should have a cost smaller than a given value. This kind of constraint is very difficult to handle in a pure CP framework. Thus integration of operational research into CP is useful for such problems.

Third, the *column generation* approach proposed in this work is a flexible way to let the user specify some additional structure of the set covering problem he wants to solve. SC is a generalization of many problems, but we tend to drop too much structure when relaxing problems into a SCP formulation. Our framework allows the user to specify this additional structure.

What could be the next challenging activities based on this work ?

Ideally, future work should be directed to continuing the implementation of the optimal search of subsets with minimum reduced cost and more

in-depth validation of this approach. At this stage, we anticipate that by specializing the framework proposed in this work to other problems whose generalization is SCP could be worth the additional implementation cost to allow the user to solve specific but well-known problems in a CP framework.

Simple reduction tests could also be applied to reduce the size of the relaxation to be solved inside our propagator. This would lead to faster time per call. The challenge is that these reductions should be managed incrementally to limit the additional constraints posted along the search tree.

Bibliography

[ALM+98] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy, *Proof verification and the hardness of approximation problems*, J. ACM **45** (1998), no. 3, 501–555.

[AÖEOP02] Ravindra K. Ahuja, Özlem Ergun, James B. Orlin, and Abraham P. Punnen, *A survey of very large-scale neighborhood search techniques*, Discrete Appl. Math. **123** (2002), no. 1-3, 75–102.

[BCT02] Nicolas Beldiceanu, Mats Carlsson, and Sven Thiel, *Cost-filtering algorithms for the two sides of the sum of weights of distinct values constraint*, Tech. Report 14, Swedish Institute of Computer Science, 2002, http://www.sics.se/libindex.html#Technical.

[Ber57] Claude Berge, *Two theorems in graph theory*, Proceedings of the National Academy of Sciences of the United States of America, vol. 43-9, 1957, pp. 842–844.

[BFS+00] Bruno De Backer, Vincent Furnon, Paul Shaw, Philip Kilby, and Patrick Prosser, *Solving vehicle routing problems using constraint programming and metaheuristics*, Journal of Heuristics **6** (2000), no. 4, 501–523.

[BHH+05] C. Bessière, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh, *Filtering Algorithms for the* NValue *Constraint*, Proceedings of the 7th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR-05) (Prague, Czech Republic) (Roman Barták and Michela Milano, eds.), Lecture Notes in Computer Science, vol. 3524, Springer-Verlag, May 2005, pp. 79–93.

[BJN+98] Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh, and Pamela H. Vance, *Branch-and-price:*

Column generation for solving huge integer programs, Operational Research **46** (1998), no. 3, 316–329.

[BL94] Julien Bramel and David Simchi Levi, *On the effectiveness of set covering formulations for the vehicle routing problem with time windows*, –.

[CFT98] A. Caprara, M. Fischetti, and P. Toth, *Algorithms for the set covering problem*, 1998.

[CFT99] Alberto Caprara, Matteo Fischetti, and Paolo Toth, *A heuristic method for the set covering problem*, Oper. Res. **47** (1999), no. 5, 730–743.

[CNS97] S. Ceria, P. Nobili, and A. Sassano, *Set covering problem*, Annotated Bibliographies in Combinatorial Optimization (M. Dell'Amico, F. Maffioli, and S. Martello, eds.), John Wiley, 1997.

[DY91] SOUMIS F. DUMAS Y., DESROSIERS J., *The pickup and delivery problem with time windows*, European journal of operational research **54** (1991), 7–22.

[EDM92] Elia El-Darzi and Gautam Mitra, *Solution of set-covering and set-partitioning problems using assignment relaxations*, The Journal of the Operational Research Society **43** (1992), no. 5, 483–493.

[FJK+02] Torsten Fahle, Ulrich Junker, Stefan E. Karisch, Niklas Kohl, Meinolf Sellmann, and Bo Vaaben, *Constraint programming based column generation for crew assignment*, Journal of Heuristics **8** (2002), no. 1, 59–81.

