UNIVERSITÀ DEGLI STUDI DI PADOVA

Sede Amministrativa: Università degli Studi di Padova

Dipartimento di Matematica Pura ed Applicata

**Dottorato di Ricerca in
Matematica Computazionale**

XVIII Ciclo

Exact and Heuristic Methods for Mixed Integer Linear Programs

Tesi di Dottorato di:
Livio Bertacco
[illegible]

Il Coordinatore
Ch.mo Prof. Michelangelo Conforti

Il Supervisore
Ch.mo Prof. Matteo Fischetti
[illegible]

Padova, 30 Dicembre 2005

Contents

Acknowledgments

Many people have taught, encouraged, supported, helped, and advised me during the time in which I worked on this thesis. I wish to express my deepest gratitude to all of them.

First of all, my thanks go to my advisor, Matteo Fischetti; he introduced me to the Operations Research and Combinatorial Optimization, helping me with suggestions and encouragements which have been indispensable for the realization of this work. He has been able to set up, during the years, a research group in which an exciting scientific activity is performed in a really special and friendly surrounding. I am indebted to all the members of this enlarged group. A special thank goes to Andrea Lodi and Lorenzo Brunetta, for their friendship and support; they have helped me with many discussions on aspects of this thesis. Thanks also to my friends and colleagues, Tatiana Bassetto and Cristiano Saturni, for the numerous brainstorming sessions and all the time spent together.

Particular thanks to Prof. Robert Wiesmantel and all the ADONET network for the excellent doctoral schools and workshops organized throughout Europe. Not only these have always proved of unsurpassed scientific level, with a highly enjoyable international atmosphere, but also have given me the chance to meet so many inspiring people working on Discrete Optimization.

I don't want to forget all the people at the Department of Mathematics and at DEI, and their work supporting the research activity. Thanks for doing this so kindly.

Finally, I warmly want to thank my family.

Padova, December 30, 2005

Livio Bertacco

List of Figures

List of Tables

Chapter 1

Mixed-Integer Programming

1.1 Introduction

Many problems in science, technology, business, and society can be modeled as mixed integer programming (MIP) problems and, as a matter of fact, in the last decade the use of integer programming models and software has increased dramatically. Nowadays, thanks to the progress of computer hardware and, even more, advances in the solution techniques and algorithms, it is possible to solve problems with thousands of integer variables on personal computers, and to obtain high quality solutions to problems with millions of variables (for example, set partitioning problems) often in a matter of minutes.

Among the currently most successful methods, to solve MIP problems, are linear programming (LP, for short) based branch-and-bound algorithms, where the underlying linear programs are possibly strengthened by cutting planes.

Todays codes, however, have become increasingly complex with the incorporation of sophisticated algorithmic components, such as preprocessing and probing techniques, cutting plane algorithms, advanced search strategies, and primal heuristics.

1.2 Definitions

A *mixed integer program* (MIP) is a system of the following form:

$$\begin{array}{rll} z_{\text{MIP}} = & \min & c^T x \\ & \text{subject to} & Ax \le b \\ & & \underline{x} \le x \le \overline{x} \\ & & x \in \mathbb{Z}^{\mathcal{G}} \times \mathbb{R}^{\mathcal{C}}, \end{array} \tag{1.1}$$

where $A \in \mathbb{Q}^{\mathcal{M} \times (\mathcal{G} \cup \mathcal{C})}$, $c \in \mathbb{Q}^{\mathcal{G} \cup \mathcal{C}}$, $b \in \mathbb{Q}^{\mathcal{M}}$. Here, $c^T x$ is the *objective function*, $Ax \le b$ are the *constraints* of the MIP, and $\mathcal{M}$, $\mathcal{G}$ and $\mathcal{C}$ are non-empty, finite sets with $\mathcal{G}$ and $\mathcal{C}$ disjoint. Without loss of generality, we may assume that the elements of $\mathcal{M}$, $\mathcal{G}$ and $\mathcal{C}$ are represented by numbers, i.e., $\mathcal{M} = \{1, \dots, m\}$,

$\mathcal{G} = \{1, \dots, p\}$ and $\mathcal{C} = \{p+1, \dots, n\}$. The vectors $\underline{x} \in (\mathbb{Q} \cup \{-\infty\})^{\mathcal{G}\cup\mathcal{C}}, and \overline{x} \in (\mathbb{Q} \cup \{\infty\})^{\mathcal{G}\cup\mathcal{C}}$ are called *lower* and *upper bounds* on x, respectively. A variable $x_j, j \in \mathcal{G} \cup \mathcal{C}$, is *unbounded from below* (*above*), if $\underline{x}_j = -\infty$ ($\overline{x}_j = \infty$). An integer variable $x_j \in \mathbb{Z}$ with $\underline{x}_j = 0$ and $\overline{x}_j = 1$ is called *binary.* If $\mathcal{G} = \emptyset$ then (1.1) is called *linear program* or *LP*. If $\mathcal{C} = \emptyset$ then (1.1) is called *integer program* or *IP*. A 0-1 MIP is a MIP where all the integer variables are binary. A vector x that satisfies all constraints is called a *feasible solution.* An *optimal solution* is a feasible solution for which the objective function achieves the smallest value.

From a complexity point of view mixed integer programming problems belong to the class of $\mathcal{NP}$-hard problems (see Garey and Johnson [26], for example) which makes it unlikely that efficient, i.e., polynomial time, algorithms for their solution exist.

The *linear programming (LP) relaxation* of a MIP is the problem obtained from (1.1) by dropping the integrality restrictions, i.e., replacing $x \in \mathbb{Z}^{\mathcal{G}} \times \mathbb{R}^{\mathcal{C}}$ with $x \in \mathbb{R}^{\mathcal{G}\cup\mathcal{C}}$. The optimal value of the LP relaxation provides a lower bound on the optimal value of the MIP. Therefore, if an optimal solution to the LP relaxation satisfies the integrality restrictions, then that solution is also optimal for the MIP. If the LP relaxation is infeasible, then the MIP is also infeasible, and if the LP relaxation is unbounded, then the MIP is either unbounded or infeasible. If the MIP is feasible and its LP relaxation is bounded, then the MIP has optimal solution(s)[1].

A common approach to solving a MIP consists of solving its LP relaxation[2] in the hope of finding an optimal solution x^* which happens to be integer. If this is not the case, there are two main ways to proceed. In the *cutting-plane* approach, one enters the *separation phase*, where a linear inequality (*cut*) $\alpha^T x \leq \alpha_0$ is identified which separates x^* from the feasible solutions of the MIP. The cut is appended to the current LP relaxation, and the procedure is iterated. In the *branch-and-bound* approach, instead, the MIP is replaced by two subproblems obtained, e.g., by imposing an additional restriction of the type $x_j \leq \lfloor x_j^* \rfloor$ and $x_j \geq \lceil x_j^* \rceil$, respectively, where x_j is an integer-constrained variable with fractional value in x^*. The procedure is then recursively applied to each of the subproblems.

1.3 Preprocessing

Preprocessing is the name for a number of techniques, employed by MIP solvers, aimed at reducing the size of an instance and strengthen its LP bound [71]. Preprocessing can alter a given formulation quite significantly by fixing, aggregating,

[1]This is in general not true when the problem is not rational. Consider for example the problem $\max\{x_1 - \sqrt{2}x_2 : x_1 \leq \sqrt{2}x_2,\ x_1,\ x_2 \text{ integer}\}$; this has feasible solutions with negative objective values arbitrarily close to 0, but none equal to 0.

[2]Linear programs can efficiently be solved using Dantzig's simplex algorithm or interior point methods. For an introduction to linear programming see, for instance, Nemhauser and Wolsey [59], or Bertsimas and Tsitsiklis [14].

and/or substituting variables and constraints of the problem, as well as changing the coefficients of the constraints and the objective function.

As an example (taken from Martin [55]), consider the following *bounds strengthening* technique, where we exploit the bounds on the variables to detect so-called forcing and dominated rows. Given some row i, let

$$\begin{aligned} L_i &= \sum_{j \in P_i} a_{ij} \underline{x}_j + \sum_{j \in N_i} a_{ij} \overline{x}_j, \\ U_i &= \sum_{j \in P_i} a_{ij} \overline{x}_j + \sum_{j \in N_i} a_{ij} \underline{x}_j \end{aligned} \tag{1.2}$$

where $P_i = \{j : a_{ij} > 0\}$ and $N_i = \{j : a_{ij} < 0\}$. Obviously, $L_i \leq \sum_{j=1}^{n} a_{ij} x_j \leq U_i$. The following cases might come up. An inequality i is an *infeasible row* if $L_i > b_i$. In this case the entire problem is infeasible. An inequality i is a *forcing row* if $L_i = b_i$. In this case all variables in P_i can be fixed to their lower bound and all variables in N_i to their upper bound. Row i can be deleted afterwards. An inequality i is a *redundant row* if $U_i < b_i$. In this case i can be removed.

This row bound analysis can also be used to strengthen the lower and upper bounds of the variables. Compute for each variable x_j and each inequality i

$$\begin{aligned} u_{ij} &= \begin{cases} (b_i - L_i)/a_{ij} + \underline{x}_j, & \text{if } a_{ij} > 0 \\ (L_i - U_i)/a_{ij} + \underline{x}_j, & \text{if } a_{ij} < 0 \end{cases} \\ l_{ij} &= \begin{cases} (L_i - U_i)/a_{ij} + \overline{x}_j, & \text{if } a_{ij} > 0 \\ (b_i - L_i)/a_{ij} + \overline{x}_j, & \text{if } a_{ij} < 0. \end{cases} \end{aligned} \tag{1.3}$$

Let $u_j = \min_i u_{ij}$ and $l_j = \max_i l_{ij}$. If $u_j \leq \overline{x}_j$ and $l_j \geq \underline{x}_j$ we speak of an *implied free variable*. The simplex method might benefit from not updating the bounds but treating variable x_j as a free variable (note that setting the bounds of x_j to $-\infty$ and $+\infty$ will not change the feasible region). Free variables will commonly be in the basis and are thus useful in finding a starting basis. For mixed integer programs however, if the variable x_j is integer, it is better in general to update the bounds by setting $\overline{x}_j = \min\{\overline{x}_j, u_j\}$ and $\underline{x}_j = \max\{\underline{x}_j, l_j\}$, because the search region of the variable within an enumeration scheme is reduced. In case x_j is an integer (or binary) variable we round u_j down to the next integer and l_j up to the next integer. For example consider the following inequality:

$$-45x_6 - 45x_{30} - 79x_{54} - 53x_{78} - 53x_{102} - 670x_{126} \leq -443 \tag{1.4}$$

Since all variables are binary we get $L_i = -945$ and $U_i = 0$. For $j = 126$ we obtain $l_{ij} = (-443 + 945)/-670 + 1 = 0.26$. After rounding up it follows that x_{126} must be 1. Note that with these new lower and upper bounds on the variables it might pay to recompute the row bounds L_i and U_i, which again might result in tighter bounds on the variables.

Techniques that are based on checking infeasibility are called primal reduction techniques. Dual reduction techniques make use of the objective function and attempt to fix variables to values that they will take in any optimal solution.

One preprocessing technique that may have a big impact in strengthening the LP relaxation of a MIP formulation is *coefficient improvement*. This technique updates the coefficients of the formulation so that the constraints define a smaller LP relaxation, hence leading to improved LP bounds (see Nemhauser and Wolsey [59], for example).

Probing is another technique that considers the implications of fixing a variable to one of its bounds. For instance, if fixing a binary variable x_1 to one, forces a reduction in the upper bound of x_2, then one can use this information in all constraints in which x_2 appears and possibly detect further redundancies, bound reductions, and coefficient improvements.

See, for example, Savelsbergh [71] or Martin [55] for a survey of these and other preprocessing techniques.

Finally, all these techniques can be applied not only before solving the initial formulation at the root node, but also before each sub-problem in the branching tree. However, since the impact of node preprocessing is limited to only the subtree defined by the node, one should take into consideration whether the time spent on node preprocessing is worthwhile.

1.4 Valid Inequalities and Cutting Planes

A *valid inequality* for a MIP is an inequality that is satisfied by all feasible solutions. A *cutting plane*, or simply *cut*, is a valid inequality that is not satisfied by all feasible points of the LP relaxation. Thus, if we find a cut, we can add it to the formulation and strengthen the LP relaxation.

In the late 50's, Gomory [30] pioneered the cutting-plane approach, proposing a very elegant and simple way to derive cuts for an IP by using information associated with an optimal LP basis. This was later generalized by Chvátal [16] (see also Wolsey [76], for instance).

We can construct a valid inequality for the set $X := P \cap \mathbb{Z}^n$, where $P := \{x \in \mathbb{R}^n_+ : Ax \leq b\}$, and A is an $m \times n$ matrix, as follows. Let u be an arbitrary vector in $\mathbb{R}^m_+$. Then the inequality

$$u^T Ax \leq u^T b \tag{1.5}$$

is a valid inequality for P. Since $x \geq 0$ in P, we can round down the coefficients of the left-hand side of (1.5) and obtain that the inequality

$$\lfloor u^T A \rfloor x \leq u^T b. \tag{1.6}$$

Finally, as x is integer in X, we can also round down the right-hand side and get

$$\lfloor u^T A \rfloor x \leq \lfloor u^T b \rfloor \tag{1.7}$$

as a valid inequality for X.

```
Generic cutting plane algorithm:
  1.   repeat
  2.     solve the current LP relaxation
  3.     if the optimal solution $x^*$ is integer feasible then
  4.       return $x^*$
  5.     else
  6.       find a cutting plane $(\pi, \pi_0)$ violated by $x^*$
  7.       add the cut $\pi^T x \leq \pi_0$ to the current formulation
  8.     endif
  9.   until stopping criterion reached
```

Figure 1.1: Generic cutting plane algorithm

Quite surprisingly, all valid inequalities for an integer program can be generated by applying repeatedly this procedure a finite number of times (for a proof see Chvátal[16], Schrijver [72], or Wolsey [76]).

In principle MIPs can be solved to optimality using the cutting plane algorithm of Figure 1.1, however practical experience with Gomory's algorithm shows that the quality of the cuts generated becomes rather poor after a few iterations, which causes the so called *tailing-off* phenomenon: a long sequence of iterations without significant improvements towards integrality. Adding too many cutting planes also leads to numerical problems, thus a stopping criterion is used to terminate the generation of cuts, even if a feasible solution is not reached.

The first successful application of a cutting plane algorithm is due to Dantzig, Fulkerson, and Johnson [20] who used it to solve large (for that time) instances of the traveling salesman problem.

The cutting planes implemented in MIP solvers can be classified into two broad categories. The first are general cuts that are valid for any MIP problem; these include Gomory mixed-integer and mixed-integer rounding cuts [54, 60]. The second category includes strong polyhedral cuts from knapsack [6, 11, 36, 39, 63], fixed-charge flow [37, 66] and path [74], and vertex packing [62] relaxations of MIPs. Strong inequalities for these simpler substructures are usually quite effective in improving LP relaxations of more complicated sets. MIP solvers automatically identify such substructures by analyzing the constraints of the formulation and try to add appropriate cuts.

1.5 Branch-and-Bound

Branch-and-bound algorithms for mixed integer programming use a "divide and conquer" strategy to explore the set of all feasible mixed integer solutions. These algorithms build a search tree, in which the nodes of the tree represent subproblems defined over subsets of the feasible region. According to Wolsey [76] the first paper presenting a branch-and-bound strategy for the solution of integer programs is due to Land and Doig [47].

Let P_0 be a mixed-integer programming problem of the form (1.1). Let $X_0 := \{x \in \mathbb{Z}^p \times \mathbb{Q}^{n-p} : Ax \leq b, \underline{x} \leq x \leq \overline{x}\}$ be the set of feasible mixed integer solutions of problem P_0. If it is too difficult to compute

$$\begin{array}{rll} z_{\text{MIP}} = & \min & c^T x \\ & \text{subject to} & x \in X_0, \end{array} \tag{1.8}$$

then we can split X_0 into a finite number of disjoint subsets $X_1, \ldots, X_k \subset X$, such that $\cup_{j=1}^k X_j = X_0$, and try to solve separately each of the subproblems

$$\begin{array}{ll} \min & c^T x \\ \text{subject to} & x \in X_j, \quad \forall\, j = 1, \ldots, k. \end{array} \tag{1.9}$$

Afterwards we compare the optimal solutions of the subproblems and choose the best one. Since each subproblem is usually only slightly easier than the original problem, this idea is iterated recursively splitting the subproblems again into further subproblems. The (fast-growing) list of all subproblems is usually organized as a tree, called a *branch-and-bound tree* and we say that a *father* or *parent* problem is split into two or more *son* or *child* problems. This is the branching part of the branch-and-bound method.

For the bounding part of this method we assume that we can efficiently compute a lower bound $z^*(P)$ of each subproblem P (with feasibility set X), so that $z^*(P) \leq \min_{x \in X} c^T x$. In the case of mixed integer programming this lower bound can be obtained by using the *LP relaxation.*

During the exploration of the search tree we can find that the optimal solution x^* of the LP relaxation of a subproblem P is also a feasible mixed integer point, i.e., $x^* \in X$. When x^* is not feasible, it is sometimes possible to obtain a feasible point by rounding the integer variables, or using more advanced heuristics. The feasible solution with the smallest objective value, z^{best}, found so far is called the *incumbent solution.* This allows us to maintain an upper bound on the optimal solution value z_{MIP} of P_0, as $z_{\text{MIP}} \leq z^{best}$. Having a good upper bound is crucial in a branch-and-bound algorithm, because it keeps the branching tree small. In fact, suppose the solution of the LP relaxation of some other subproblem P' satisfies $z^*(P') \geq z^{best}$, then the subproblem P' can be pruned, without further processing, because the optimal solution of this subproblem cannot be better than the incumbent one.

The algorithm of Figure 1.2 summarizes the whole branch-and-bound procedure for mixed-integer programs.

Initially the active node list L contains only the root node. Then, whenever a node is *processed*, its LP relaxation is solved and the node is either pruned or split into sub-problems which are added back to the list. Therefore, at any given time, the nodes in the list L correspond to the unsolved problems in the branch-and-bound tree, which are the leaves of the tree.

```
Generic branch-and-bound algorithm:
  1.   let $L := \{P_0\}$
  2.   let $z^{best} := +\infty$
  3.   repeat
  4.     select a problem $P$ from $L$
  5.     remove $P$ from $L$
  6.     solve the LP relaxation of $P$
  7.     if LP feasible then
  8.       let $x^*$ be an optimal solution of $P$
  9.       let $z^*(P) := c^T x^*$
 10.       if $x^*$ feasible (for $P_0$) then
 11.         if $z^{best} > z^{opt}(P)$ then
 12.           let $z^{best} := z^*(P)$
 13.           let $\tilde{x} := x^*$
 14.           delete from $L$ all subproblems $P$ with $z^*(P) \geq z^{best}$
 15.         end if
 16.       else
 17.         split problem $P$ into subproblems and add them to $L$
 18.       endif
 19.     endif
 20.   until $L = \emptyset$
 21.   return $z^{best}$ and $\tilde{x}$
```

Figure 1.2: Generic branch-and-bound algorithm

Within this general framework, there are two aspects that requires a choice to be taken. The first one is how to perform the *branching* at step 17, that is how split a problem P into subproblem. And the second one is how to choose which problem to process next at step 4 (*node selection*).