[FR76] B. A. Foster and D. M. Ryan, *An integer programming approach to the vehicle scheduling problem*, Operational Research Quarterly **27** (1976), no. 2, 367–384.

[gec] *Gecode: Generic constraint development environment, 2005. library*, Available as an open-source from www.gecode.org.

[Ger97] Carmen Gervet, *Interval propagation to reason about sets: Definition and implementation of a practical language*, Constraints **1** (1997), no. 3, 191–244.

[GSMD99] Michel Gamache, Francois Soumis, Gerald Marquis, and Jacques Desrosiers, *A column generation approach for large-scale aircrew rostering problems*, Operations Research **47** (1999), no. 2, 247–263.

[HP93] Karla L. Hoffman and Manfred Padberg, *Solving airline crew scheduling problems by branch-and-cut*, Manage. Sci. **39** (1993), no. 6, 657–682.

[HWC74] Michael Held, Philip Wolfe, and Harlan P. Crowder, *Validation of subgradient optimization*, Journal Mathematical Programming **6** (1974), no. 1, 62–88.

[JKK+] Ulrich Junker, Stefan E. Karisch, Niklas Kohl, Bo Vaaben, Torsten Fahle, and Meinolf Sellmann, *A framework for constraint programming based column generation.*

[JL87] J. Jaffar and J.-L. Lassez, *Constraint logic programming*, POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (New York, NY, USA), ACM Press, 1987, pp. 111–119.

[JM92] Current J and O'Kelly M, *Locating emergency warning sirens.*, Decision Sciences **23** (1992), no. 1, 221–234.

[JMSY92] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap, *The clp(r) language and system*, ACM Trans. Program. Lang. Syst. **14** (1992), no. 3, 339–395.

[Kar39] W. Karush, *Minima of functions of several variables with inequalities as side constraints*, Master's thesis, Dept. of Mathematics, Univ. of Chicago, Chicago, Illinois, 1939.

[Kar72] Richard M. Karp, *Reducibility among combinatorial problems*, Complexity of Computer Computations (R. E. Miller and J. W. Thatcher, eds.), Plenum Press, 1972, pp. 85–103.

[KT50] H. W. Kuhn and A. W. Tucker, *Nonlinear programming*, Proceedings of 2nd Berkeley Symposium, Berkeley: University of California Press, 1950, pp. 481–492.

[Kuh55] Harold W. Kuhn, *The hungarian method for the assignment problem*, Naval Research Logistic Quarterly **2** (1955), 83–97.

[lps] *lp_solve library.*

[LW79] Nemhauser George L. and Glenn M. Weber, *Optimal set partitioning, matchings and lagrangian duality*, Naval Research Logistics Quarterly **26** (1979), 553–563.

[LY94] Carsten Lund and Mihalis Yannakakis, *On the hardness of approximating minimization problems*, J. ACM **41** (1994), no. 5, 960–981.

[MN99] Kurt Mehlhorn and Stefan Näher, *Leda: a platform for combinatorial and geometric computing*, Cambridge University Press, New York, USA, 1999.

[P.91] Van Hentenryck P., *The clp language chip: constraint solving and applications*, Compcon Spring '91. Digest of Papers, 1991, pp. 382–387.

[Pie68] John F. Pierce, *Application of combinatorial programming to a class of all-zero-one integer programming problems*, Management Science **15** (1968), no. 3, 191–209.

[PR99] François Pachet and Pierre Roy, *Automatic generation of music programs*, CP '99: Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (London, UK), Springer-Verlag, 1999, pp. 331–345.

[Pug96] J F Puget, *Finite set intervals*, Proceedings Workshop on Set Constraints, held in Conjunction with CP'96, 1996.

[Rég94] Jean-Charles Régin, *A filtering algorithm for constraints of difference in csps*, AAAI '94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 1) (Menlo Park, CA, USA), American Association for Artificial Intelligence, 1994, pp. 362–367.