1.5.1 Branching

A natural way to divide the feasible region of a MIP problem is to choose a variable x_i that is fractional in the current linear programming solution x^* and create two subproblems, one with the updated bound $x_i \leq \lfloor x_i^* \rfloor$ and the other with $x_i \geq \lceil x_i^* \rceil$. This type of branching is referred to as *variable dichotomy.*

In general there are several fractional variables to choose from. Since the effectiveness of the branch-and-bound algorithm depends heavily on the convergence of upper and lower bounds, we would like to choose the variable that leads to the highest bound improvement. However this has been proved to be "difficult" so what is typically done is select a list of "candidate" branching variables, among those that are most fractional, and then, for each of these, estimate the LP bound that a branching on that variable would lead to. One of the methods used to estimate the bound improvement consists in performing a small number of pivots and observe what happens to the objective function (*strong branching*).

Chapter 4 presents a more general approach where branching is performed on general disjunctions rather than on individual variables.

1.5.2 Node selection

There are two main possible strategies to visit the branching tree. In the *best-first* search, the node P with the lowest $z^*(P)$ is chosen, since it is supposed to be closer to the optimal solution. At the other extreme is the *depth-first* search: nodes are ordered according to their depth in the branching tree, and the deepest pending node is processed first.

For a fixed branching rule, best-first search minimizes the number of nodes evaluated before completing the search. However, there are two main drawbacks: one is that the search tends to stay in the higher levels of the branching tree, where problems are less constrained and, thus, hardly lead to improvements of the incumbent solution. The second one is that the search tree tends to be explored in a breadth-first fashion, so subsequent linear programs have little relation to each other, leading to longer evaluation times. One way to mitigate this second issue could be to save the basis information at all the nodes, but in this case the memory requirements for searching the tree in a best-first manner might become prohibitive.

Depth-first is easier to implement, has lower memory requirement, and changes in the linear program, from one node to the next, are minimal (only a variable bound). It also usually finds feasible solutions more quickly than with best-first search as feasible solutions are typically found deep in the search tree. One disadvantage is that, when a "bad" branch (i.e., not containing good feasible solutions) is visited, this strategy searches exhaustively the whole sub-tree before backtracking to different areas. Also, it can spend a lot of time solving nodes that could have been pruned if a better incumbent had been known.

Most integer programming solvers employ a hybrid of best-first search and depth-first search, trying to benefit from the strengths of both, regularly switching between the two strategies during the search. In the beginning the emphasis is usually more on depth-first, to find high quality solutions quickly, whereas in the later stages of the search, the emphasis is usually more on best-first, to improve the lower bounds.

1.6 Branch-and-Cut

Combination of cutting-plane and branch-and-bound techniques was attempted since the early 70's. Initially, however, the constraint generators were used only at the root node, as a simple preprocessor, to obtain a tighter LP relaxation of the original MIP formulation. In the mid 80's, Padberg and Rinaldi [64, 65] introduced a new methodology for an effective integration of the two techniques, which they named *branch-and-cut*. This is an overall solution scheme whose main ingredients include: the generation at every node of the branching tree of (facet-defining) cuts globally valid along the tree; efficient cut management by means of a constraint pool structure; column/row insertion and deletion from the current LP; variable fixing and setting; and the treatment of inconsistent LP's.

Branch-and-cut has a number of advantages over pure cutting-plane and branch-and-bound schemes. With respect to the branch-and-bound approach, the addition of new cuts improves the LP relaxation at every branching node. With respect to the pure cutting-plane technique, one can resort to branching as soon as tailing-off is detected. As the overall convergence is ensured by branching, the cut separation can be of heuristic type, and/or can restrict to subfamilies of problem-specific cuts which capture some structures of the problem in hand. Moreover, the run-time variable pricing and cut generation/storing mechanisms allow one to deal effectively with tight LP relaxations having in principle a huge number of variables and constraints.

While a branch-and-cut algorithm spends more time in solving the LP relaxations, the resulting improved LP bounds usually leads to a significantly smaller search tree. Naturally, as with all techniques designed to improve the performance of the basic branch-and-bound algorithm, the time spent on cut generation must be contained in order not to outweigh the speed-up due to improved LP bounds.

For more information about branch-and-bound and branch-and-cut see also Nemhauser and Wolsey [59], for instance.

1.7 Primal Heuristics

In order to reduce the size of the branching tree, it is very useful to find good incumbent solutions as soon as possible. On the other hand, waiting to find feasible solutions at a node just by solving its LP relaxations can take a very long time. Therefore MIP solvers attempt to find feasible solutions early in the search tree by means of simple and quick heuristics. As an extreme example, if the optimal solution would be known at the root node, then branch-and-bound would be used only to prove the optimality of the solution. In this case, for a fixed branching rule, any node selection rule would produce the same tree with the minimum number of nodes.

While cutting planes (and, to some extent, bounds) are used to strengthen the lower bound on the optimal solution, primal heuristics have the complementary role of improving the upper bound, given by incumbent solutions, and help close the gap between the two.

Finding a feasible solution of a given MIP model is, however, a very important ($\mathcal{NP}$-complete) problem that can be extremely hard in practice. Several techniques are typically used, involving simple rounding, partial enumeration, diving (into the branching-tree) and variable fixing. Once one feasible solution has been found, other *improvement algorithms* can be used to iteratively try to obtain a better solution. Neighborhood search algorithms (alternatively called local search algorithms) are a wide class of improvement algorithms where at each iteration an improving solution is found by searching the “neighborhood” of the current solution.

Since these procedures can be quite time consuming, it seems reasonable to spend more effort on finding good feasible solutions early in the search tree, since this would have the most impact on the solution process. Very recently, Fischetti, Glover and Lodi proposed a heuristic scheme for finding a feasible solution to 0-1 MIPs, called *Feasibility Pump* (FP). In this Chapter 2, this technique is further elaborated and extended in two main directions, namely (i) handling as effectively as possible MIP problems with both binary and general-integer variables, and (ii) exploiting the FP information to drive a subsequent enumeration phase.

1.8 Truncated search

While mixed-integer linear programming plays a central role in modeling difficult-to-solve ($\mathcal{NP}$-hard) combinatorial problems of practical interest, the exact solution of the resulting models often cannot be carried out (in a reasonable time) for the problem sizes of interest in real-world applications, hence one is interested in effective heuristic methods.

Although several heuristics have been proposed in the literature for specific classes of problems, only a few papers deal with general-purpose MIP heuristics, including [8, 9, 27, 28, 29, 42, 52, 53, 58] among others.

Exact MIP solvers are nowadays very sophisticated tools designed to hopefully deliver, within acceptable computing time, a provable optimal solution of the input MIP model, or at least a heuristic solution with a practically-acceptable error. In fact, what matters in many practical cases is the possibility of finding reasonable solutions as early as possible during the computation.

In this respect, the “heuristic behavior” of the MIP solver plays a very important role: an aggressive solution strategy that improves the incumbent solution at very early stages of the computation is strongly preferred to a strategy designed for finding good solutions only at the late steps of the computation (that, for difficult problems, will unlikely be reached within the time limit).

Many commercial MIP solvers allow the user to have a certain control on their heuristic behavior through a set of parameters affecting the visit of the branching tree, the frequency of application of the internal heuristics, the fact of emphasizing the solution integrality rather than its optimality, etc. Some recently proposed techniques (Local Branching [24] and RINS [19]), have provided a considerable improvement in this direction enhancing the heuristic behavior of MIP solvers through appropriate diversification mechanisms borrowed from local search paradigms.

Therefore, even if branch-and-cut algorithms are $\mathcal{NP}$-hard, it may be reasonable to model a problem as a MIP instance and search for “heuristic” solutions by means of a black-box general purpose MIP solver with a truncated search - thus exploiting the level of sophistication reached nowadays by these tools.

This same idea is also used, for example, in Chapter 2, when resorting to enumeration for finding a feasible solution, and in Chapter 4 for searching good branching disjunctions.

Chapter 2

Feasibility Pump

2.1 Introduction

In this chapter we address the problem of finding heuristic solutions of a generic MIP problem of the form

$$(MIP) \quad \min c^T x \tag{2.1}$$
$$Ax \geq b \tag{2.2}$$
$$x_j \text{ integer} \quad \forall j \in \mathcal{I} \tag{2.3}$$

where A is an $m \times n$ matrix and $\mathcal{I}$ is the nonempty index-set of the integer variables. We assume without loss of generality that the MIP constraints $Ax \geq b$ include the variable bounds

$$l_j \leq x_j \leq u_j \quad \forall j \in \mathcal{N}$$

(possibly $l_j = -\infty$ and/or $u_j = +\infty$ for some j), where $\mathcal{N}$ denotes the set of all (continuous and integer) variables.

Finding any feasible MIP solution is an $\mathcal{NP}$-complete problem that can be extremely hard in practice. As a matter of fact, state-of-the-art MIP solvers may spend a very large computational effort before initializing their incumbent solution. Therefore, heuristic methods aimed at finding (and then refining) any feasible solution for hard MIPs are very important in practice; see [8], [9], [27], [28], [29], [40], [42], [52], [53], [58], [24], [19], and [10] among others.

Very recently, Fischetti, Glover and Lodi [23] proposed a heuristic scheme for finding a feasible solution to general MIPs, called *Feasibility Pump* (`FP`), that works as follows. Let $P := \{x : Ax \geq b\}$ denote the polyhedron associated with the LP relaxation of the given MIP. With a little abuse of notation, we say that a point x is integer if x_j is integer $\forall j \in \mathcal{I}$ (no matter the value of the other components). Analogously, the rounding $\tilde{x}$ of a given x is obtained by setting $\tilde{x}_j := [x_j]$ if $j \in \mathcal{I}$ and $\tilde{x}_j := x_j$ otherwise, where $[\cdot]$ represents scalar rounding to the nearest integer. The (L_1-norm) distance between a generic point $x \in P$ and a given integer $\tilde{x}$ is defined as

$$\Delta(x, \tilde{x}) = \sum_{j \in \mathcal{I}} |x_j - \tilde{x}_j|$$

Notice that $\tilde{x}$ is assumed to be integer; moreover, the continuous variables x_j with $j \notin \mathcal{I}$, if any, do not contribute to the distance function $\Delta(x, \tilde{x})$. For any given integer $\tilde{x}$, the distance function can be written as[1]:

$$\Delta(x, \tilde{x}) := \sum_{j \in \mathcal{I}: \tilde{x}_j = l_j} (x_j - l_j) + \sum_{j \in \mathcal{I}: \tilde{x}_j = u_j} (u_j - x_j) + \sum_{j \in \mathcal{I}: l_j < \tilde{x}_j < u_j} d_j \tag{2.4}$$

where variables $d_j (= |x_j - \tilde{x}_j|)$ satisfy constraints

$$d_j \geq x_j - \tilde{x}_j \quad \text{and} \quad d_j \geq \tilde{x}_j - x_j \quad \forall j \in \mathcal{I} : l_j < \tilde{x}_j < u_j \tag{2.5}$$

It then follows that the closest point $x^* \in P$ to $\tilde{x}$ can easily be determined by solving the LP

$$\min\{\Delta(x, \tilde{x}) : \widetilde{A}x \geq \widetilde{b}\} \tag{2.6}$$

where $\widetilde{A}x \geq \widetilde{b}$ is the original system $Ax \geq b$ possibly amended by constraints (2.5). If $\Delta(x^*, \tilde{x}) = 0$, then $x^*_j \, (= \tilde{x}_j)$ is integer $\forall j \in \mathcal{I}$, so x^* is a feasible MIP solution. Conversely, given a point $x^* \in P$, the integer point $\tilde{x}$ closest to x^* is easily determined by just rounding x^*. The `FP` heuristic works with a pair of points $(x^*, \tilde{x})$ with $x^* \in P$ and $\tilde{x}$ integer, that are iteratively updated with the aim of reducing as much as possible their distance $\Delta(x^*, \tilde{x})$. To be more specific, one starts with any $x^* \in P$, and initializes a (typically infeasible) integer $\tilde{x}$ as the rounding of x^*. At each FP iteration, called *pumping cycle*, $\tilde{x}$ is fixed and one finds through linear programming the point $x^* \in P$ which is as close as possible to $\tilde{x}$. If $\Delta(x^*, \tilde{x}) = 0$, then x^* is a MIP feasible solution, and the heuristic stops. Otherwise, $\tilde{x}$ is replaced by the rounding of x^* so as to further reduce $\Delta(x^*, \tilde{x})$, and the process is iterated.

The `FP` scheme (as stated) tends to stop prematurely due to stalling issues. This happens whenever $\Delta(x^*, \tilde{x}) > 0$ is not reduced when replacing $\tilde{x}$ by the rounding of x^*, meaning that all the integer-constrained components of $\tilde{x}$ would stay unchanged in this iteration. In this situation, a few components $\tilde{x}_j$ are heuristically chosen and modified, even if this operation increases the current value of $\Delta(x^*, \tilde{x})$. The reader is referred to [23] for a detailed description of this (and related) anti-stalling mechanisms.

According to the computational analysis reported in [23], `FP` is quite effective in finding feasible solutions of hard 0-1 MIPs. However, as observed in the conclusions of that paper, MIPs with general-integer variables are much more difficult to solve by using the `FP` approach. This can be explained by observing that, for

[1]This expression is slightly different from the one proposed in [23]; both definitions assume an objective function that tends to minimize the value of the d_j variables.

a general integer variable, one has to decide not just the rounding direction (up or down), as for binary variables, but also the new value of the variable; e.g., if a variable x_j is between 0 and 10 and takes value 6.7 (say) in the LP relaxation, the decision of "moving up" its value still leaves four values (7, 8, 9, and 10) to choose from. The same difficulty arises in case of stalling: in the binary case, one only needs to choose the variables to flip (from 0 to 1 or viceversa), whereas for general integer variables one also has to decide their new value.

In this chapter we build on the ideas presented in [23] for 0-1 MIPs and extend them in two main directions. The first one is to handle effectively MIP problems with both binary and general integer variables. The second is to exploit the information obtained from the feasibility pump to drive an enumeration stage.

The chapter is organized as follows. In Section 2.2 we propose an `FP` extension to deal with MIPs with general-integer variables. Computational results are presented in Section 2.3, where we compare the `FP` performance with that of the commercial solvers `Xpress Optimizer` 16.01.05 and `ILOG Cplex` 9.1 on a set of hard general MIPs taken from MIPLIB 2003 library [5] and other sources. Section 2.4 considers the important issue of improving the quality of the first solution found by `FP`. Finally, we draw some conclusions in Section 2.5.

2.2 The Feasibility Pump for general MIPs

The basic scheme of our `FP` implementation for general MIPs is illustrated in Figure 2.1. As already stated, the method generates two (hopefully convergent) trajectories of points x^* and $\tilde{x}$ that satisfy feasibility in a complementary but partial way—one satisfies the linear constraints, the other the integer requirement. The current pair $(x^*, \tilde{x})$ is initialized at steps 1-2. The while-do loop at step 4 is executed until the distance $\Delta(x^*, \tilde{x})$ becomes zero (in which case, x^* is a feasible MIP solution), or the current iteration counter `nIter` reaches a given limit (`maxIter`). At each pumping cycle, at step 6 we fix $\tilde{x}$ and re-define x^* as the closest point in P, so as to hopefully reduce the current distance $\Delta(x^*, \tilde{x})$. At step 7 we check whether $\Delta(x^*, \tilde{x}) = 0$, in which case x^* is feasible and we stop. Otherwise, at step 9 we replace $\tilde{x}$ by $[x^*]$ (the rounding of x^*), and repeat. In case the components of the new $\tilde{x}$ indexed by $\mathcal{I}$ would coincide with the previous ones, however, a more involved computation is needed to avoid entering a loop. We first compute, at step 11, a score $\sigma_j = |x^*_j - \tilde{x}_j|$, $j \in \mathcal{I}$, giving the likelihood of component $\tilde{x}_j$ to *move*, i.e., to change its current value from $\tilde{x}_j$ to $\tilde{x}_j + 1$ (if $x^*_j > \tilde{x}_j$) or to $\tilde{x}_j - 1$ (if $x^*_j < \tilde{x}_j$). Then, at step 12 we update $\tilde{x}$ by performing the `TT` (say) moves with largest score, where `TT` is generated as a uniformly-random integer in range (`T`/2, 3`T`/2), and `T` is a given parameter.

In order to avoid cycling, at step 13 we check whether one of the following situations occurs:

- the current point $\tilde{x}$ is equal (in its components indexed by $\mathcal{I}$) to the one found in a previous iteration;

```
The Feasibility Pump for general MIPs (basic scheme):
   1. initialize $x^* := \mathrm{argmin}\{c^T x : Ax \geq b\}$
   2. $\tilde{x} := [x^*]$ (:= rounding of $x^*$)
   3. nIter := 0
   4. while ($\Delta(x^*, \tilde{x}) > 0$ and nIter < maxIter) do
   5.   nIter := nIter+1
   6.   $x^* := \mathrm{argmin}\{\Delta(x, \tilde{x}) : \widetilde{A}x \geq \widetilde{b}\}$
   7.   if $\Delta(x^*, \tilde{x}) > 0$ then
   8.     if $[x^*_j] \neq \tilde{x}_j$ for at least one $j \in \mathcal{I}$ then
   9.       update $\tilde{x} := [x^*]$
  10.     else
  11.       for each $j \in \mathcal{I}$ define the score $\sigma_j := |x^*_j - \tilde{x}_j|$
  12.       move the TT=rand(T/2, 3T/2) components $\tilde{x}_j$ with largest $\sigma_j$
  13.       if cycling is detected, perform a restart operation
  14.     endif
  15.   endif
  16. enddo
```

Figure 2.1: The basic FP scheme for general-integer MIPs

- distance $\Delta(x^*, \tilde{x})$ did not decrease by at least 10% in the last KK (say) iterations.

If this is the case, we perform a *restart* operation (to be detailed later), consisting in a random perturbation of some entries of $\tilde{x}$.

As a further step to reduce the likelihood of cycling, we found it useful to also perturb the rounding function used at step 2 of Figure 2.1. Indeed, the rounded components are typically computed as $[x_j] := \lfloor x_j + \tau \rfloor$ with τ fixed at 0.5. However, in our tests we obtained better results by taking a random τ defined as follows:

$$\tau(\omega) := \begin{cases} 2\omega(1-\omega) & \text{if } \omega \leq \frac{1}{2} \\ 1 - 2\omega(1-\omega) & \text{if } \omega > \frac{1}{2} \end{cases}$$

where ω is a uniform random variable in $[0, 1)$. Using the definition above, threshold τ can take any value between 0 and 1, but values close to 0.5 are more likely than those near 0 or 1; see Figure 2.2 for an illustration of the probability distribution and density for $\tau(\omega)$.