[Sha98] Paul Shaw, *Using constraint programming and local search methods to solve vehicle routing problems.*

[Smo98] Gert Smolka, *Concurrent constraint programming based on functional programming*, Programming Languages and Systems (Lisbon, Portugal) (Chris Hankin, ed.), Lecture Notes in Computer Science, vol. 1381, Springer-Verlag, 1998, pp. 1–11.

[SS] M. Sol and M.W.P. Savelsbergh, *A branch-and-price algorithm for the pickup and delivery problem with time windows.*

[SZSF02] Meinolf Sellmann, Kyriakos Zervoudakis, Panagiotis Stamatopoulos, and Torsten Fahle, *Crew assignment via constraint programming: Integrating column generation and heuristic tree search*, Annals of Operations Research **115** (2002), 207–225.

[TLZO01] K. Tan, L. Lee, Q. Zhu, and K. Ou, *Heuristic methods for vehicle routing problem with time windows*, 2001.

[TSRB71] Constantine Toregas, Ralph Swain, Charles ReVelle, and Lawrence Bergman, *The location of emergency service facilities*, Operations Research **19** (1971), no. 19, 1363–1373.

[Tur41] P. Turán, *On an extremal problem in graph theory*, Mat. Fiz. Lapok **48** (1941), 436–452.

[van01] W. J. van Hoeve, *The alldifferent constraint: A survey*, 2001.

[vrp] *The vrp web*, http://neo.lcc.uma.es/radi-aeb/WebVRP/.

[Wol98] L.A. Wolsey, *Integer programming*, Wiley, New York, 1998.