2.2.1 Binary and general-integer stages

Difficult MIPs often involve both binary and general-integer variables playing a quite different role in the model. A commonly-used rule in MIP solvers is to branch first on binary variables, then on general integers. This corresponds to the "smallest domain first" rule in constraint programming: branching on a variable with a large domain (e.g., a general integer variable) will not enforce

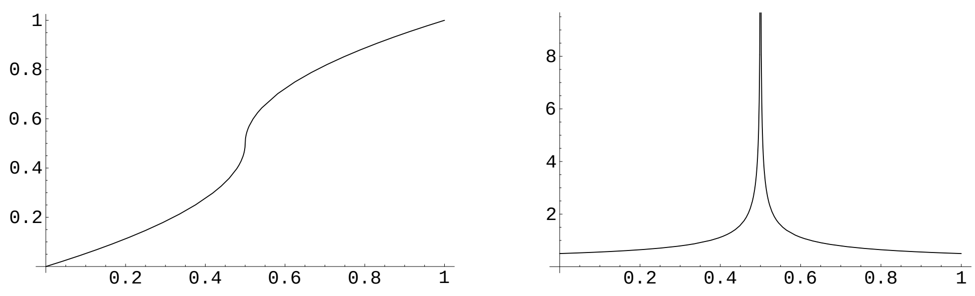


Figure 2.2: Probability distribution and density of the rounding threshold

as powerful constraints as branching on a variable with a small domain (e.g., a binary variable), therefore it is postponed to the bottom of the tree. Following this approach, we found useful to split the FP execution in two stages.

At Stage 1 (binary stage), we concentrate on the *binary* variables x_j with $j \in \mathcal{B}$ (say), defined as the integer variables x_j with $u_j - l_j = 1$, and relax the integrality condition on all other x_j, $j \in \mathcal{I} \setminus \mathcal{B}$. This produces an easier MIP, with a distance function $\Delta(x, \tilde{x})$ that does not involve any additional variable d_j. The purpose of the binary stage is to reach as soon as possible a solution that is feasible with respect to the binary variables, with the hope that the general-integer ones are also "almost integer" and only a few of them will require the introduction of the additional variables d_j and of the associated constraints (2.5).

At Stage 2, instead, the integrality condition on all (binary and non-binary) variables x_j, $j \in \mathcal{I}$, is restored, and the FP method continues by taking into account all the integrality requirements (this requires the introduction, at each iteration, of the additional variables d_j needed to express the distance function with respect to the current point $\tilde{x}$).

During Stage 1, the restart operation at step 13 is only performed in case of cycling (i.e., when the same $\tilde{x}$ is found in different iterations), and consists in a random flip of the binary variables that did not change in the last iteration, with probability $|x_j^* - [x_j^*]| + \rho$, where $\rho = 0.03$ is an hardwired parameter.

The algorithm exits Stage 1 and moves to Stage 2 when: (a) a "feasible" (with respect to the binary variables only) solution x^* has been found, or (b) the incumbent minimum $\Delta(x^*, \tilde{x})$ has not been updated in the last KK $= 70$ iterations, or (c) an iteration limit has been reached. The point $\tilde{x}$ that produced the smallest $\Delta(x^*, \tilde{x})$ during Stage 1 is stored and passed to Stage 2 as the initial $\tilde{x}$.

2.2.2 Enumeration stage

For some difficult instances, FP (as stated) turns out to be unable to find a feasible solution within acceptable computing time. In this case, instead of insisting with the classical FP scheme one can think of resorting to a sort of "enumeration stage" based on the information provided by the previous FP execution. Following this idea, we have implemented the following simple scheme.

Let x^B (B for best) be the LP point x^* computed at step 6 of the algorithm of Figure 2.1 which is as close as possible to its rounding $\tilde{x}$. Even in case x^B is not feasible, we typically have that the infeasibility measure $\Delta(x^B, \tilde{x})$ is small. Therefore it seems reasonable to concentrate on $\tilde{x}$ and use a (possibly heuristic) enumerative MIP method in the attempt of finding a feasible integer solution which is close to $\tilde{x}$. In our implementation, this is obtained by applying a general-purpose (truncated) MIP solver to the original problem (2.1), after having replaced the original objective function $c^T x$ by the distance function (2.4), where $\tilde{x} := [x^B]$ is the "almost feasible" solution available after Stage 2. The idea here is to exploit the full power of the MIP solver, but with an objective function that penalizes the solutions that are far from the available "almost feasible" `FP` solution $\tilde{x}$.

As the enumeration phase above is applied at the end of Step 2, it will be referred to as the Stage 3 of the overall `FP` method.

2.3 Computational experiments

In this section we report computational results comparing the performance of the proposed `FP` method with that of the two state-of-the-art commercial solvers, namely, `Xpress Optimizer` 16.01.05 [18] and `ILOG Cplex` 9.1 [43]. Of course, the heuristic performance of a MIP solver depends heavily on the branching rule (as discussed, e.g., in [15]), on the tree-exploration strategy [19], and on the tuning of specific parameters of the MIP solver at hand. In our experiments, however, we decided to use as much as possible the default parameter values of the codes under comparison. To be specific, in our `FP` implementation we used the following parameters:

- iteration limit set to 10000 and 2000 for Stage 1 and 2, respectively;
- parameter `TT` set to 20;
- parameter `KK` set to 70 for stage 1 and 600 for Stage 2;
- in the perturbation phase of steps 11-12 of Figure 2.1, we consider only variables with fractional value (defined as $|x_j - [x_j]|$) greater than 0.02, and always leave the other variables unmodified.

In order to have a fair comparison, the LP/MIP functions used within `FP` are the same used by the method under comparison. To be more specific, we ran `Xpress Optimizer` against an `FP` implementation based on `Xpress Optimizer` procedures (called `FP-xpress` in the sequel), and `ILOG Cplex` against an `FP` implementation based on `ILOG Cplex` procedures (called `FP-cplex` in the sequel). Within our `FP` code, we used the `ILOG Cplex` function `CPXoptimize` and `Xpress Optimizer` function `XPRSminim` to solve the initial LP (thus leaving to the solver the choice of the actual LP algorithm to invoke), with the default parameter setting except for disabling the presolver. For the subsequent LPs with the modified objective function, we forced the use of the primal simplex algorithm. The reason for this choice is that, from iteration to iteration, the feasibility of the current basis is always preserved during stage 1 and, quite often, also during stage 2 (according to our computational tests, using the primal simplex yields to better computing times on 31 out of 35 instances and, on average, to a 25% performance improvement). Finally, within the enumeration Stage 3 we set the `ILOG Cplex` parameter `MIP emphasis` to 4 (i.e., *Emphasize hidden feasible solutions*, so as to activate the RINS heuristic [19]), in order to bias the search towards feasibility rather than optimality. All other solver parameters were left at their default values.

The MIP solvers compared against `FP` have been run in their default settings (with presolver enabled), except the `ILOG Cplex` parameter `MIP mphasis` (set to 1, i.e., *Emphasize integer feasibility*) and the `Xpress Optimizer` parameter `XPRScutstrategy` (set to 3, i.e., *Aggressive cut strategy*). According to our computational experience, these settings gave the best average results, for the respective solvers, on the instances we considered.

Our testbed is made by general-integer MIP instances drawn from these four sources:

1. instances from MIPLIB 2003 [5];
2. instances from MILPlib [56];
3. the periodic scheduling instances described in [68];
4. the network design and multicommodity routing instances described in [19].

Pure 0-1 instances from all sets have been excluded from the comparison, as they have been addressed in [23].

Table 2.1 reports the instance name, the corresponding number of variables (n), of 0-1 variables ($|\mathcal{B}|$), of general-integer variables ($|\mathcal{G}| = |\mathcal{I}| - |\mathcal{B}|$) and of constraints ($m$).

The results of our experiments are reported in Table 2.2 (`Xpress Optimizer` vs `FP-xpress`) and in Table 2.3 (`ILOG Cplex` vs `FP-cplex`). The focus was to evaluate the capability of the compared methods to converge to an initial feasible solution, hence all methods were stopped as soon as the first feasible solution was found. For the MIP solvers, the tables report the computing time to get the first feasible solution (*time*) and the corresponding number of branching nodes (*nodes*).

As to `FP`, we report the computing time to get the first feasible solution (*time*), the stage where this solution was found (*stage*), the overall number of `FP` iterations (*iter*) and of restarts (*restarts*) in stages 1 and 2. The last column of the tables gives the computing time speedup of `FP` over the compared methods (a value greater than 1 meaning that `FP` was faster). Finally, the last rows of the tables report the total computing time needed to process the whole testbed, and the average speedup of `FP` over the compared method.

Name	n	$\|\mathcal{B}\|$	$\|\mathcal{G}\|$	m	source
arki001	1388	415	123	1048	[5]
atlanta-ip	48738	46667	106	21732	[5]
gesa2	1224	240	168	1392	[5]
gesa2-o	1224	384	336	1248	[5]
manna81	3321	18	3303	6480	[5]
momentum2	3732	1808	1	24237	[5]
momentum3	13532	6598	1	56822	[5]
msc98-ip	21143	20237	53	15850	[5]
mzzv11	10240	9989	251	9499	[5]
mzzv42z	11717	11482	235	10460	[5]
noswot	128	75	25	182	[5]
roll3000	1166	246	492	2295	[5]
rout	556	300	15	291	[5]
timtab1	397	64	107	171	[5]
timtab2	675	113	181	294	[5]
neos10	23489	23484	5	46793	[56]
neos16	377	336	41	1018	[56]
neos20	1165	937	30	2446	[56]
neos7	1556	434	20	1994	[56]
neos8	23228	23224	4	46324	[56]
ic97_potential	728	450	73	1046	[68]
ic97_tension	703	176	4	319	[68]
icir97_potential	2112	1235	422	3314	[68]
icir97_tension	2494	262	573	1203	[68]
rococoB10-011000	4456	4320	136	1667	[19]
rococoB10-011001	4456	4320	136	1677	[19]
rococoB11-010000	12376	12210	166	3792	[19]
rococoB11-110001	12431	12265	166	8148	[19]
rococoB12-111111	9109	8910	199	8978	[19]
rococoC10-001000	3117	2993	124	1293	[19]
rococoC10-100001	5864	5740	124	7596	[19]
rococoC11-010100	12321	12155	166	4010	[19]
rococoC11-011100	6491	6325	166	2367	[19]
rococoC12-100000	17299	17112	187	21550	[19]
rococoC12-111100	8619	8432	187	10842	[19]

Table 2.1: Test bed of MIPs with general integer variables

Computing times are expressed in CPU seconds, and refer to an Intel Pentium IV 2.4GHz personal computer with 512 Mbyte of main memory. A time limit of 1 hour of CPU time was imposed for all methods.

	`Xpress Optimizer`		`FP-xpress`				
name	time	nodes	time	stage	iter	restarts	speedup
arki001	7.03	1	66.70	3	1132	100	0.11
atlanta-ip	962.36	220	191.83	1	53	12	5.02
gesa2	0.05	1	0.06	2	5	0	0.75
gesa2-o	0.07	1	0.14	2	25	5	0.50
manna81	0.16	1	3.78	2	3	0	0.04
momentum2	1996.14	295	$>$ 3600.00	3	442	123	$<$ 0.55
momentum3	$>$ 3600.00	1	1479.75	3	350	128	$>$ 2.43
msc98-ip	303.23	334	23.91	1	30	6	12.68
mzzv11	251.56	194	26.66	1	1	0	9.44
mzzv42z	8.45	1	19.52	1	2	0	0.43
noswot	0.02	1	0.00	2	4	0	5.21
roll3000	12.45	72	0.84	2	7	0	14.80
rout	0.06	1	0.05	1	25	9	1.33
timtab1	3.75	1819	0.77	2	293	31	4.90
timtab2	124.58	65387	6.97	3	806	60	17.88
neos10	19.41	1	13.31	1	2	0	1.46
neos16	$>$ 3600.00	1154567	$>$ 3600.00	3	726	70	1.00
neos20	12.11	634	9.95	3	685	82	1.22
neos7	0.20	1	0.17	2	3	0	1.21
neos8	19.30	1	45.08	1	1	0	0.43
ic97_potential	0.05	1	4.75	3	991	35	0.01
ic97_tension	2.92	1325	2.13	2	659	47	1.38
icir97_potential	$>$ 3600.00	99765	13.75	3	767	17	$>$ 261.82
icir97_tension	10.20	714	22.74	3	775	116	0.45
rococoB10-011000	0.69	1	1.13	1	18	1	0.61
rococoB10-011001	0.66	1	0.83	1	27	2	0.79
rococoB11-010000	2.03	1	2.33	1	25	1	0.87
rococoB11-110001	5.47	1	4.95	1	14	0	1.10
rococoB12-111111	1520.30	2376	$>$ 3600.00	3	736	102	$<$ 0.42
rococoC10-001000	0.20	1	0.75	1	63	13	0.27
rococoC10-100001	0.95	1	3.44	1	63	10	0.28
rococoC11-010100	2.08	1	2.45	1	19	1	0.85
rococoC11-011100	1.03	1	1.82	1	20	1	0.57
rococoC12-100000	8.39	1	8.08	1	14	0	1.04
rococoC12-111100	3.06	1	2.02	1	13	0	1.52
Total times	16078.96		12760.64	Geometric mean			1.14

Table 2.2: Convergence to a first feasible solution using `Xpress Optimizer`

According to the tables, `FP` compares favorably with both MIP solvers. Indeed, both `FP-xpress` and `Xpress Optimizer` found a feasible solution for all but 3 instances, but `FP-xpress` was 14% (in geometric mean) faster than `Xpress Optimizer` in finding its first solution. As to the `ILOG Cplex` implementation, `FP-cplex` found a feasible solution to all but one instance, thus solving one instance more than `ILOG Cplex` and was 2.00 times (geometric mean) faster than `ILOG Cplex`. Also to be noted is that 25 out of the 35 instances have been solved

	`ILOG Cplex`		`FP-cplex`				
name	time	nodes	time	stage	iter	restarts	speedup
arki001	2.83	474	46.53	3	937	74	0.06
atlanta-ip	1562.58	230	113.64	1	5	0	13.75
gesa2	0.05	0	0.02	2	4	0	3.00
gesa2-o	0.25	90	0.03	2	6	0	8.00
manna81	0.22	0	0.34	2	3	0	0.64
momentum2	$>$ 3600.00	0	$>$ 3600.00	3	585	131	1.00
momentum3	$>$ 3600.00	0	1248.13	3	393	125	$>$ 2.88
msc98-ip	1330.23	120	97.09	1	37	4	13.70
mzzv11	243.34	80	214.83	1	3	0	1.13
mzzv42z	46.58	50	68.56	1	2	0	0.68
noswot	0.00	0	0.00	2	3	0	1.00
roll3000	7.05	300	0.83	2	6	0	8.51
rout	0.34	90	0.05	1	29	5	7.33
timtab1	0.88	752	0.08	2	37	3	11.20
timtab2	129.31	49264	2.14	2	631	64	60.41
neos10	6.88	0	8.28	1	2	0	0.83
neos16	1272.05	400000	1660.88	3	755	99	0.77
neos20	2.17	194	7.41	3	696	93	0.29
neos7	0.64	50	1.84	3	296	139	0.35
neos8	6.80	0	5.00	1	1	0	1.36
ic97_potential	0.52	40	2.98	3	775	18	0.17
ic97_tension	5.11	4730	2.67	3	1110	99	1.91
icir97_potential	3.48	120	61.09	3	787	7	0.06
icir97_tension	2380.35	464527	4.38	2	344	54	544.08
rococoB10-011000	1.14	0	1.41	1	23	1	0.81
rococoB10-011001	8.06	70	0.89	1	23	1	9.05
rococoB11-010000	1.86	0	3.20	1	22	0	0.58
rococoB11-110001	5.75	0	7.80	1	22	0	0.74
rococoB12-111111	1808.09	3590	718.55	3	899	101	2.52
rococoC10-001000	0.28	0	0.50	1	53	11	0.56
rococoC10-100001	558.73	1520	2.03	1	58	8	275.07
rococoC11-010100	1.48	0	3.34	1	27	1	0.44
rococoC11-011100	2.13	0	2.39	1	26	1	0.89
rococoC12-100000	51.72	20	7.13	1	21	0	7.26
rococoC12-111100	2.00	0	3.30	1	13	0	0.61
Total times	16642.90		7897.33	Geometric mean			2.00

Table 2.3: Convergence to a first feasible solution using `ILOG Cplex`

by `FP-cplex` either in Stage 1 or 2, i.e., without the enumeration of Stage 3.

To test the effectiveness of the binary stage, we also ran `FP-cplex` with its Stage 1 disabled. The results are reported in Table 2.4 and show that the binary stage has a really big impact on the overall performance: without Stage 1, 4 more instances could not be solved by `FP-cplex`, whose computing time was on average 9 times worse due to the increased number of iterations and of auxiliary variables (the latter reported in column *aux var*) required.

	FP with binary stage			FP without binary stage					
			aux				iter	aux	var
Name	time	iter	vars	time	ratio	iter	diff	vars	diff
arki001	46.53	937	96	30.33	0.652	685	-252	95	-1
atlanta-ip	113.64	5	0	168.47	1.482	223	218	68	68
gesa2	0.02	4	35	0.09	6.000	13	9	26	-9
gesa2-o	0.03	6	25	0.08	2.500	11	5	27	2
manna81	0.34	3	2497	0.44	1.273	3	0	2504	7
momentum2	$>$ 3600.00	585	1	$>$ 3600.00	1.000	616	31	1	0
momentum3	1248.13	393	1	$>$ 3600.00	2.884	441	48	1	0
msc98-ip	97.09	37	0	105.50	1.087	72	35	49	49
mzzv11	214.83	3	0	873.75	4.067	638	635	131	131
mzzv42z	68.56	2	0	488.70	7.128	662	660	141	141
noswot	0.00	3	4	0.02		19	16	12	8
roll3000	0.83	6	27	64.20	77.528	900	894	466	439
rout	0.05	29	0	0.11	2.333	41	12	7	7
timtab1	0.08	37	88	0.17	2.200	67	30	91	3
timtab2	2.14	631	163	1.72	0.803	421	-210	160	-3
neos10	8.28	2	0	7.31	0.883	1	-1	0	0
neos16	1660.88	755	41	874.41	0.526	978	223	41	0
neos20	7.41	696	30	9.67	1.306	978	282	30	0
neos7	1.84	296	20	1.91	1.034	197	-99	20	0
neos8	5.00	1	0	5.66	1.131	1	0	0	0
ic97_potential	2.98	775	68	4.81	1.613	1183	408	73	5
ic97_tension	2.67	1110	4	1.95	0.731	938	-172	4	0
icir97_potential	61.09	787	291	96.58	1.581	713	-74	292	1
icir97_tension	4.38	344	556	11.31	2.586	431	87	573	17
rococoB10-011000	1.41	23	0	629.59	447.711	633	610	134	134
rococoB10-011001	0.89	23	0	91.72	102.982	632	609	134	134
rococoB11-010000	3.20	22	0	2146.19	670.029	632	610	166	166
rococoB11-110001	7.80	22	0	$>$ 3600.00	461.723	636	614	166	166
rococoB12-111111	718.55	899	173	$>$ 3600.00	5.010	612	-287	193	20
rococoC10-001000	0.50	53	0	22.59	45.188	456	403	124	124
rococoC10-100001	2.03	58	0	2012.59	990.815	416	358	122	122
rococoC11-010100	3.34	27	0	1234.38	369.159	524	497	165	165
rococoC11-011100	2.39	26	0	527.63	220.706	621	595	163	163
rococoC12-100000	7.13	21	0	$>$ 3600.00	505.263	574	553	187	187
rococoC12-111100	3.30	13	0	$>$ 3600.00	1091.943	518	505	186	186
				mean	8.986		224		69

Table 2.4: Comparison of FP with and without binary stage

Table 2.5 reports the total time and percent time spent by `FP-cplex` in each individual stage.

	times			percentages		
Name	stage 1	stage 2	stage 3	stage 1	stage 2	stage 3
arki001	0.17	39.30	8.31	0.36%	82.24%	17.40%
atlanta-ip	8.73			100.00%		
gesa2	0.00	0.02		0.00%	100.00%	
gesa2-o	0.00	0.03		0.00%	100.00%	
manna81	0.00	0.27		0.00%	100.00%	
momentum2	175.77	224.83	3199.41	4.88%	6.25%	88.87%
momentum3	160.08	432.44	160.77	21.25%	57.41%	21.34%
msc98-ip	7.81			100.00%		
mzzv11	2.13			100.00%		
mzzv42z	1.09			100.00%		
noswot	0.00	0.00				
roll3000	0.52	0.06		89.19%	10.81%	
rout	0.06			100.00%		
timtab1	0.02	0.08		16.67%	83.33%	
timtab2	0.14	2.02		6.52%	93.48%	
neos10	4.72			100.00%		
neos16	0.55	1.94	1713.63	0.03%	0.11%	99.86%
neos20	0.45	4.17	2.95	5.98%	55.05%	38.97%
neos7	0.28	1.52	0.08	15.00%	80.83%	4.17%
neos8	1.67			100.00%		
ic97_potential	0.52	2.47	0.13	16.58%	79.40%	4.02%
ic97_tension	0.19	2.00	0.52	6.94%	73.99%	19.08%
icir97_potential	1.73	8.98	52.67	2.74%	14.17%	83.09%
icir97_tension	0.95	3.47		21.55%	78.45%	
rococoB10-011000	0.30			100.00%		
rococoB10-011001	0.33			100.00%		
rococoB11-010000	0.94			100.00%		
rococoB11-110001	1.28			100.00%		
rococoB12-111111	45.13	79.66	608.11	6.16%	10.87%	82.97%
rococoC10-001000	0.39			100.00%		
rococoC10-100001	1.42			100.00%		
rococoC11-010100	1.02			100.00%		
rococoC11-011100	0.66			100.00%		
rococoC12-100000	1.55			100.00%		
rococoC12-111100	0.53			100.00%		
mean over all instances				54.68%	29.33%	13.14%
mean over instances performing the stage				54.68%	57.02%	93.31%

Table 2.5: Time spent in each stage

Finally, in order to validate the effectiveness of our approach we compared these results with the performance of the original `FP` algorithm [23]. Since this method can only handle 0-1 MIPs, we converted each model in our testbed to a 0-1 problem by replacing each general-integer variable with a set of binary variables representing the binary encoding of the integer values. More precisely, we replaced each general-integer variable x_i, where $0 \leq x_i \leq u_i$, with $n_i := \lceil \log_2(u_i + 1) \rceil$ binary variables x_{ik} such that $x_i = \sum_{k=0}^{n_i-1} 2^k x_{ik}$. The original `FP` applied to the resulting 0-1 MIPs turned out to be faster in reaching its first feasible solution on just 3 instances (namely, arki001, neos10, and rococoC11-011100), whereas, on all other instances, it took much longer or could not find any solution at all.

2.4 Improving feasible solutions

As already mentioned, in the previous experiments our main attention was on the computing time spent to find a first feasible solution. In this respect, the `FP` results were very satisfactory. However, the quality of the solution delivered by `FP` is often considerably worse than that computed (in a typically longer time) by `ILOG Cplex` or `Xpress Optimizer`. This can be explained by noting that the `FP` method uses the original objective function only at step 1, when the solution of the LP relaxation is used to initialize x^*, while the original costs are completely disregarded during the next iterations. As a consequence, the quality of x^* and $\tilde{x}$ tends to deteriorate rapidly with the number of iterations and of restarts performed. This explains why the same behavior is much less pronounced in the binary case studied in [23], where driving the pair $(x^*, \tilde{x})$ towards feasibility turns out to be much easier than in the general-integer case and requires a considerably smaller number of iterations and of restarts.

In this section we investigate three simple `FP` strategies aimed at improving the quality of the solutions found by the method.

The first strategy is based on the idea of adding an artificial upper bound constraint $c^T x \leq UB$ to the LP solved at step 6, where UB is updated dynamically each time an improved feasible solution is found. To be more specific, right after step 1 we initialize $z^*_{LP} = c^T x^*$ (= LP relaxation value) and $UB = +\infty$. Each time an improved feasible MIP solution x^* of value $z^H = c^T x^*$ is found at step 6, we update $UB = \alpha z^*_{LP} + (1 - \alpha) z^H$ for a certain $\alpha \in (0, 1)$, and continue the while-do loop. We observed that, due to the additional constraint $c^T x \leq UB$, it is often the case that the integer components of $\tilde{x}$ computed at step 9 define a feasible point of the *original* system $Ax \geq b$, but not of the current one. In order to improve the chances of updating the incumbent solution, we therefore apply (right after step 9) a simple post-processing of $\tilde{x}$ consisting in solving the LP $\min\{c^T x : Ax \geq b,\ x_j = \tilde{x}_j\ \forall j \in \mathcal{I}\}$ and comparing the corresponding solution $\overline{x}$ (if any) with the incumbent one–solution $\overline{x}$ being guaranteed to be feasible for the original problem, as all the integer-constrained variables have been fixed at their corresponding value in $\tilde{x}$.

In the other two strategies, we stop `FP` as soon as it finds a feasible solution, and pass this solution either to a Local Branching heuristic [24], or to a MIP solver using RINS strategy [19].

	(Ref) Cplex	Cplex				
name	emp=1	emp=4	FP-20%	FP-30%	FP-lb	FP-rins
arki001	7.581E+06	1.0000	1.0007	1.0006	0.9999 *	1.0000
atlanta-ip	1.000E+02	N/A	0.9600	0.9800	0.9600	0.9500 *
gesa2	2.578E+07 *	1.0000 *	1.0004	1.0004	1.0000	1.0000 *
gesa2-o	2.578E+07 *	1.0000 *	1.0011	1.0013	1.0000 *	1.0000 *
manna81	-1.316E+04 *	1.0000 *	1.0000 *	1.0005	1.0000 *	1.0000 *
msc98-ip	2.250E+07	N/A	0.8993	0.8984 *	0.9529	0.9699
mzzv11	-2.172E+04 *	1.0000 *	1.2209	1.0950	1.1144	1.0018
mzzv42z	-2.054E+04 *	1.0000	1.0235	1.0188	1.0118	1.0000
noswot	-4.100E+01 *	1.0000 *	1.0000 *	1.0000 *	1.0000 *	1.0000 *
roll3000	1.343E+04	0.9596 *	1.0708	1.1188	0.9800	0.9657
rout	1.078E+03	1.0000	1.0151	1.0061	1.0000 *	1.0000 *
timtab1	7.927E+05	0.9647 *	1.3123	1.1528	1.0034	1.6313
timtab2	1.232E+06	0.8990 *	1.3245	1.1675	1.0224	0.9648
neos10	-1.135E+03 *	1.0000 *	4.4862	2.9481	1.0000 *	1.0000 *
neos16	4.510E+02	N/A	1.0067	1.0067	1.0067	0.9978 *
neos20	-4.340E+02 *	1.0000 *	4.1731	4.1731	1.0383	1.0000
neos7	7.219E+05 *	1.0000	1.0582	1.0028	1.0000	1.0000
neos8	-3.719E+03	1.0000	1.0005	3.1570	1.0000 *	1.0000 *
ic97_potential	3.961E+03	0.9965 *	1.0106	1.0155	0.9970	0.9965 *
ic97_tension	3.942E+03 *	1.0003	1.0018	1.0032	1.0000 *	1.0000 *
icir97_potential	6.410E+03	0.9964	1.0264	1.0434	1.0034	0.9945 *
icir97_tension	6.418E+03	0.9949	0.9948	0.9996	0.9956	0.9938 *
rococoB10-011000	1.951E+04	0.9967 *	1.1947	1.0873	1.0365	1.0593
rococoB10-011001	2.131E+04 *	1.0501	1.2451	1.2443	1.0037	1.1349
rococoB11-010000	3.348E+04	0.9901 *	1.1968	1.0978	1.0138	1.1833
rococoB11-110001	4.947E+04	0.9738 *	1.5647	1.2136	1.0573	1.2941
rococoB12-111111	4.623E+04	0.8589 *	2.0923	2.0923	1.0035	1.0372
rococoC10-001000	1.146E+04 *	1.0004	1.1645	1.0883	1.0013	1.0013
rococoC10-100001	1.943E+04	0.9336 *	1.5803	1.7790	0.9377	1.0649
rococoC11-010100	2.163E+04 *	1.0189	1.1680	1.0668	1.0389	1.3361
rococoC11-011100	2.192E+04	0.9561 *	1.1306	1.2290	1.0410	1.1887
rococoC12-100000	3.753E+04 *	1.0177	1.6447	1.4960	1.0742	1.0775
rococoC12-111100	4.097E+04	0.9138 *	1.0858	1.0176	0.9794	0.9448
Geometric means		0.9833 (+)	1.2352	1.2292	1.0078	1.0469

(+) not counting the 3 cases of failure

Table 2.6: Solution quality with respect to ILOG Cplex (emp=1)

Table 2.6 compares the quality of the best solution returned by `ILOG Cplex` with that of the solution found (within a 3600-second time limit) by `FP-cplex` and then improved by means of one of the three strategies above. In the table, the first four columns report the instance name (*name*), the value of the LP relaxation (*LP relax*) and of best feasible solutions found within the 3600-second time limit by `ILOG Cplex` (*Cplex*) with two different settings of its `MIP emphasis` parameter, namely "`emp=1`" for *integer feasibility* and "`emp=4`" for *hidden feasible* *solutions* (i.e., RINS heuristic). As to `FP-cplex` with the artificial upper bound, it was run with the same settings described earlier, by requiring a 20% (respectively, 30%) improvement at each main iteration (i.e., with $\alpha \in \{0.2, 0.3\}$); see columns *FP-XX%*. Tests with $\alpha = 0.1$ and $\alpha = 0.4$ led to slightly worse solutions (with an average quality ratio of about 1.26) and are not shown in the table.

Column *FP-lb* refers to the Local Branching implementation available in `ILOG Cplex` 9.1 by activating its *local branching* flag, whereas column *FP-rins* refers to `ILOG Cplex` with MIP emphasis 4 (that activates the internal RINS improvement heuristics). For both *FP-lb* and *FP-rins*, the incumbent solution is initialized, via an MST file, by taking the first `FP-cplex` solution.

For all strategies, the table gives the solution ratio with respect to the best solution found by `ILOG Cplex` (emp=1). Ratios were computed as the value of the best solution found by the various methods over the value of the solution found by `ILOG Cplex` (emp=1); if the values were negative, the problem was viewed as a maximization one and the ratio was inverted, hence a ratio smaller than 1.0 always indicates an improvement over `ILOG Cplex`. In the last line of the table, the average ratio (geometric mean) is reported; the average does not take into account the instances where `FP` succeeded in finding a solution, while `ILOG Cplex` did not. For each instance, we marked with an asterisk the method that produced the best feasible solution.

According to the table, all the `FP` methods are able to improve significantly the quality of their incumbent solution. The most effective `FP` strategies seem to be *FP-lb* and *FP-rins*, that produced the best solutions in 8 and 13 cases, respectively.

`ILOG Cplex` (emp=1) ranked first 14 times. As to `ILOG Cplex` (emp=4), it produced the best solution in 18 cases but failed in 3 cases to find any solution within the 3600-second time limit. Moreover, pure `ILOG Cplex` methods seem to be particularly suited for exploiting the structure of *rococo** instances–if these 11 instances were removed from the testbed, *FP-rins* would have ranked first 13 times, thus outperforming both `ILOG Cplex` (emp=1, that ranked first 10 times) and `ILOG Cplex` (emp=4, first 11 times but with 3 failures).

Among the compared *FP-XX%* methods, the one requiring 30% improvement at each main iteration is the more effective one, though its performance is still inferior to the one of the LB/RINS local search methods.

Finally, Figures 2.3, 2.4, 2.5, and 2.6 plot the value of the best feasible solution over time, for the four instances atlanta-ip, msc98-ip, icir97_tension, and rococoC10-100001.

2.5 Conclusions

In this chapter we addressed the problem of finding a feasible solution of a given MIP model, which is a very important ($\mathcal{NP}$-complete) problem that can be extremely hard in practice.

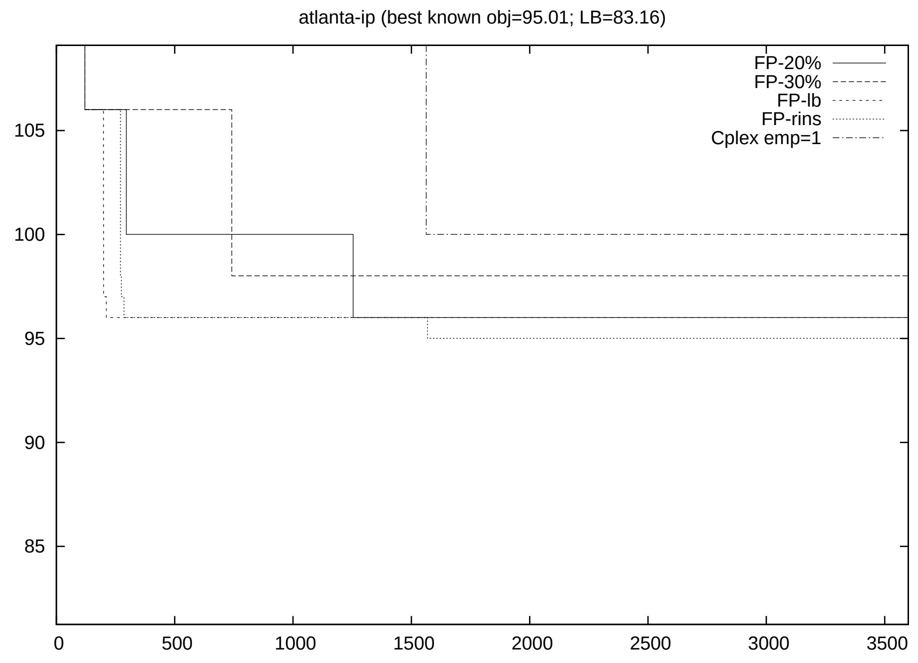


Figure 2.3: Incumbent solution over time (instance atlanta-ip)

We elaborated the Feasibility Pump (FP) heuristic presented in [23], and extended it in two main directions, namely (i) handling as effectively as possible MIP problems with both binary and general-integer variables, and (ii) exploiting the FP information to drive an effective enumeration phase.

We presented extensive computational results on large sets of test instances from the literature, showing the effectiveness of our improved FP scheme for finding feasible solutions to hard MIPs with general-integer variables.

As to the solution quality, it appears to be rather poor when the very first feasible solution is found, but it can be improved considerably by integrating FP with improvement tools such as Local Branching or RINS.

Future directions of work include extending the FP idea by using a nonlinear (quadratic) distance function, to be applied to linear and (even more interestingly) to nonlinear problems with integer variables. Also interesting is the incorporation of the original objective function (through an adaptive scaling multiplier) in the definition of the FP distance function; interesting results in this directions have been recently reported by Achterberg and Berthold [3] and used in the non-commercial MIP solver SCIP [2].

Finally, a topic to be investigated is the integration of FP within an overall enumerative solution scheme. In this context, the FP heuristic can of course be applied at the root node, so as to hopefully initialize the incumbent solution. But one can also think of running FP (possibly without its time-consuming stage 3) from the LP relaxation of different nodes in the branch-and-cut tree, thus increasing the chances of finding improved feasible solutions.

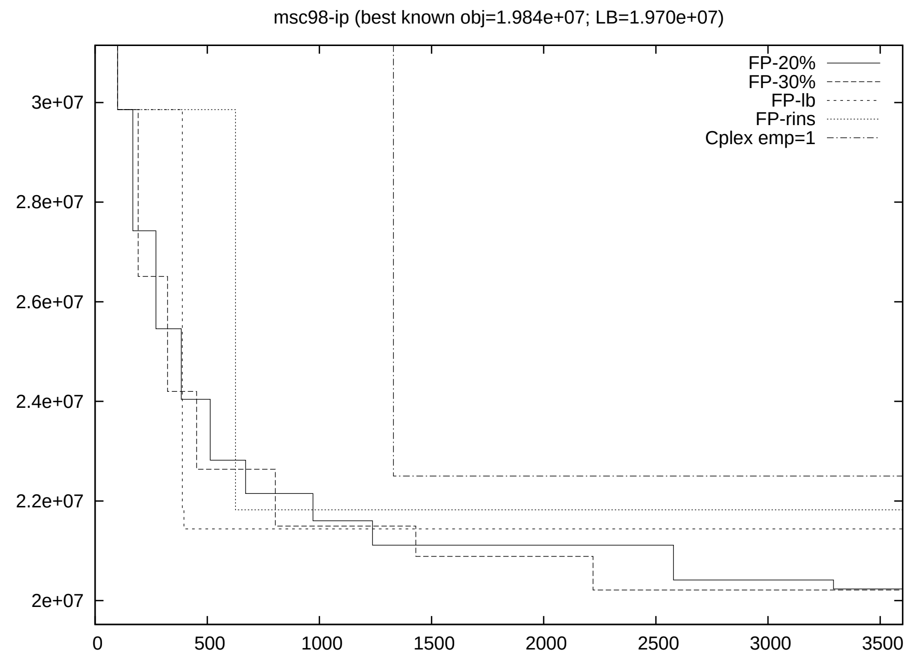


Figure 2.4: Incumbent solution over time (instance msc98-ip)

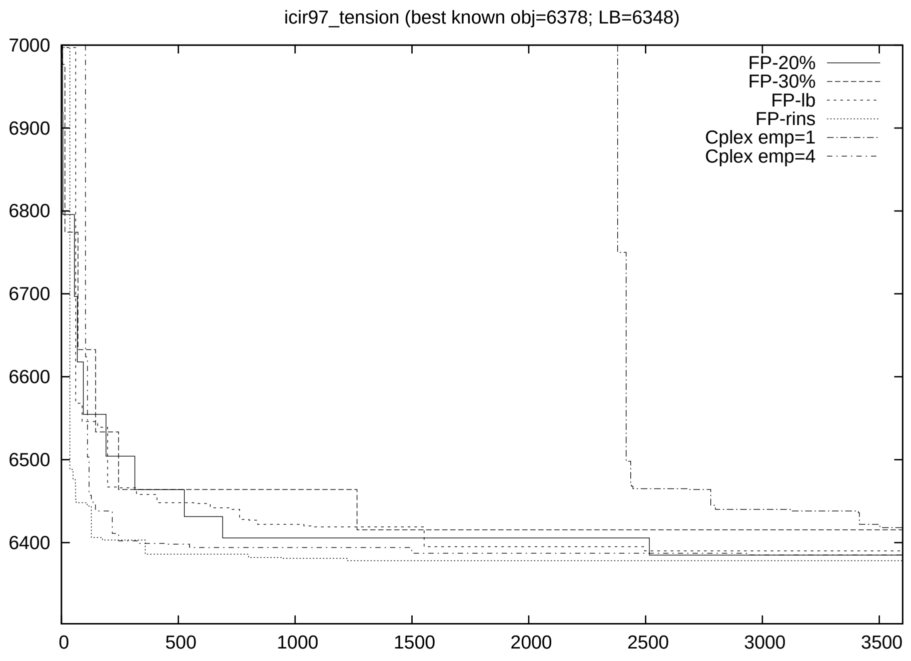


Figure 2.5: Incumbent solution over time (instance icir97_tension)