Appendix A

Experiences for Tuning Parameters of the Heuristic

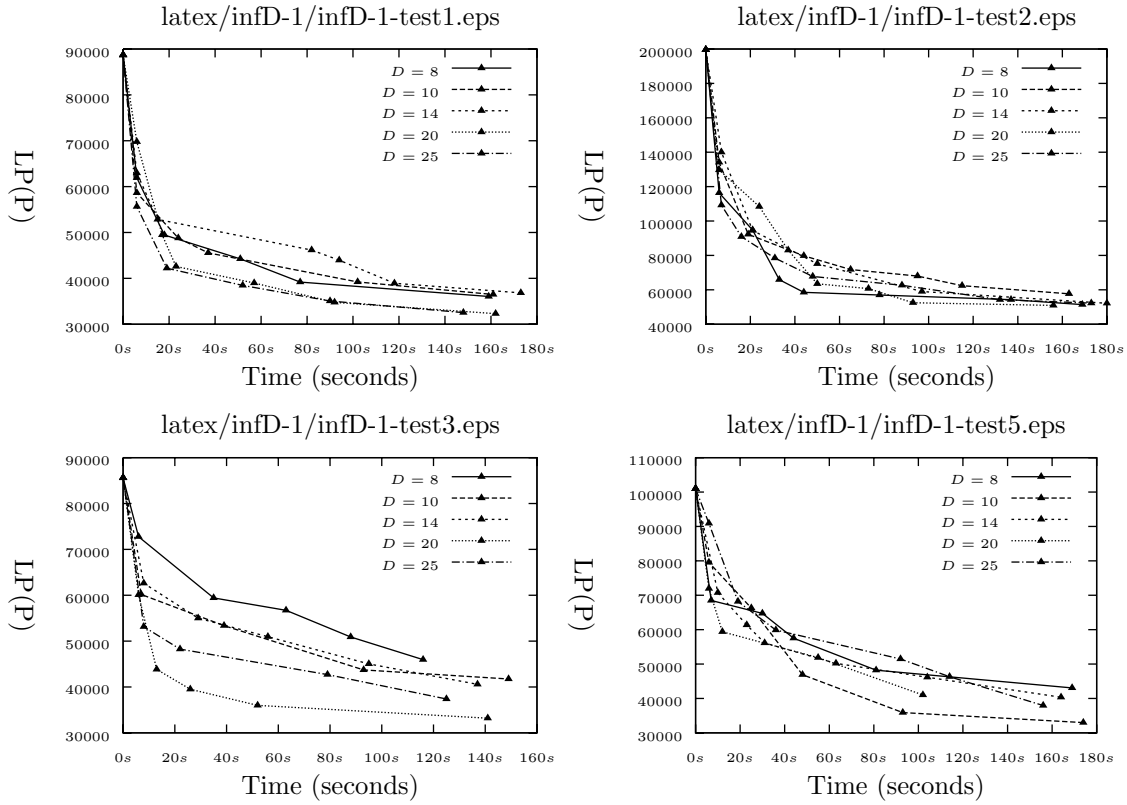

Figure A.1: Analysis of the impact of parameters: DFILE=test{1-3,5}.vrp, MA=25, MT=5, MI=60000, ITR=4, MTR=30, INCTR=-1, TS=3500

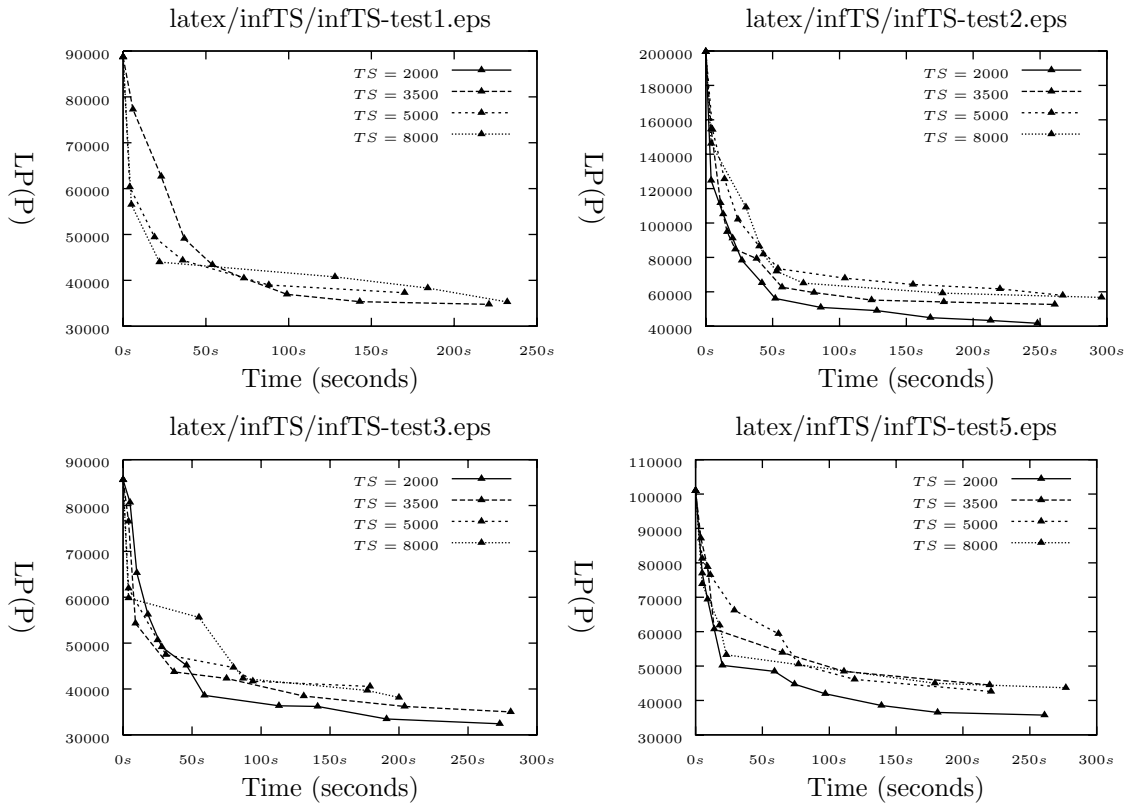

Figure A.2: Analysis of the impact of parameters: TSFILE=test{1-3,5}.vrp, MA=20, MT=10, MI=60000, ITR=4, MTR=30, INCTR=-1, D=20

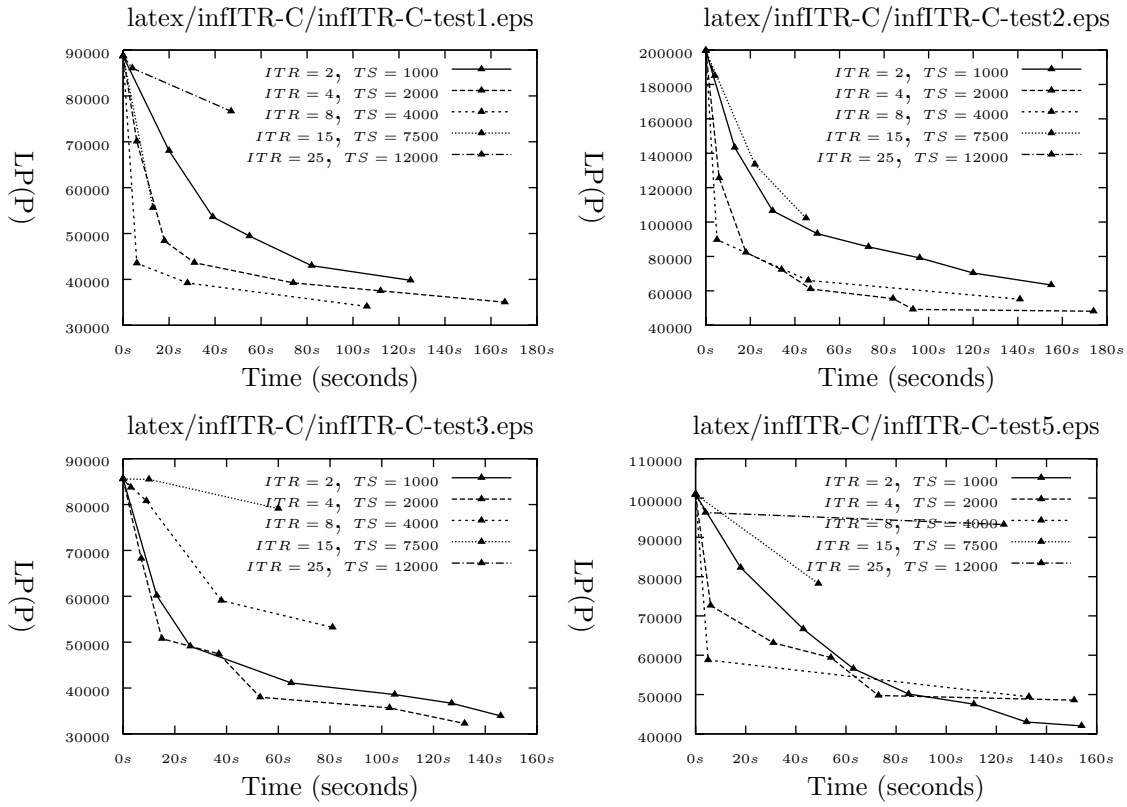

Figure A.3: Analysis of the impact of parameters: ITR, TSFILE=test{1-3,5}.vrp, MA=25, MT=5, MI=60000, MTR=30, INCTR=-1, D=10