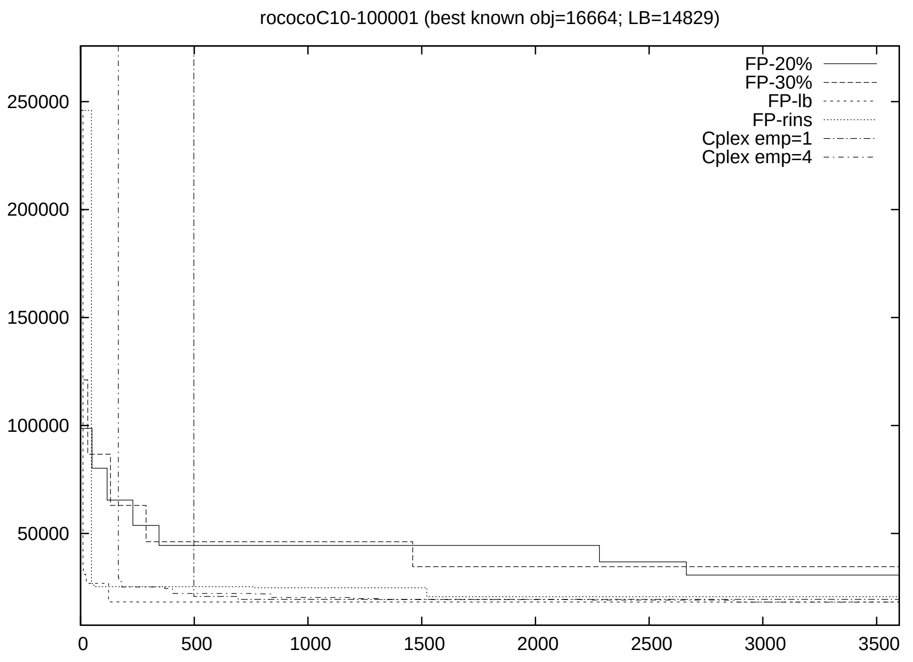


Figure 2.6: Incumbent solution over time (instance rococoC10-100001)

Chapter 3

Linear Ordering Problem with Cumulative Costs

3.1 Introduction

Several optimization problems require finding a permutation of a given set of items that minimizes a certain cost function. These problems are naturally modeled in graph-theory terms by introducing a complete (loopless) digraph $G = (V, A)$ whose vertices $v \in V := \{1, \cdots, n\}$ correspond to the n items to be sorted. By construction, there is a 1-1 correspondence between the Hamiltonian paths $P = \{(k_1, k_2), \cdots, (k_{n-1}, k_n)\}$ in G (viewed as arc sets) and the item permutations $\mathcal{K} = \langle k_1, \cdots, k_n \rangle$.

Depending on the cost function to be be used, different optimization problems can be defined on G. The most familiar one arises when the cost of a given permutation $\mathcal{K}$ only depends on the consecutive pairs (k_i, k_{i+1}), $i = 1, \cdots, n-1$. In this case, one can typically associate a cost c_{uv} with each arc $(u, v) \in A$, and the problem reduces to finding a min-cost *Hamiltonian Path* (HP) in G, a relative of the famous Traveling Salesman Problem (TSP) [50, 38]. Note however that this model is only appropriate when the overall cost is simply the sum of the "direct costs" of putting an item right after another in the final permutation. A more complex situation arises when a given cost g_{uv} has to be paid whenever item u is ranked before item v in the final permutation. In this case, a feasible solution can be more conveniently associated with an *acyclic tournament*, defined as the transitive closure of an Hamiltonian path $P = \{(k_1, k_2), \cdots, (k_{n-1}, k_n)\}$:

$$[P] := \{(k_i, k_j) \in A : i = 1, \cdots, n-1, j = i+1, \cdots, n\}$$

see Figure 3.1 for an illustration. The resulting problem then calls for a min-cost acyclic tournament in G, and is known as the *Linear Ordering Problem* (LOP) [32, 33, 34, 70]. Both HP and LOP are known to be $\mathcal{NP}$-hard problems.

In some applications, both the HP and the LOP frameworks are unappropriate to describe the cost function. In this chapter we introduce and study, for the first time, a relevant case arising when the overall permutation cost can be expressed

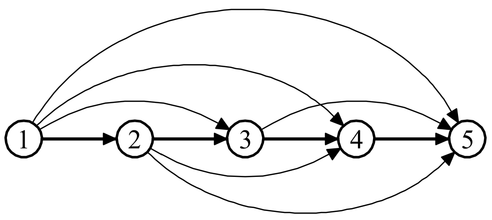


Figure 3.1: Acyclic tournaments as an Hamiltonian path (thick arcs) plus its transitive closure (thin arcs)

as the sum of terms α_u associated with each item u, each defined as a linear combination of the values α_v of all items v that follow u in the permutation. To be more specific, we address the following problem:

Definition 3.1.1 (LOP-CC). *Given a complete digraph $G = (V, A)$ with nonnegative node weights p_v and nonnegative arc costs c_{uv}, the* Linear Ordering Problem with Cumulative Costs *(LOP-CC) is to find an Hamiltonian path $P = \{(k_1, k_2), \cdots, (k_{n-1}, k_n)\}$ and the corresponding node values α_v that minimize the total cost*

$$\pi(P) = \sum_{v=1}^{n} \alpha_v$$

under the constraints

$$\alpha_{k_i} = p_{k_i} + \sum_{j=i+1}^{n} c_{k_i k_j} \alpha_{k_j}, \quad \text{for } i = n, n-1, \cdots, 1 \tag{3.1}$$

Constraints (3.1) imply a *cumulative* "backward propagation" of the value of variables α_v for $v = n, n-1, \cdots, 1$, hence the name of the problem. We will also address a constrained version of the same problem, namely:

Definition 3.1.2 (BLOP-CC). *The* Bounded Linear Ordering Problem with Cumulative Costs *(BLOP-CC) is defined as the problem LOP-CC above, plus the additional constraints:*

$$\alpha_i \leq U \quad \forall i \in V \tag{3.2}$$

where U is a given nonnegative bound.

Notice that BLOP-CC can be infeasible. As shown in the next section, BLOP-CC finds important practical applications, in particular, in the optimization of mobile telecommunication systems.

As G is assumed to be complete, in the sequel we will not distinguish between an Hamiltonian path $P = \{(k_1, k_2), \cdots, (k_{n-1}, k_n)\}$ and the associated node permutation $\mathcal{K} = \langle k_1, \cdots, k_n \rangle$. Moreover, given any Hamiltonian path $P = \{(k_1, k_2), \cdots, (k_{n-1}, k_n)\}$, we call *direct* all arcs $(k_i, k_{i+1}) \in P$ (the thick ones in Figure 3.1), whereas the arcs (k_i, k_j) for $j \geq i + 1$ are called *transitive* (these are precisely the arcs in $[P] \setminus P$, depicted in thin line in Figure 3.1). Finally, we use notation $\pi(P)$ to denote the *cumulative cost* of an Hamiltonian path P, defined as the LOP-CC cost $\pi = \sum_{v=1}^{n} \alpha_v$ of the corresponding permutation.

In this chapter we introduce and study both problems LOP-CC and BLOP-CC. In Section 3.2, we give the practical application that motivated the present study and leaded to the patented new methodology for cellular phone management described in [13]. In Section 3.3, we show that both LOP-CC and BLOP-CC are $\mathcal{NP}$-hard. A Mixed-Integer linear Programming (MIP) model is presented in Section 3.4, whereas an ad-hoc enumerative method is introduced in Section 3.5. Extensive computational results on a large set of instances are presented in Section 3.6, whereas a dynamic-programming heuristic is also described and evaluated in Section 3.7. Some conclusions are finally drawn in Section 3.8.

3.2 Motivation

In this section we outline the practical problem that motivated the present chapter; the interested reader is referred to [12], [41] and [69] for more details.

In wireless cellular communications, mobile terminals (MTs) communicate simultaneously with a common Base Station (BS). In order to distinguish among the signals of different MTs, the Universal Mobile Telecommunication Standard (UMTS) [1] adopts the so-called code division multiple access technique, where each terminal is identified by a specific code. Due to the distortions introduced by radio propagation, the MTs partially interfere with each other, hence the need to keep the multiuser access interference below an acceptable level. A very effective technique for interference reduction has been proposed [67], and is called Successive Interference Cancelation (SIC). According to this method, MT signals are detected sequentially from the received signal, according to a predetermined order. After each detection, interference is removed from the received signal, thus allowing for improved detection for the next users.

A crucial problem in the design of the SIC system is therefore the choice of the detection order. Usually, users are ordered by decreasing received power [67], although a better performance can be obtained by considering also the level of mutual interference among users. A second issue is the choice of the power level α_i at which the i-th user has to transmit its data. Indeed, a large power level typically allows for an improved signal detection, whereas the minimization of the transmission power yields a longer duration of the batteries of the MT.[1] Moreover, physical and regulatory constraints impose an upper bound, U, on the transmission power of the mobile terminals.

Both the choice of the cancelation order and of the transmission power levels must ensure a reliable detection of the signals coming from all MTs. A proper reception is ensured when the average Signal-to-Noise (plus Interference) power ratio (SNIR) is equal to a target level Γ. For a SIC receiver, the SNIR is related to the power of the interference generated from user i on user j, denoted by ρ_{ij}.

[1]Battery lifetime is one of the main limiting factors for mobile communication systems

In particular, upon detection of user k_p the SNIR is

$$SNIR^{(p)} = \frac{\alpha_{k_p}\rho_{k_pk_p}}{N_0\sqrt{\rho_{k_pk_p}} + \sum_{i\in\mathcal{U}_p}\alpha_i N_{\mathcal{S}}\rho_{ik_p}} \tag{3.3}$$

where N_0 (noise power) and $N_{\mathcal{S}}$ (spreading factor) are given parameters, and

$$\mathcal{U}_p = \{k_{p+1}, k_{p+2}, \cdots, k_n\} \tag{3.4}$$

is the set of undetected user at stage p.

One then faces the problem of jointly optimizing the SIC detection order and the transmission power levels, with the aim of minimizing the overall transmission power while ensuring a proper reception for all users. This problem, called joint power-control and receiver optimization (JOPCO), has been introduced in [12], and can be formalized as follows: given a set of users $\{1, 2, \ldots, n\}$, the interference factors ρ_{ij} $(i, j = 1, \ldots, n)$, the noise power N_0, the spreading factor $N_{\mathcal{S}}$, the target ratio Γ, and the maximum allowed power level U, find the transmission power levels α_i $(i = 1, \cdots, n)$ and the detection permutation $\mathcal{K} = \langle k_1, \ldots, k_n\rangle$ that minimize the total transmission power $\pi = \sum_{i=1}^{n}\alpha_{k_i}$ under the following constraints:

$$\Gamma = \frac{\alpha_{k_i}\rho_{k_ik_i}}{N_0\sqrt{\rho_{k_ik_i}} + \sum_{l\in\mathcal{U}_i}\alpha_l N_{\mathcal{S}}\rho_{lk_i}}, \quad \text{for } i = 1, \cdots, n \tag{3.5}$$

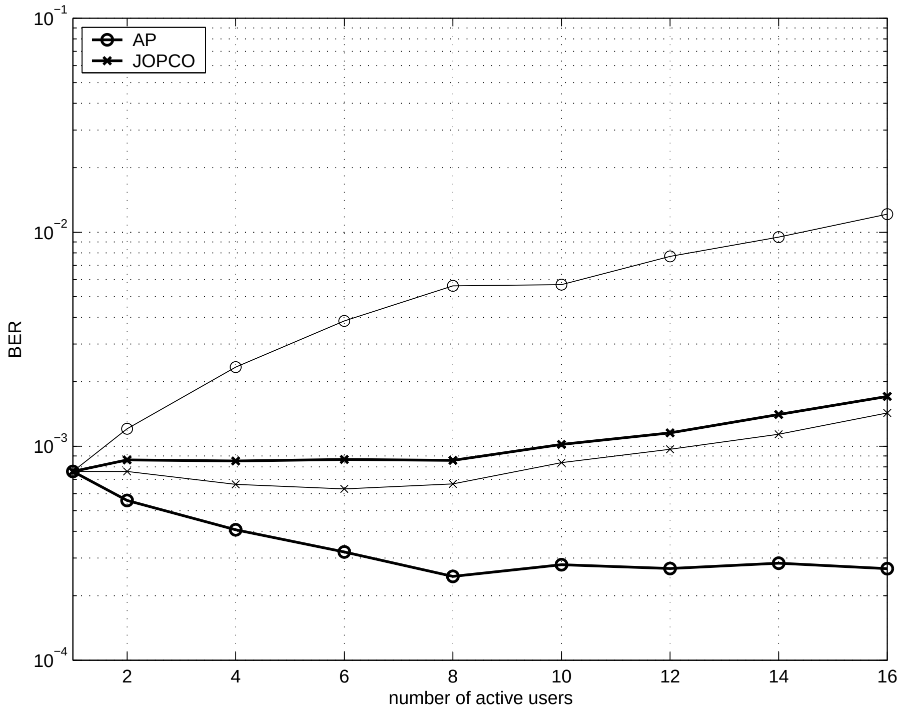
$$\alpha_i \leq U \tag{3.6}$$

In [12] a simple GRASP heuristic is proposed for JOPCO with the aim of minimizing the system transmit power under the constraint of ensuring the same quality of the transmission (measured by the average raw Bit Error Rate, BER) to all users.

In particular, the requirement on the BER is translated into a constraint on the SNIR at the detection point of each user, as discussed before. Extensive experiments are reported, showing that the JOPCO technique performs much better than the usual Average Power (AP) approach in all the four scenarios simulated, both in terms of quality of the transmission (BER) and of allocated transmission power.

Figure 3.2 (taken from [12]) illustrates the average BER vs. the number n of active users for *synchronous* and *asynchronous* transmission systems, and compares the JOPCO and AP methodologies. Thin and bold lines correspond to the case with and without the so-called *scrambling* operation on transmitted data, respectively.

It can be seen that, both with and without scrambling, JOPCO ensures approximately a constant average raw BER of 10^{-3} up to 10 active users with respect to the classical AP technique. In the case of synchronous transmission without scrambling, AP gives a BER that is even lower than the target, just because it allocates much more transmission power than necessary to guarantee the target quality, as can be seen in Figure 3.3 (top). When a larger number of active users



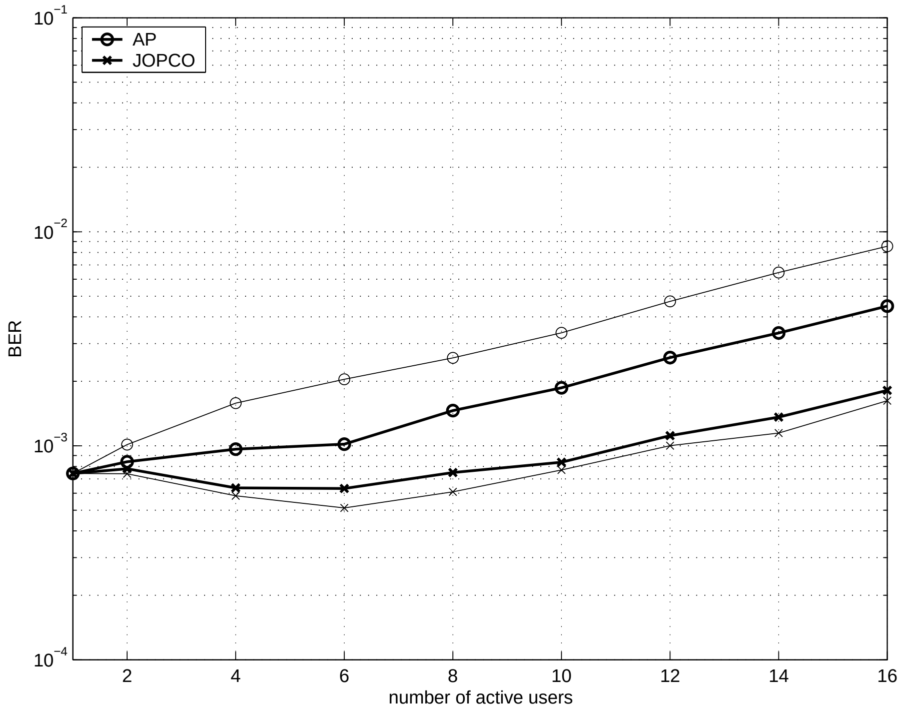


Figure 3.2: The average raw bit error rate BER vs. the number f active users for synchronous (top) and asynchronous (bottom) transmissions, with scrambling (thin line) and without scrambling (bold line)

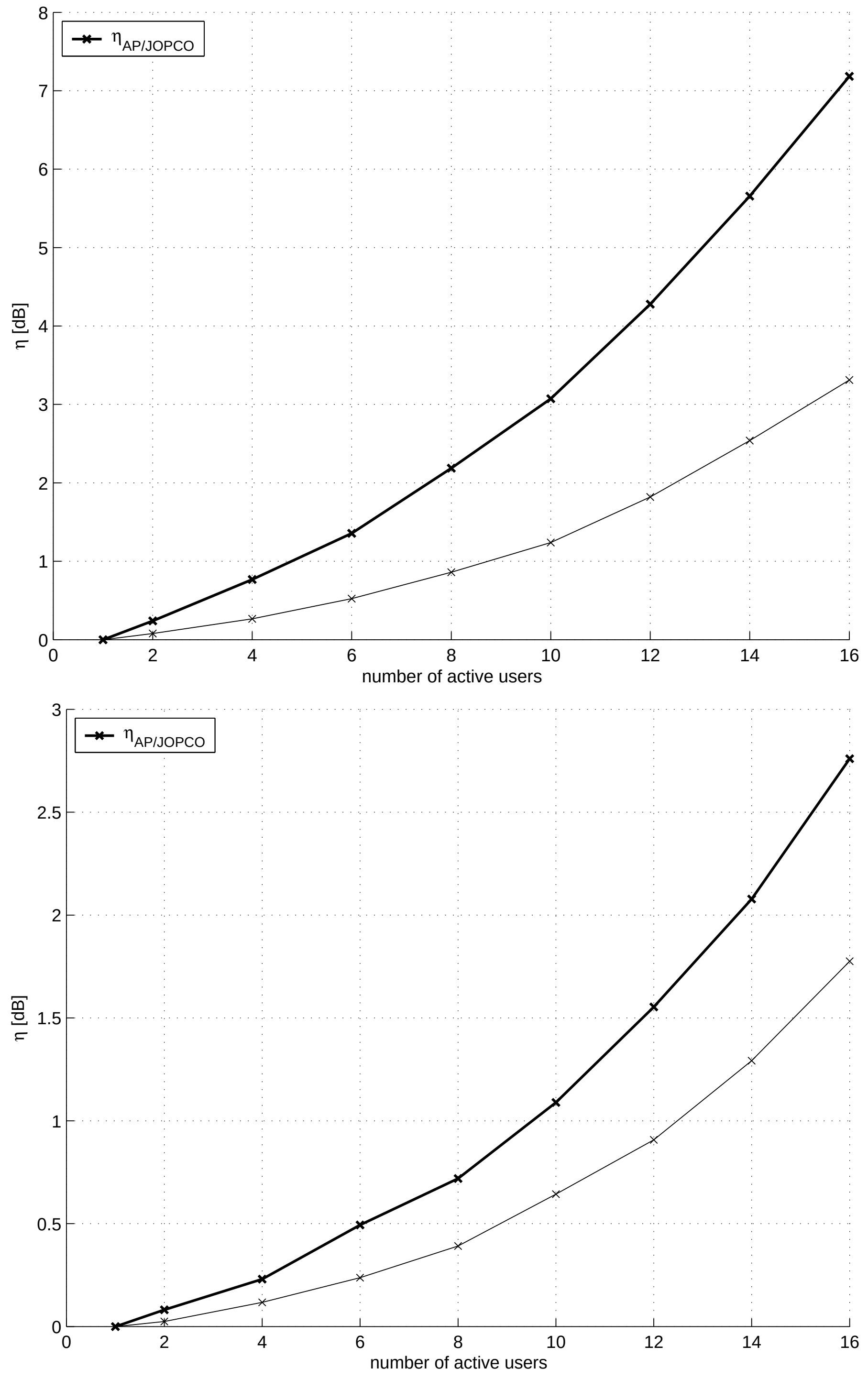


Figure 3.3: The expected total transmission power ratio η vs. the number of active users for synchronous (top) and asynchronous (bottom) transmissions, with scrambling (thin line) and without scrambling (bold line)

is present, instead, JOPCO has slight performance degradation due to errors in the interference cancelation.

Figure 3.3 (also taken from [12]) gives the expected total transmission power ratio expressed in dB, η, as a function of the number of active users (with and without scrambling), i.e.,

$$\eta = 10\log_{10}\mathrm{E}\left[\frac{P_{\mathrm{tot}}^{(AP)}}{P_{\mathrm{tot}}^{(JOPCO)}}\right]. \tag{3.7}$$

where $P_{\mathrm{tot}}^{(AP)}$ and $P_{\mathrm{tot}}^{(JOPCO)}$ represent the total transmission power allocated by AP and JOPCO, respectively.

In all cases, we observe that the JOPCO approach requires a reduced transmission power with respect to the AP approach. In particular, for a full loaded synchronous (resp., asynchronous) system without scrambling, on average the system power requirement using the JOPCO technique is 7 dB (resp., 3 dB) lower than that for AP. When scrambling is considered, instead, the JOPCO power requirement is 3 dB (resp., 2dB) lower than that for AP.

We next show how JOPCO can be formulated as a BLOP-CC. Clearly, for any given user permutation $\mathcal{K}$ the power levels α_i are univocally determined by the SNIR constraints (3.5). Indeed, rewriting (3.5) as

$$\alpha_{k_i} = \frac{\Gamma N_0\sqrt{\rho_{k_ik_i}} + \Gamma N_{\mathcal{S}}\sum_{l\in\mathcal{U}_i}\alpha_l\rho_{lk_i}}{\rho_{k_ik_i}}$$

one has that values α_{k_i} can easily be computed in the reverse order $i = n, n-1, \cdots, 1$. Defining the weights $p_i = \Gamma N_0/\sqrt{\rho_{ii}}$ and the costs $c_{ij} = \Gamma N_{\mathcal{S}}\rho_{ji}/\rho_{ii}$ one then obtains precisely the BLOP-CC formulation introduced in the previous section.

3.3 Complexity of the LOP-CC

We start proving that the LOP-CC problem is $\mathcal{NP}$-hard. We first give a simple outline of the proof, and then address in a formal proof the details required.

Our proof is by reduction from the following Hamiltonian Path problem, HP, which is known to be $\mathcal{NP}$-complete.

Definition 3.3.1 (HP). *Given a digraph $G_{HP} = (V_{HP}, A_{HP})$, decide whether G contains any directed Hamiltonian path.*

Our reduction takes any HP instance $G_{HP} = (V_{HP}, A_{HP})$ and computes the following LOP-CC instance:

$$V := V_{HP} = \{1, \cdots, n\} \tag{3.8}$$

$$p_i := 1 \quad \forall i \in V \tag{3.9}$$

$$c_{ij} = \begin{cases} M & \text{if } (i,j) \in A_{HP} \\ 2M & \text{otherwise} \end{cases} \qquad \forall i,j \in V,\ i \neq j \tag{3.10}$$

where M is a sufficiently large positive value (to be defined later). The construction can clearly be carried out in polynomial time, provided that value M can be stored in a polynomial number of bits a property that will be asserted in the formal proof.

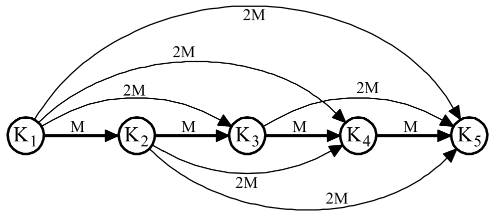


Figure 3.4: The worst-case "good" path used in the complexity proof

"Good" Hamiltonian paths of G (i.e., those corresponding to Hamiltonian paths in G_{HP}) only involve direct arcs of cost equal to M, while all the transitive arcs have a cost not larger than $2M$; see Figure 3.4 for an illustration. It is therefore not difficult to show that, for sufficiently large M, the overall cumulative cost associated with such a path is $\pi(P) = M^{n-1} + O(M^{n-2})$. On the other hand, if P does not correspond to a Hamiltonian path in G_{HP}, then it has to involve at least one direct arc of cost $2M$, hence its cumulative cost $\pi(P)$ cannot be smaller than $2M^{n-1}$. For a large M, our construction then ensures that $\pi(P) < \pi(P')$ for any "good" Hamiltonian path P and for any "non-good" Hamiltonian path P', which implies that G_{HP} contains an Hamiltonian path if and only if any arbitrary optimal LOP-CC solution corresponds to a "good" Hamiltonian path (or, equivalently, if the optimal LOP-CC value is strictly less than $2M^{n-1}$).

Theorem 3.3.1. *LOP-CC is $\mathcal{NP}$-hard*

Proof. Given the transformation above, our formal proof amounts to establishing an upper bound $UB_{good}(n, M)$ on the cumulative cost $\pi(P)$ of any "good" Hamiltonian path P as well as a lower bound $LB_{nogood}(n, M)$ on the cumulative cost $\pi(P')$ of any "non-good" Hamiltonian path P', and to show that $UB_{good}(n, M) < LB_{nogood}(n, M)$ for all n and for a value of M such that $\log(M)$ is polynomial in n.

The lower bound $LB_{nogood}(n, M)$ corresponds to the case where only one direct arc in P' has cost $2M$, hence it can be computed in a straightforward way as

$$LB_{good}(n, M) = 2M^{n-1} \tag{3.11}$$

As to upper bound $UB_{good}(n, M)$, it is computed by considering the cumulative cost of a Hamiltonian path P where all direct arcs have cost M, whereas all transitive arcs have cost $2M$. This case is illustrated in Figure 3.4. To be more precise, we claim that

$$\pi(P_n) \leq UB_{good}(n, M) := M^{n-1} + 4^n M^{n-2} \tag{3.12}$$

holds for any Hamiltonian path P_n in G whose direct arcs all have cost M, where $M > 1$ is assumed. The proof of this claim is by induction on n. The claim clearly holds in cases $n = 1$ and $n = 2$, where we have $\pi(P_1) = 1$ and $\pi(P_2) = M + 2$, respectively. We assume now that (3.12) holds for all $n \leq h$ for a given $h \geq 2$, and we prove that it also holds for $n = h + 1$. Let $P_{n=h+1} = \{(k_1, k_2), \cdots, (k_h, k_{h+1})\}$ be any Hamiltonian path whose direct arcs all have cost M, and let $P_h = \{(k_2, k_3), \cdots, (k_h, k_{h+1}\}$ and $P_{h-1} = \{(k_3, k_4), \cdots, (k_h, k_{h+1}\}$ be obtained from P_{h+1} by removing its first arc and its first two arcs, respectively. We have

$$\begin{aligned}
\pi(P_{h+1}) &= \sum_{i=1}^{h+1} \alpha_{k_i} = \sum_{i=2}^{h+1} \alpha_{k_i} + \alpha_{k_1} \\
&= \pi(P_h) + \alpha_{k_1} \\
&\leq \pi(P_h) + M\alpha_{k_2} + 1 + 2M\sum_{i=3}^{h+1} \alpha_{k_i} \quad \text{(because of (3.1) and (3.10))} \\
&\leq \pi(P_h) + M\pi(P_h) \quad (\text{since } \pi(P_h) \geq \alpha_{k_2} + 1) \\
&\qquad + 2M\,\pi(P_{h-1}) \quad (\text{since } \pi(P_{h-1}) = \textstyle\sum_{i=3}^{h+1} \alpha_{k_i}) \\
&\leq (1 + M)\pi(P_h) + 2M\pi(P_{h-1})
\end{aligned}$$

The claim then follows from the induction hypothesis, as we have

$$\begin{aligned}
\pi(P_{h+1}) &\leq (1 + M)(M^{h-1} + 4^h M^{h-2}) + 2M(M^{h-2} + 4^{h-1}M^{h-3}) \\
&\leq M^h + (3 + 4^h)M^{h-1} + \frac{3}{2}4^h M^{h-2} && (3.13) \\
&\leq M^h + (3 + \frac{5}{2}4^h)M^{h-1} \quad (\text{since } M^{h-2} \leq M^{h-1}) && (3.14) \\
&\leq M^h + 4^{h+1}M^{h-1} \quad (\text{since } h \geq 2) && (3.15)
\end{aligned}$$

To complete the complexity proof, we have to choose a value for M that guarantees $UB_{good}(n, M) < LB_{nogood}(n, M)$, i.e.,

$$M^{n-1} + 4^n M^{n-2} < 2M^{n-1} \Longrightarrow 4^n M^{n-2} < M^{n-1} \Longrightarrow 4^n < M$$

We then set $M = 4^n + 1$, whose size $log(M) = O(n)$ is polynomial in n, as required. □

Corollary 3.3.1.1. *BLOP-CC is $\mathcal{NP}$-hard*

Proof. We use the same construction as in the proof of the previous theorem, with $U := UB_{good}(n, M) = O(M^n) = O(4^{n^2})$ large enough to make all "good" Hamiltonian paths in G feasible for the BLOP-CC instance, but still with size $log(U) = O(n^2)$. □

3.4 A MIP model

In this section we introduce a MIP model for BLOP-CC, derived from the LOP model of Grötschel, Jünger and Reinelt [32].

As already mentioned, in a standard linear ordering problem we have n items to be placed in a convenient order. If we place item i before item j, we pay a cost of g_{ij}. The objective is to choose the item order that minimizes the total cost. This problem can then be modeled as

$$\begin{array}{rl} \min & \sum_{(i,j)\in A} g_{ij}x_{ij} \\ \text{subject to} & \text{“}x\text{ is the incidence vector of an acyclic tournment”} \end{array}$$

where $x_{ij} = 1$ if item i is placed before j in the final order, $x_{ij} = 0$ otherwise.

In order to get an acyclic tournament, it is shown in [32] that, besides the obvious conditions

$$x_{ij} + x_{ji} = 1, \quad \forall(i,j) \in A, i < j \tag{3.16}$$

it is sufficient to prevent 3-node cycles of the form $x_{ij} = x_{jk} = x_{ki} = 1$, leading to the *triangle inequalities*

$$x_{ij} + x_{jk} + x_{ki} \leq 2 \tag{3.17}$$

Using the same set of variables, one can rewrite the LOP-CC constraints (3.1) as the nonlinear equalities:

$$\alpha_i = p_i + \sum_{j=1}^{n} c_{ij}\alpha_j x_{ij} \quad \forall i \in V \tag{3.18}$$

In order to get linear constraints, we introduce the following $n(n-1)$ new variables:

$$y_{ij}(= \alpha_j x_{ij}) = \begin{cases} \alpha_j & \text{if } x_{ij} = 1 \\ 0 & \text{otherwise} \end{cases} \quad \forall i,j \in V : i \neq j \tag{3.19}$$

Thus (3.18) becomes linear in the y variables,

$$\alpha_i = p_i + \sum_{j=1}^{n} c_{ij} y_{ij}$$

and conditions (3.19) become

$$\begin{array}{lcl} x_{ij} = 0 \Rightarrow y_{ij} = 0 & \longrightarrow & y_{ij} \leq Mx_{ij} \\ x_{ij} = 1 \Rightarrow y_{ij} \geq \alpha_j & \longrightarrow & y_{ij} \geq \alpha_j - M(1 - x_{ij}) \\ x_{ij} = 1 \Rightarrow y_{ij} \leq \alpha_j & \longrightarrow & y_{ij} \leq \alpha_j + M(1 - x_{ij}) \end{array}$$

where M is a sufficiently large positive value. Notice that constraints $y_{ij} \leq \alpha_j + M(1 - x_{ij})$ can be removed from the model, as the minimization of variables α_j implies that of variables y_{ij}. (As to constraints $y_{ij} \leq Mx_{ij}$, they are redundant as well; however, our computational experience showed that they improve the numerical stability of the MIP solver, so we keep them into our model.) Also, from (3.19), the y variables are bounded by the α variables, thus we can take $M = U$. Finally, $\alpha \geq 0$ can be assumed since $p, c \geq 0$).

As it is customary in LOP models, one can use equations (3.16) to eliminate all variables x_{ij} with $i > j$,[2] and modify the triangle inequalities into:

$$\begin{array}{rl} x_{ij} + x_{jk} - x_{ik} \leq 1 & \text{for all } 1 \leq i < j < k \leq n \\ -x_{ij} - x_{jk} + x_{ik} \leq 0 & \text{for all } 1 \leq i < j < k \leq n \end{array}$$

This leads to the following MIP model for BLOP-CC:

$$\begin{array}{rlr} \text{minimize} & \sum_{i=1}^{n} \alpha_i & \\ \text{subject to} & x_{ij} + x_{jk} - x_{ik} \leq 1 & \text{for } 1 \leq i < j < k \leq n \\ & x_{ij} + x_{jk} - x_{ik} \geq 0 & \text{for } 1 \leq i < j < k \leq n \\ & \alpha_i = p_i + \sum_{j=1}^{n} c_{ij} y_{ij} & \text{for } 1 \leq i \leq n \\ & y_{ij} \leq U x_{ij} & \text{for } 1 \leq i < j \leq n \\ & y_{ji} \leq U(1 - x_{ij}) & \text{for } 1 \leq i < j \leq n \\ & y_{ij} \geq \alpha_j - U(1 - x_{ij}) & \text{for } 1 \leq i < j \leq n \\ & y_{ji} \geq \alpha_i - U x_{ij} & \text{for } 1 \leq i < j \leq n \\ & 0 \leq \alpha_i \leq U & \text{for } 1 \leq i \leq n \\ & y_{ij} \geq 0 & \text{for } 1 \leq i \neq j \leq n \\ & x_{ij} \in \{0, 1\} & \text{for } 1 \leq i < j \leq n \end{array}$$

3.5 An exact enumerative algorithm

Our enumerative algorithm is based on a standard backtracking technique (akin to those used in Constraint Programming) to generate all permutations and to find one with the lowest total cost. To limit the number of permutations actually evaluated, we use a pruning mechanism based on lower bounds.

Permutations are built progressively, and are extended backwards from the last node. At the root of the search tree (depth 0), none of the permutation elements in $\mathcal{K} = \langle k_1, \cdots, k_n \rangle$ is fixed. At depth 1, the last element of the permutation, k_n, is fixed to one of the n possible choices (thus, the root has n sons). At depth 2, the next to last item, k_{n-1}, is fixed to one of the remaining $n - 1$ possible choices, and so on. The search tree is visited in depth-first manner. The only required data structures to implement this method are an array to store the current partial permutation, and another to keep track of the nodes that have not yet been inserted in the current partial permutation.

We chose this method to enumerate permutations, rather than more sophisticated ones, because we can compute very quickly a parametric lower bound for partial permutations (to be used for pruning purposes), thus enumerating very effectively a large number of nodes.

[2]The α variables could be removed as well, but this would have a marginal effect on the solution time of the model.

```
Permutation generation:
  TraverseSearchTree(n)
  1.   initialize set $S := \{1, 2, \cdots, n\}$
  2.   EnumPermutationElement(S, $n$)

  EnumPermutationElement($i$)
  3.   for each element $e$ in $S$ do
  4.     perm[$i$] $\leftarrow$ $e$
  5.     evaluate partial permutation
                $\langle *, \cdots, *, \text{perm}[i], \text{perm}[i+1], \cdots, \text{perm}[n] \rangle$
  6.     if $i > 1$ then EnumPermutationElement($S \setminus \{e\}, i-1$)
  7.   enddo
```

Figure 3.5: The basic method

Our lower bound is computed as follows. Given a permutation $\mathcal{K} = \langle k_1, \cdots, k_n \rangle$, we can write the corresponding node values α_v as:

$$\begin{array}{lllllll}
\alpha_{k_n} & = & p_{k_n} & & & & \\
\alpha_{k_{n-1}} & = & p_{k_{n-1}} & + & \alpha_{k_n} \; c_{k_{n-1},k_n} & & \\
\alpha_{k_{n-2}} & = & p_{k_{n-2}} & + & \alpha_{k_n} \; c_{k_{n-2},k_n} & + & \alpha_{k_{n-1}} \; c_{k_{n-2},k_{n-1}} \\
\cdots & & & & & &
\end{array}$$

and the total permutation cost π is the sum of all the α_v. Notice that all the node weights p_v contribute to the total cost, so their sum can be used as an initial lower bound for the cost of any permutation.

Assume now that nodes $k_{i+1}, k_{i+2}, \cdots, k_n$ have already been chosen. When node k_i is also chosen, one can easily compute the corresponding α_{k_i} by using equation (3.1). Furthermore, the contribution of this α_{k_i} to the final cost of any permutation of the type $\langle *, \cdots, *, k_i, k_{i+1}, \cdots, k_n \rangle$ is given by:

$$\alpha_{k_i} \sum_{u \notin \{k_i, \cdots, k_n\}} c_{u k_i} \tag{3.20}$$

regardless of the rank of the remaining nodes in the complete permutation. This property allows us to compute easily, in a parametric way, a valid lower bound on the cost of any such permutation.

To be more specific, our pruning mechanism works as follows. We start with lower bound $LB := \sum_v p_v$ and with an empty permutation. We then build-up partial permutations recursively. Every time a new node k_i is inserted in front of the current permutation, we compute the corresponding α_{k_i} and add (3.20) to the current lower bound LB. If the resulting LB is strictly smaller than the incumbent solution value, we proceed with the recursion; otherwise we backtrack, and update LB accordingly.