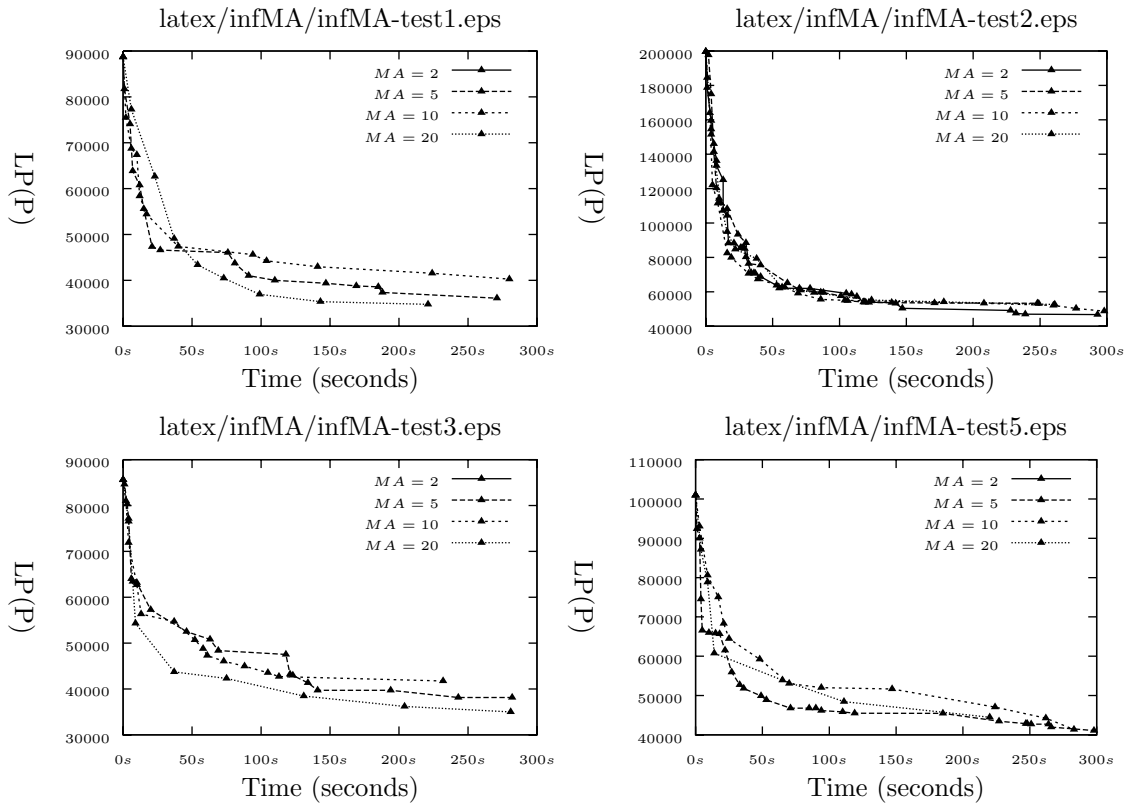

Figure A.4: Analysis of the impact of parameters: MAFILE=test{1-3,5}.vrp, MT=10, MI=60000, ITR=4, MTR=30, INCTR=-1, D=20, TS=3500

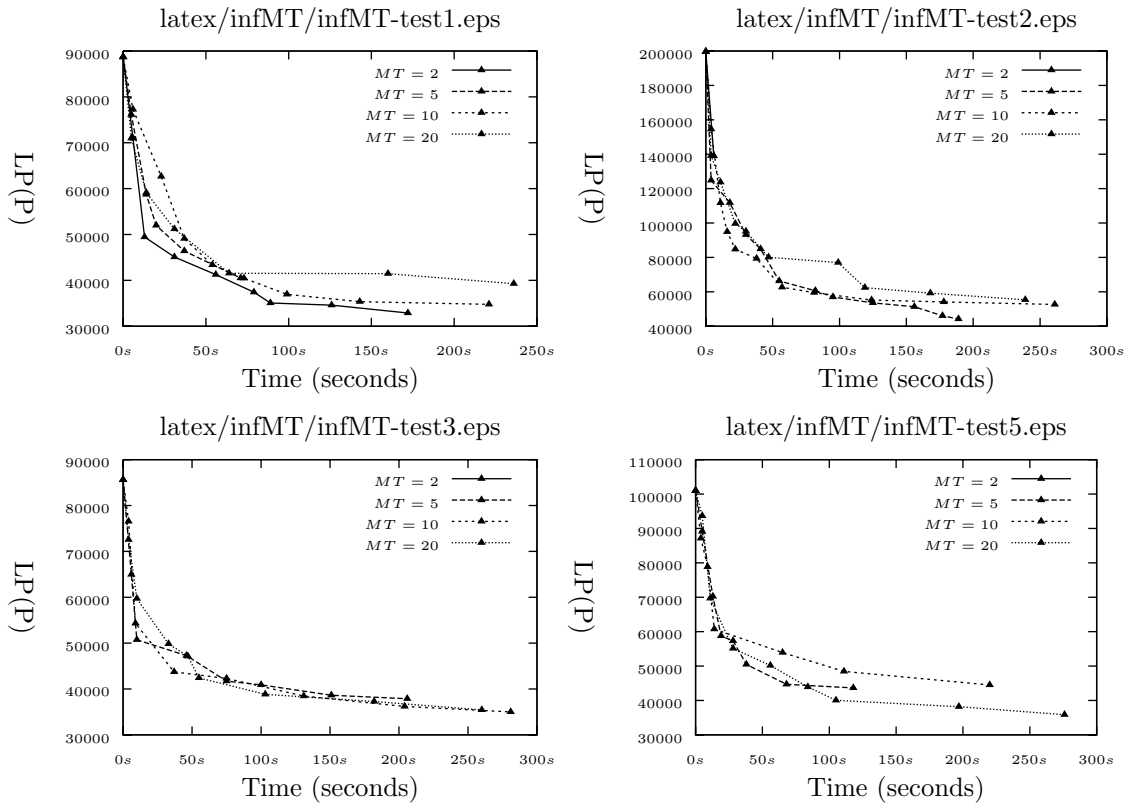

Figure A.5: Analysis of the impact of parameters: MTFILE=test{1-3,5}.vrp, MA=20, MI=60000, ITR=4, MTR=30, INCTR=-1, D=20, TS=3500