A small modification of this algorithm can be used to start with a given permutation (rather than from scratch), thus allowing the enumerative search to

```
Optimization by backtracking:
  1.   find a starting heuristic solution $H$
  2.   initialize the fifo queue $Q$ with the elements in $H$
  3.   LB $\leftarrow \sum_{i=1}^{n}$ p[i]
  4.   UB $\leftarrow \infty$
  5.   EnumPermutationElement($n$, LB)
  6.   output bestperm

  EnumPermutationElement(i, LB)
  7.   repeat i times
  8.     q $\leftarrow$ pop $Q$
  9.     perm[$i$] $\leftarrow$ q
 10.     calculate alpha[q]
 11.     if i = 1 then
 12.       bestperm $\leftarrow$ perm
 13.       UB $\leftarrow$ LB
 14.     else
 15.       LB $\leftarrow$ LB +  alpha[q] * $\sum_{r \in Q}$c[r,q]
 16.       if LB<UB then EnumPermutationElement(i-1, LB)
 17.     endif
 18.     push q into $Q$
 19.   enddo
```

Figure 3.6: The overall enumerative algorithm

quickly produce improved permutations through successive modifications of the starting one. To this end, the only change required at Step 3 of the algorithm of Figure 3.5 is to implement the set S as a FIFO queue, to be initialized with the desired starting permutation (the last node in the sequence being the first to be extracted).

In our implementation we use a simple yet effective heuristic to find a “good” initial solution for the BLOP-CC instances arising in our telecommunication application. Namely, we sort the nodes by decreasing autocorrelation factors ρ_{ii}, and consider the associated permutation (the first and last node in the permutation being those with the largest and smallest ρ_{ii}, respectively) to initialize our incumbent solution. Alternative heuristics to find a good initial permutation are presented in [12].

Our complete algorithm is outlined in Figure 3.6.

3.6 Computational analysis (exact methods)

We have compared the performance of our exact enumerative algorithm with that of a state-of-the-art commercial MIP solver (`ILOG Cplex` 9.0.2 [43]) applied to the

MIP model of Section 4. The outcome of this experiment is reported in Table 3.1 and 3.2, where 4 large sets of BLOP-CC instances have been considered.

The instances used in our study have been provided by the telecommunications group of the Engineering School of the University of Padova, and are related to detection-order optimization in UMTS networks [12]. All the four communication scenarios considered in [12] have been addressed: synchronous and asynchronous transmission, with and without scrambling. For each scenario, 500 matrices (ρ_{ij}) have been randomly generated assuming the presence of 16 users[3] uniformly distributed in the cell (with a radius of 580 m), according to the so-called log-distance path loss model. The input values Γ, U, N_S, and N_0 have been set to 0.625, 10.0, 16, and 0.50476587558415, respectively. From each matrix ρ, we have derived 8 BLOP-CC instances of different sizes ($n = 2, 4, \ldots, 16$), using the equations given at the end of Section 2 to compute the weights p_v and costs c_{uv}. The full set of these matrices ρ is available for download at `http://www.math.unipd.it/~bertacco/LOPCC_instances.zip`

For each instance, the corresponding MIP model is built-up through a C++ code based on `ILOG Concert Technology 2.0` [44]. All the model constraints are statically incorporated in the initial formulation, and the outcome is solved through `ILOG Cplex` 9.0.2. Since the default `ILOG Cplex` tolerances are too large to solve correctly even small instances, we used the following parameter setting (all other parameters being left at default values):

- Absolute mipgap tolerance (CPX_PARAM_EPAGAP): 1e-13
- Relative mipgap tolerance (CPX_PARAM_EPGAP): 1e-11
- Integrality tolerance (CPX_PARAM_EPINT): 1e-9
- Optimality tolerance (CPX_PARAM_EPOPT): 1e-9
- Feasibility tolerance (CPX_PARAM_EPRHS): 1e-9

Unfortunately, in some cases these tolerances are not small enough to ensure a valid resolution, even when $n = 2$. The reason is that the ρ matrix contains coefficients that may vary by several orders of magnitude, hence even our feasibility tolerance of 1e-9 can be insufficient. On the other hand, `ILOG Cplex` does not support smaller values for this parameter, so we could not fix this pathological situation, and we had to report in Table 3.1 the number of instances that `ILOG Cplex` could not solve correctly.

Table 3.1 reports, for each group of instances and for each size n, the following information: the number of instances solved, the average solution times for both our enumerative code (*Enum*) and `ILOG Cplex` (*MIP*), the speedup of the enumerative code with respect to `ILOG Cplex`, the maximum solution times, and the number of instances not solved correctly by `ILOG Cplex` (*# fails*). Computing

[3]The UMTS technology considers up to 8-16 active users at a time; a larger number of active users is unrealistic for this application.

Set 1: Synchronous communication without scrambling

	# instances		average time (sec.s)		Enum	max time (sec.s)		
size	Enum	MIP	Enum	MIP	speedup	Enum	MIP	# fails
2	500	500	-	0.0		-	0.0	2
4	500	500	-	0.0		0.0	0.0	2
6	500	500	-	0.0		0.0	0.0	7
8	500	500	-	0.0	497	0.0	0.6	7
10	500	500	0.0	0.5	186	0.0	5.3	6
12	500	500	0.0	9.6	787	0.2	455.5	9
14	500	50	0.2	395.0	1,906	12.5	2,537.6	1
16	500	5	4.8	22,233.0	4,574	381.9	60,096.2	1

Set 2: Synchronous communication with scrambling

	# instances		average time (sec.s)		Enum	max time (sec.s)		
size	Enum	MIP	Enum	MIP	speedup	Enum	MIP	# fails
2	500	500	-	0.0		-	0.0	3
4	500	500	-	0.0		-	0.0	5
6	500	500	-	0.0		-	0.0	8
8	500	500	-	0.0		0.0	1.0	7
10	500	500	0.0	0.6	237	0.0	10.0	7
12	500	500	0.0	8.8	441	0.9	319.8	10
14	500	50	0.5	267.4	503	43.1	2,256.9	1
16	500	4	15.0	14,473.2	964	794.6	49,424.0	1

Set 3: Asynchronous communication without scrambling

	# instances		average time (sec.s)		Enum	max time (sec.s)		
size	Enum	MIP	Enum	MIP	speedup	Enum	MIP	# fails
2	500	500	-	0.0		-	0.0	2
4	500	500	-	0.0		-	0.0	3
6	500	500	-	0.0		-	0.0	6
8	500	500	-	0.1		0.0	0.5	4
10	500	500	0.0	1.2	1,212	0.0	13.1	4
12	500	500	0.0	30.0	2,447	0.2	786.6	10
14	500	50	0.2	1,130.8	5,218	11.9	8,058.2	0
16	500	5	5.6	28,191.6	4,978	407.8	61,402.8	1

Set 4: Asynchronous communication with scrambling

	# instances		average time (sec.s)		Enum	max time (sec.s)		
size	Enum	MIP	Enum	MIP	speedup	Enum	MIP	# fails
2	500	500	-	0.0		0.0	0.0	3
4	500	500	-	0.0		0.0	0.0	2
6	500	500	-	0.0		0.0	0.0	3
8	500	500	-	0.1		0.0	0.7	4
10	500	500	0.0	1.3	1,096	0.0	15.8	4
12	500	500	0.0	31.3	2,310	0.2	473.4	8
14	500	50	0.2	1,801.5	6,880	6.6	35,469.5	1
16	500	2	6.3	8,362.0	1,316	204.8	9,821.5	1

Table 3.1: Computational analysis of the exact methods

Overall statistics

	# instances		average time (sec.s)		Enum	max time (sec.s)		
size	Enum	MIP	Enum	MIP	speedup	Enum	MIP	# fails
2	2000	2000	-	0.0		0.0	0.0	10
4	2000	2000	-	0.0		0.0	0.0	12
6	2000	2000	-	0.0		0.0	0.0	24
8	2000	2000	-	0.1	1,146	0.0	1.0	22
10	2000	2000	0.0	0.9	488	0.0	15.8	21
12	2000	2000	0.0	20.0	1,373	0.9	786.6	37
14	2000	200	0.3	898.7	2,952	43.1	35,469.5	3
16	2000	16	7.9	20,421.2	2,562	794.6	61,402.8	4

Table 3.2: Overall statistics for the exact methods

times are expressed in CPU seconds, and refer to a Pentium M 1.4 Ghz notebook with 512 MBytes of main memory.

The computational results clearly show that our enumerative approach outperforms `ILOG Cplex`, and is up to three orders of magnitude faster. As a matter of fact, in no instance `ILOG Cplex` beat the enumerative code. As to scalability, the enumerative code proved capable of solving instances with $n = 20$ in about 3 hours.

All the instances in our testbed were solved to proven optimality, thus enabling us to benchmark the GRASP heuristic proposed in [12] (evaluating the performance of this method was indeed our initial motivation in studying BLOP-CC).

3.7 Heuristics

We propose next a heuristic dynamic programming solution scheme, based on the observation that the classic min-cost Hamiltonian path problem on small graphs can be solved very effectively by using dynamic programming [73]. Indeed, for the min-cost HP problem, a suitable dynamic programming state is defined as the pair (S, h) (see Figure 3.7), where S is a nonempty node set and $h \in S$. The optimal cost $T(S, h)$ of a path starting from node h and visiting exactly once all the nodes in S can then be computed through the recursion

$$T(S, h) = \min\{d(h, i) + T(S \setminus \{h\}, i) : i \in S \setminus \{h\}\},$$

where $d(h, i)$ is the cost of the arc (h, i), and $T(\{v\}, v) = 0$ for all $v \in V$. It is then quite natural to adapt the above recursion to LOP-CC, by simply reinterpreting values $T(S, h)$ as the lower bound described in Section 4. To be more specific, one can think of initializing $T(\{k\}, k) = \sum_{v \in V} p_v$ and of using equation (3.20) instead of $d(h, i)$ in the recursion formula. In this way, $T(S, h)$ represents an estimate of the contribution that the trailing part of a path starting from h and covering exactly once all the nodes in S, will give to the overall

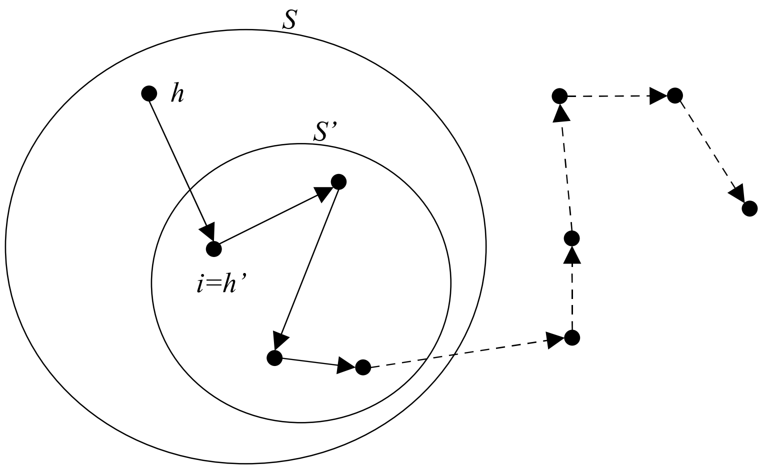


Figure 3.7: Dynamic programming state and recursion

cumulative cost – regardless of how it is extended by adding nodes before h. Since the computation of the lower bounds takes linear time and there are $n2^{n-1}$ states, the overall complexity is $O(n^2 \cdot 2^n)$, hence it is affordable for the small-size graphs arising in our telecommunication application.

Unfortunately, differently from the HP problem, the above recursion applied LOP-CC does not guarantee the optimality of the resulting path. The reason is that the cost of adding a node in front of a given path $T(S, h)$, given by equation (3.20), is not independent from the order of the nodes in S (because of the a_{k_i} term). Therefore, for LOP-CC the above simple dynamic-programming approach is not exact, although in many practical cases it can still be interesting as a heuristic method.

Table 3.3 analyzes the quality of the heuristic solutions found by the dynamic programming heuristic (DP) over the same set of instances considered in the bottom part of Table 1 (2000 random instances for each size).

The first column in the table gives the size of the problems. The second column reports the number of instances for which DP found an optimal solution, while the next two columns report the average and maximum gap of the solution found by DP with respect to the optimal one (found by our exact enumerative method, Enum). The final two columns compare the average CPU times of the DP and Enum algorithms. For the instance of size 20, mentioned above, DP found a solution with a gap of 0.1% in 46 seconds (versus 3 hours taken by the Enum).

In order to better evaluate our dynamic-programming heuristic, we compared it with a GRASP heuristic for UMTS applications akin to that proposed in [12] (but with a slightly better performance for the instances in our testbed), that works as follows.

The Greedy Randomized Adaptive Search Procedure (GRASP) is based on a greedy approach performing n steps to build the ordered set $\mathcal{K}$. At each step a user (i.e., node) is inserted in the current sequence $\mathcal{K}$, starting from the last

size	instances solved to optimality by DP	average gap	maximum gap	average time by Enum (sec.s)	average time by DP (sec.s)
4	2000 / 2000	0%	0%	-	-
6	1990 / 2000	0.001%	0.74%	-	-
8	1901 / 2000	0.01%	1.24%	-	-
10	1734 / 2000	0.05%	4.38%	0.001	0.002
12	1457 / 2000	0.14%	6.99%	0.014	0.014
14	1146 / 2000	0.27%	7.15%	0.305	0.097
16	852 / 2000	0.47%	11.75%	7.972	0.752

Table 3.3: Performance of the dynamic programming heuristic

position k_n, and coming back up to the first one k_1. As in the enumerative approach, building $\mathcal{K}$ in reverse order allows one to easily compute the power α_{k_i} associated with a user as soon as it is inserted in $\mathcal{K}$ (by using equation (3.1)).

In order to perform the choice of the user to be inserted at each step i, the *minimum power criterion* is used. With this criterion, for each user k not yet in $\mathcal{K}$, we compute the corresponding power received at the detection point as

$$P_k^{(i)} = \alpha_k^{(i)} \rho_{kk} = (p_{k_i} + \sum_{j=i+1}^{n} c_{k_i k_j}) \rho_{kk}. \tag{3.21}$$

This figure gives the power that the user k would be assigned, if selected for the position i in the cancelation order. The user yielding the *minimum* power is then determined as

$$k_i = \text{argmin}_{k \notin \mathcal{K}} \{P_k^{(i)}\}. \tag{3.22}$$

Using a GRASP approach, several locally-optimal solutions are examined and the solution achieving the best performance is selected. Let N_{Max} by an integer parameter. The GRASP algorithm still involves K steps. At each step i of the algorithm, let $W^{(i)} = \{w_1^{(i)}, w_2^{(i)}, \ldots, w_{N_{\text{Max}}}^{(i)}\}$ be the N_{Max} user indices not yet in $\mathcal{K}$ and having the lowest powers $P_k^{(i)}$. The user k_i to be inserted in the set $\mathcal{K}$ is chosen randomly from the set $W^{(i)}$. Note that the GRASP algorithm for $N_{\text{Max}} = 1$ yields the greedy solution, as the user with the lowest $P_k^{(i)}$ is always selected. In general, because of the random choice of the users, the algorithm can provide different solutions for each run. In practice, the GRASP algorithm is iterated I times (say), plus one for the pure greedy run, and the best solution is selected.

Table 3.4 compares the GRASP and the DP heuristics over the same set of 2,000 instances considered in the previous experiments. For this benchmark, we ran GRASP with $N_{\text{Max}} = 3$ for a number of iterations I such that GRASP heuristics takes the same CPU running time as the DP one. The first two columns in the table give the size of the problems and the number of GRASP iterations (I). The third and fourth columns report, respectively, the average and maximum gap of the GRASP solutions with respect to the optimal ones. Finally the last two columns show the difference between these gaps and those obtained with our DP method (as shown in the previous table).

As a comparison, running the original GRASP heuristic described in [12] on all the instances of size 16, yields an average gap of 9.48% and a maximum gap of 66.16%.

size	iters I	average gap	maximum gap	Δ average gap wrt DP	Δ maximum gap wrt DP
4	12	0.003%	2.631%	+0.003%	+2.631%
6	48	0.243%	9.643%	+0.242%	+8.903%
8	198	0.430%	8.470%	+0.420%	+7.230%
10	862	0.941%	21.698%	+0.891%	+17.318%
12	3936	1.364%	26.251%	+1.224%	+19.261%
14	20074	1.874%	27.921%	+1.604%	+20.771%
16	116506	2.620%	23.623%	+2.150%	+11.873%

Table 3.4: Performance of the GRASP heuristic

The table shows that DP heuristic performs considerably better than the GRASP method. Also, for an instance of size 20, DP found a solution with a gap of 0.100%, GRASP reached a gap of 0.415% (within the same computing time), whereas for an instance of size 21, DP found a solution 0.31% better than the GRASP one.

3.8 Conclusions

We have introduced and studied, for the first time, a new optimization problem related to the well-known Linear Ordering Problem, in which the solution cost is non-linear due to a cumulative backwards propagation mechanism. This model was motivated by a practical application in UMTS mobile-phone telecommunication system.

We have formalized the problem, in two versions, and proved that they are both $\mathcal{NP}$-hard. We have proposed a Mixed-Integer Linear Programming model as well as an ad-hoc enumerative algorithm for the exact solution of the problem. A dynamic-programming heuristic has been also described. Extensive computational results on large sets of instances have been presented, showing that the proposed techniques are capable of solving, in reasonable computing times, all the instances coming from our application. As a byproduct, our method allowed to benchmark the simple GRASP heuristic proposed in [12].

Future research should be devoted to enhancing the MIP formulation, and/or to embed more sophisticated pruning mechanisms in our enumerative scheme. Also worth studying are more complex (nonlinear) cost functions applied to the basic Linear Ordering model.

Chapter 4

Branching on General Disjunctions

4.1 Introduction

As discussed in Chapter 1, branch-and-bound algorithms use *branching* to split the feasible region of a problem into disjoint subsets that are both "easier" to solve, and cut off the current LP optimal vertex. In other words, branching means to impose a disjunction that is valid for all feasible (integer) solutions, but not the current fractional LP solution.

The choice of the branching object is traditionally an individual variable, chosen among the most fractional ones in the current basic solution (see Section 1.5.1). If x_i is an integer-constraint variable, with a fractional value x_i^* in the current solution, then we can impose the constraint $x_i \leq \lfloor x_i^* \rfloor$ for one child and $x_i \geq \lceil x_i^* \rceil$ for the second child (*variable dichotomy*).

More generally, we can consider branching on arbitrary integer vectors by imposing a *general (or* split*) disjunction* [17] of the form

$$\alpha x \leq \alpha_0 \quad \bigvee \quad \alpha x \geq \alpha_0 + 1 \tag{4.1}$$

where $(\alpha, \alpha_0) \in \mathbb{Z}^{n+1}$ (for an integer linear program with n variables,). Variable dichotomy can therefore be seen as a disjunction where the vector α is the i-th elementary vector and $\alpha_0 = \lfloor x_i^* \rfloor$ (*simple disjunction*).

The main reason for studying general disjunctions is that they can lead to a drastic reduction of the branching tree size. While branching on a disjunction implies adding a new constraint at each level, thus making the LP subproblems increasingly more complex, the decrease in tree size should more than offset this effect.

One difficulty with the application of this idea, however, is how to choose, at each node of the branching tree, the "best" disjunctions among the infinite space of integer vectors. In [61], Owen and Mehrotra proposed to branch on general disjunctions generated by a neighborhood search heuristic. The neighborhood contains all disjunctions with coefficients in $\{-1, 0, 1\}$ on the integer variables with fractional values at the current node. The quality of the disjunctions is evaluated by solving the children nodes in the spirit of strong branching. More recently, Karamanov and Cornuéjols [46] considered a specific class of general disjunctions, the ones defining mixed integer Gomory cuts derived from the tableau, and select the most promising disjunctions based on a heuristic measure of disjunction quality (given by the distance cut off by the corresponding cut). Their experiments show that branching on general disjunctions is more efficient than branching on variables, based both on solution time and tree size.