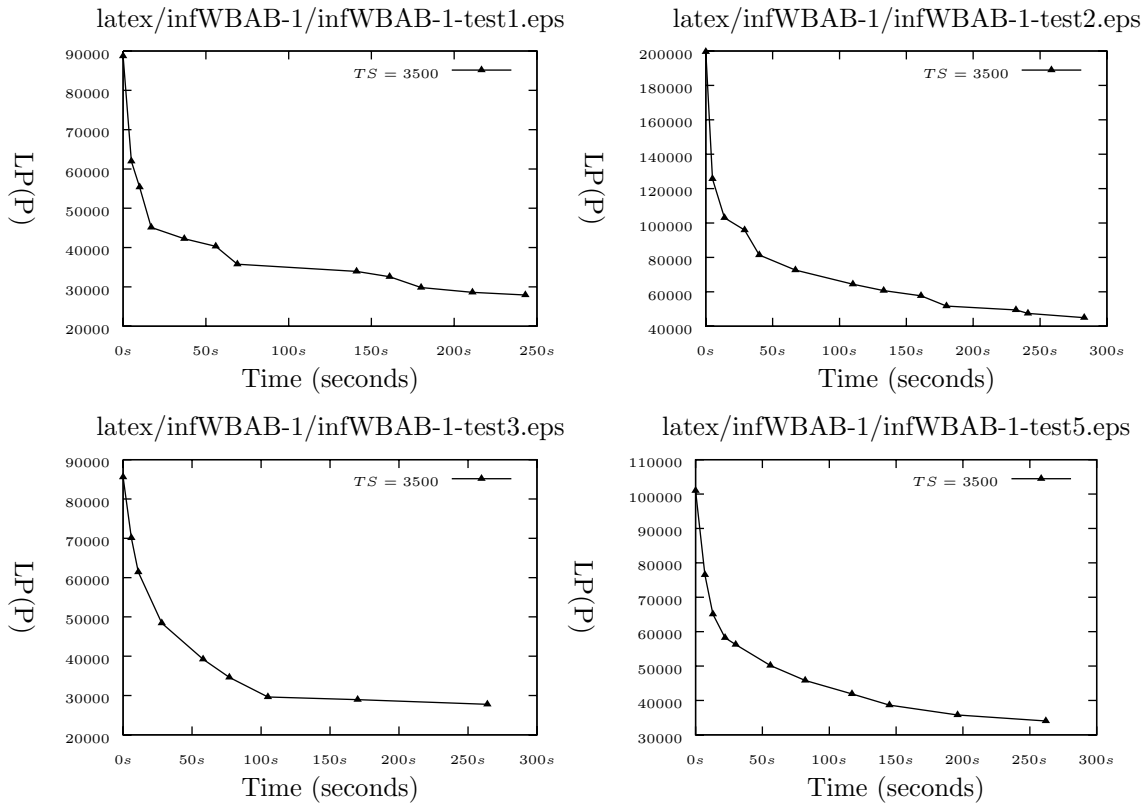

Figure A.6: Analysis of the impact of parameters: TSFILE=test{1-3,5}.vrp, MA=20, MT=10, MI=60000, ITR=4, MTR=30, INCTR=-1, D=20

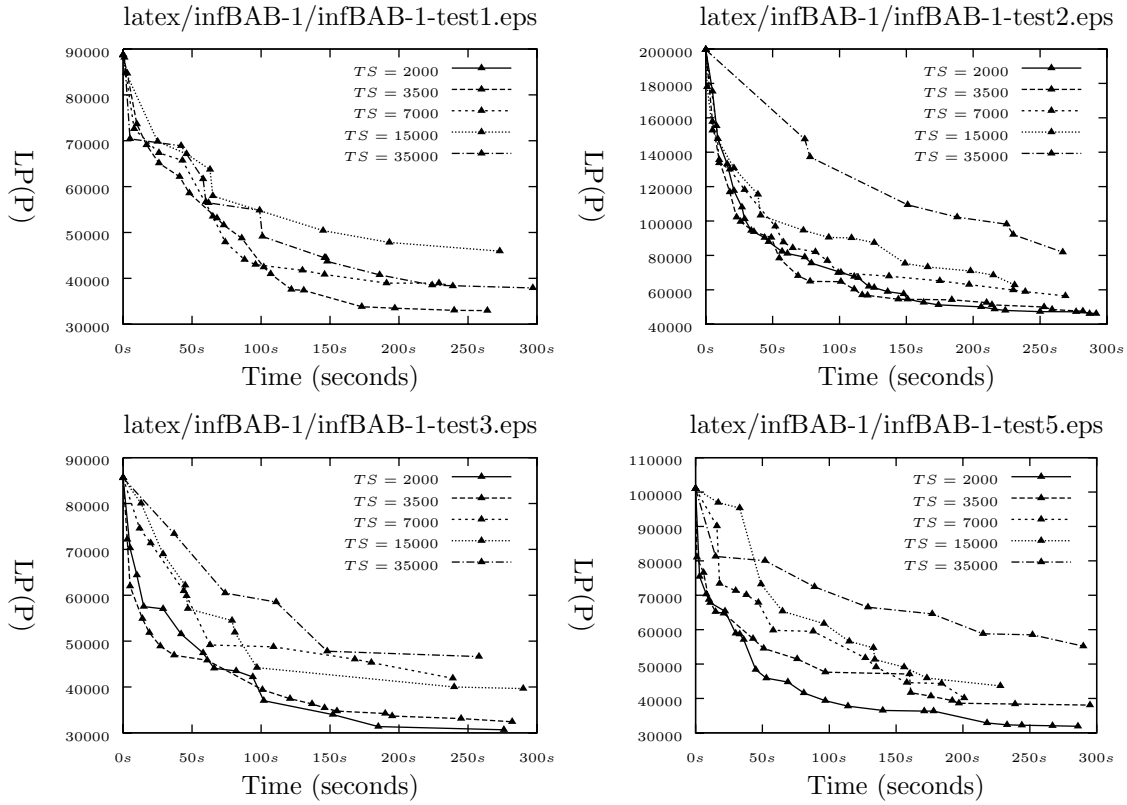

Figure A.7: Analysis of the impact of parameters: TSFILE=test{1-3,5}.vrp, MA=20, MT=10, MI=60000, ITR=4, MTR=30, INCTR=-1, D=20