In this chapter we propose two different approaches to branching on general disjunctions, and study the impact that they can have on the branching tree.

4.2 Slicing

Our first approach is inspired by the algorithms for integer programming in fixed dimension. Given a full-dimensional convex body $K \subseteq \mathbb{R}^n$, the *width of* K *along a direction* $c \in \mathbb{R}^n$ is the quantity $w_c(K) := \max\{c^T x | x \in K\} - \min\{c^T x | x \in K\}$. The *width* of K, $w(K)$ is the minimum of its widths along nonzero integral vectors $c \in \mathbb{Z}^n \backslash \emptyset$.

If K does not include any lattice points, then K must be "flat", meaning that there is a constant f_n (depending only on the dimension n) bounding the width of K. This fact is know as Kinchin's *flatness theorem* (see [45]) and is exploited in Lenstra's algorithm [48, 35] for the integer feasibility problem as follows. If one has to decide whether a full-dimensional polyhedron $P = \{x \in \mathbb{R}^n | Ax \leq b\}$ is integer feasible or not, one computes an integral direction $c \in \mathbb{Z}^n \backslash \emptyset$ along which P is not much wider than its (minimum) width[1]: $w(P) \leq w_c(P) \leq \gamma w(P)$. If $w_c(P)$ is larger than γf_n, then P must contain integer points by the flatness theorem. Otherwise, an integer point of P must lie in one of the constant number of $(n-1)$-dimensional polyhedra $P \cap (c^T x = \delta)$ for δ integer in the range $[\min\{c^T x | x \in P\}, \max\{c^T x | x \in P\}]$. In the original space, this corresponds to branching on a general disjunction, with one child node for each possible value of δ.

In our first approach, we keep the idea of reducing the dimension of (at least one of the) subproblems, but depart from this strategy in two main aspects. First we avoid a multi-term disjunction where a child is created for each sectioning hyperplane; instead we create only two subproblem, one associated to the hyperplane closer to the LP fractional optimum, and the other associated to all further hyperplanes. Moreover, rather than looking for a "thin" direction, we want the first hyperplane to be a valid cut and look for a direction that maximizes the LP bound at the right son.

[1]This direction can be computed by lattice basis reduction based on the work of Lenstra, Lenstra, and Lovász [49]

4.2.1 A MIP model

We consider the Integer Linear Program problem

$$(IP) \quad \min\{c^T x : Ax \leq b, x \geq 0 \text{ integer}\} \tag{4.2}$$

where A is a $m \times n$ matrix, $b \in \mathbb{R}^m$, and $c \in \mathbb{R}^n$, along with its linear relaxation

$$(LP) \quad \min\{c^T x : Ax \leq b, x \geq 0\}$$

and the two associated polyhedra:

$$P := \{x \in \mathbb{R}^n_+ : Ax \leq b\} \tag{4.3}$$
$$P_I := conv\{x \in \mathbb{Z}^n_+ : Ax \leq b\} = conv(P \cap \mathbb{Z}^n) \tag{4.4}$$

Given a fractional solution x^* of (LP), we want to find a valid Chvátal-Gomory cut $\alpha^T x \leq \alpha_0$ that cuts off x^*, and then branch on the disjunction

$$\alpha^T x = \alpha_0 \bigvee \alpha^T x \leq \alpha_0 - 1. \tag{4.5}$$

. In particular, we would like the *right-son lower bound*

$$LB_r := \min\{c^T x : Ax \leq b,\ \alpha^T x \leq \alpha_0 - 1,\ x \geq 0\}$$

be as large as possible (eventually greater or equal to the value of the incumbent solution). This problem can be rephrased as

(i) $\alpha^T x < \alpha_0 + 1$ valid for P, with (α, α_0) integer
which imposes that $\alpha^T x \leq \alpha_0$ is a valid Chvátal-Gomory cut for (MIP),

(ii) $\alpha^T x^* > \alpha_0$
to cut off the current (fractional) LP optimum x^*, and

(iii) the linear system $\{x \in P,\ \alpha^T x \leq \alpha_0 - 1,\ c^T x < LB_r\}$ is infeasible, for LB_r as large as possible

Our approach is to model this optimization problem as a mixed integer program which is then solved through a general-purpose MIP solver. In order to model condition (i) and (iii) through linear constraints we use the Farkas' Lemma (see for example [22]):

Lemma 4.2.1 (Farkas' Lemma)**.** *The inequality $\alpha^T x \leq \alpha_0$ is valid for the non empty polyhedron $P := \{x \geq 0 : Ax \leq b\}$ if and only if there exists a vector $u \geq 0$ such that*

$$\alpha^T \leq u^T A, \quad \alpha_0 \geq u^T b. \tag{4.6}$$

Equivalently, P has no solution if and only if there exists $u \geq 0$ such that

$$u^T A \geq 0, \quad u^T b < 0. \tag{4.7}$$

By (4.6), condition (i) becomes

$$\exists\ u \geq 0 : \alpha^T \leq u^T A, \alpha_0 + 1 > u^T b.$$

As to condition (iii), again by Farkas (4.7) it becomes

$$\exists\ (\mu, \lambda, \lambda') \geq 0 : \mu^T A + \lambda' \alpha^T + \lambda c^T \geq 0^T,\ \mu^T b + \lambda'(\alpha_0 - 1) + \lambda(LB_r - \epsilon) < 0$$

for a certain very small positive value ϵ. Assuming w.l.o.g. $\lambda' > 0$ we can set $\lambda' = 1$, thus (iii) can be expressed as

$$\exists(\mu, \lambda) \geq 0 : \mu^T A + \alpha^T + \lambda c^T \geq 0^T,\ \mu^T b + (a_0 - 1) + \lambda(LB_r - \epsilon) < 0. \quad (4.8)$$

Note that the above condition can also be derived, perhaps more directly, as follows. By LP duality, and assuming the right-son LP relaxation is feasible, $LB_r = \min\{c^T x : -Ax \geq -b, -\alpha^T x \geq -(\alpha_0 - 1), x \geq 0\} = \max\{-w^T b - \sigma(\alpha_0 - 1) : c^T + w^T A + \sigma\alpha^T \geq 0^T, (w, \sigma) \geq 0\}$, so there exists a dual-feasible solution (w, σ) such that $LB_r \leq -w^T b - \sigma(\alpha_0 - 1)$. Assuming w.l.o.g. $\sigma > 0$ (otherwise the branching condition would be ineffective and LB_r cannot improve the father-node bound), dividing by σ the conditions above, and making the substitutions $\lambda = 1/\sigma$ and $\mu = w/\sigma$, we obtain (4.8).

Thus we obtain our final (nonlinear because of term λLB_r) MIP model:

$$\begin{aligned}
\max \quad & LB_r && (4.9)\\
& \alpha^T \leq u^T A && (4.10)\\
& u^T b - \alpha_0 \leq 1 - \epsilon' && (4.11)\\
& \alpha^T x^* - \alpha_0 \geq \delta && (4.12)\\
& \mu^T A + \alpha^T + \lambda c^T \geq 0^T && (4.13)\\
& \mu^T b + (a_0 - 1) + \lambda(LB_r - \epsilon) \leq -\epsilon' && (4.14)\\
& u \geq 0,\ \mu \geq 0,\ \lambda \geq 0 && (4.15)\\
& (\alpha, \alpha_0) \text{ integer} && (4.16)
\end{aligned}$$

In order to get a standard MIP model, one can approximate LB_r by a weighted sum of binary variables, and then linearize λLB_r according to the following construction. In our model, the only values of interest for LB_r are between the LP relaxation bound at the current node, say LB, and the value of the incumbent solution, say UB. Hence we can write $LB_r := LB + GAP \cdot \sigma$, where $GAP := UB - LB$ and $\sigma \in [0, 1]$. Now we approximate σ with b binary variables $z_1, z_2, \ldots, z_b$: $\sigma \approx \sum_{h=1}^{b-1} 2^{-h} z_h + 2^{1-b} z_b$, so as to be able to obtain $\sigma = 0$ (for $z_h = 0$ for all h) and $\sigma = 1$ (for $z_h = 1$ for all h). Then we write $\lambda LB_r = \lambda LB + \lambda \cdot GAP \cdot \sigma \approx \lambda LB + GAP\lambda(\sum_{h=1}^{b-1} 2^{-h} z_h + 2^{1-b} z_b) = \lambda LB + GAP(\sum_{h=1}^{b-1} 2^{-h}\gamma_h + 2^{1-b}\gamma_b)$ where $\gamma_h := \lambda z_h$ is linearized through the constraints

$$-Mz_h \leq \gamma_h \leq Mz_h,\ \ \lambda - M(1 - z_h) \leq \gamma_h \leq \lambda + M(1 - z_h) \quad \forall h = 1, \ldots, b \ (4.17)$$

where M is a sufficiently large value. The final model reads:

$$(SM) \qquad \max \quad LB + GAP(\sum_{h=1}^{b-1} 2^{-h} z_h + 2^{1-b} z_b) \tag{4.18}$$
$$\alpha^T \leq u^T A \tag{4.19}$$
$$u^T b - \alpha_0 \leq 1 - \epsilon' \tag{4.20}$$
$$\alpha^T x^* - \alpha_0 \geq \delta \tag{4.21}$$
$$\mu^T A + \alpha^T + \lambda c^T \geq 0^T \tag{4.22}$$
$$\mu^T b + \alpha_0 + \lambda(LB - \epsilon) + GAP(\sum_{h=1}^{b-1} 2^{-h} \gamma_h + 2^{1-b} \gamma_b) \leq 1 - \epsilon' \tag{4.23}$$
$$\lambda - M(1 - z_h) \leq \gamma_h \tag{4.24}$$
$$u \geq 0, \ \mu \geq 0, \ \lambda \geq 0 \tag{4.25}$$
$$\gamma_h \geq 0 \qquad \forall h = 1, \ldots, b \tag{4.26}$$
$$z_h \in \{0, 1\} \qquad \forall h = 1, \ldots, b \tag{4.27}$$
$$(\alpha, \alpha_0) \text{ integer} \tag{4.28}$$

where the objective function can of course be replaced by the simpler term $\sum_{h=1}^{b-1} 2^{-h} z_h + 2^{1-b} z_b$. Note also that some of the inequalities (4.17) are not constraining and have been removed.

In addition, one can also allow for the presence of continuous variables. Indeed, if a variable x_j is not restricted to be integer, one can still derive a valid cut by setting $\alpha_j = 0$ and imposing $u^T A_j \geq 0$ (or $u^T A_j = 0$ in case x_j is a free variable).

In section 4.4, we evaluate a branching strategy that uses, at each node, (eventually heuristic) solutions of the above MIP to obtain effective branching disjunctions of the type (4.5). To improve the optimization of this model, before solving it, we also look for valid CG cuts to pass to the solver as incumbent solutions, so that local search heuristics can be activated. These cuts are obtained either from the current optimal tableau or from a reduced version of the above model where only inequalities (4.19) and (4.20) are kept and the violation (4.21) is set as the objective function (see also [25]).

4.3 Maximizing the son-node bounds

Another approach is to search for a disjunction of the following type

$$\alpha^T x \leq \alpha_0 - 1 \bigvee \alpha^T x \geq \alpha_0 \tag{4.29}$$

with (α, α_0) integer, that produces the largest LP bound in both sides of the disjunction. More precisely we try to maximize the minimum of the two son-node bounds that would result from this disjunction.

This is also the rule commonly used for choosing a branching variable when using simple disjunctions. Specifically, let x_1^* and x_2^* be the optimal solutions for the left and right child, respectively, and let $z_1^* = c^T x_1^*$ and $z_2^* = c^T x_2^*$ be the corresponding objective values. In order to choose a branching variable, the typical criteria is to maximize a function of z_1^* and z_2^*, usually $\min(z_1^*, z_2^*)$ or, sometimes, a weighted sum of this min and the average $\frac{1}{2}[z_1^* + z_2^*]$ (see [4, 51], for instance).

Exploiting the Farkas' Lemma as before, this problem can be rephrased as the problem of finding the maximum value LB^* such that there exist (α, α_0) integer such that:

(a) the linear system $Ax \le b,\ \alpha^T x \le \alpha_0 - 1,\ c^T x \le LB^* - \epsilon,\ x \ge 0$ is infeasible, where ϵ is a very small positive value

(b) the linear system $Ax \le b,\ -\alpha^T x \le -\alpha_0,\ c^T x \le LB^* - \epsilon,\ x \ge 0$ is infeasible, where ϵ is a very small positive value

(c) $\alpha_0 - 1 < \alpha^T x^* < \alpha_0$ so that the current fractional LP optimum x^* is cut off by the disjunction.

We are interested only in disjunctions that improve the current node LP bound, LB. If we can't find any such disjunction, then it seems more reasonable to use the simpler branching on variables. On the other hand, whenever the bound can be improved, with $LB^* > LB$, then the current node optimum x^* is certainly cut-off and condition (c) is redundant. This leads to the MIP model:

$$\max \quad LB^* \tag{4.30}$$
$$\mu^T A + \alpha^T + \lambda c^T \ge 0^T \tag{4.31}$$
$$\mu^T b + \alpha_0 + \lambda(LB^* - \epsilon) \le 1 - \epsilon' \tag{4.32}$$
$$w^T A - \alpha^T + \lambda' c^T \ge 0^T \tag{4.33}$$
$$w^T b - \alpha_0 + \lambda'(LB^* - \epsilon) \le -\epsilon' \tag{4.34}$$
$$\mu \ge 0, \quad w \ge 0, \quad \lambda \ge 0, \quad \lambda' \ge 0 \tag{4.35}$$

In order to optimize this model with a general-purpose MIP solver, we need to handle the two non-linear terms λLB^* and $\lambda' LB^*$ in constraints (4.32) and (4.34), respectively. Rather than linearize both of them through discretization as for the (SM) model, which would make the problem much harder to solve, it seems more convenient to use an external binary search for the best value of LB^* for which the model is feasible. More precisely we perform an n-steps binary search for the best value of LB^*, in the range $[LB, LB + GAP]$ (with LB and GAP defined as before) for which we can find a disjunction. At each step we solve the following model

$$
\begin{aligned}
(BM) \quad \min \quad & \Delta && (4.36)\\
& \mu^T A + \alpha^T + \lambda c^T \geq 0^T && (4.37)\\
& \mu^T b + \alpha_0 + \lambda(LB^* - \epsilon) - \Delta \leq 1 - \epsilon' && (4.38)\\
& w^T A - \alpha^T + \lambda' c^T \geq 0^T && (4.39)\\
& w^T b - \alpha_0 + \lambda'(LB^* - \epsilon) - \Delta \leq -\epsilon' && (4.40)\\
& \mu \geq 0, \quad w \geq 0, \quad \lambda \geq 0, \quad \lambda' \geq 0, \quad \Delta \geq 0 && (4.41)
\end{aligned}
$$

where inequalities (4.32) and (4.34) have been relaxed and their violation moved to the objective function, so we only accept solutions with $\Delta = 0$. Finally, since it is often impossible to close even a small fraction of the gap and in order to contain the computing time, we don't start the binary search with a value of LB^* in the middle of gap, but first try with the smallest candidate value (namely, $LB + GAP/2^n$) and then continue with the standard binary search.

4.4 Computational experiments

In this section we compare the number of nodes explored by a state-of-the-art commercial solver, namely, `ILOG Cplex` 9.1 [43] using three different branching strategies: `ILOG Cplex` default variable branching, and the two strategies presented in the previous sections.

Our aim is to establish if either or both the proposed strategies can lead to a decrease of the search tree size and to what extent.

The set of instances used for this experiment is taken from the MIPLIB2 and MIPLIB3 libraries [57] and from a set of randomly generated knapsack problems (instances `kp80`, `kp100`, and `mkp18_3`). The instances have been selected according to this criteria: they involve only integer variables[2], and a feasible solution for the (SM) model at the root node could be found within 5 minutes of CPU time.

Preliminary results of our experiments are reported in Tables 4.1 and 4.2. The focus is on the total number of nodes processed to prove optimality. The tables show the name of the instance in the first column, followed by the number of nodes processed by the solver with default branching, and then by the number of nodes processed using the disjunctions obtained either with the (SM) model (Table 4.1) or the (BM) model (Table 4.2). `ILOG Cplex` was used both as the external branch-and-bound framework and as the MIP solver for the (SM) and (BM) models. The number of nodes processed by `ILOG Cplex` with its default branching was obtained running it with all parameters left to the default values except for the `integrality tolerance` set to 0. In particular, both pre-solving and cut generation were left enabled.

[2]The `misc*` models actually have 1 continuous variable, which just defines the objective function and can be easily substituted and removed.

In Table 4.1, the *slicing branching* columns report the number of nodes processed (and its percentage with respect to `ILOG Cplex` default) for the case of branching on the disjunctions obtained with the (SM) model. For this experiment, the (SM) model was solved at each node of the first 11 levels of the branching tree (i.e., for node depth from 0 to 10) and with a time-limit of 10 minutes.

Name	cplex branching	slicing branching		dis. bra.	tot. bra.
bm23	103	30	29.13%	26	26
enigma	8,790	11,930	135.72%	12	12
kp80	89,874	89,682	99.79%	3	13
l152lav	676	868	128.46%	17	18
lseu	133	22	16.54%	19	19
misc01	516	488	94.57%	63	63
misc02	40	37	92.50%	17	17
misc03	1,215	628	51.69%	57	57
misc07	72,266	65,568	90.73%	45	45
mkp18_3	894,474	409,411	45.77%	45	47
mod008	380	9	2.37%	9	9
mod010	10	2	20.00%	1	2
p0201	131	543	414.50%	61	61
pipex	17	8	47.06%	7	8
stein9	9	5	55.56%	3	3
		geometric mean	53.35%		

Table 4.1: Comparison of tree size for slicing branching

Whenever a feasible solution could be found within this time-limit, branching was performed using the general disjunction given by the best solution. For nodes where no solution to the (SM) model could be found, and for nodes deeper than level 10, the solver default branching was used. The last two columns of the table show the number of times that branching on a general disjunction has been possible, and the total number of branching in the first 11 levels of the tree. These correspond, respectively, to the number of (SM) models for which a feasible solution was found, and the total number of (SM) models solved. Parameters ϵ, ϵ' and δ were set all to 10^{-3}. The number b of binary variables used to discretize the non-linear term λLB_r was 4, thus splitting the gap in 8 intervals. Finally, for solving the (SM) models, `ILOG Cplex` was run with all settings left to the default values except for the `integrality tolerance` set to 0, `feasibility tolerance` set to 10^{-9}, and `MIP emphasis` set to 4 (i.e., *Emphasize hidden feasible solutions*).

Table 4.2 is organized similarly and reports the results for the case of branching on disjunctions obtained with the (BM) model. This model was solved with the same parameters described above, except that, in this case, a 3-step binary search is used in place of the discretization, each step with a time-limit of 1 minute.

All tests were performed on an Intel Pentium IV 2.4GHz personal computer with 512 Mbyte of main memory.

Name	cplex branching	best bound branching		dis. bra.	tot. bra.
bm23	103	107	103.88%	28	37
enigma	8,790	8,790	100.00%	0	6
kp80	89,874	25,565	28.45%	6	17
l152lav	676	264	39.05%	12	27
lseu	133	32	24.06%	11	15
misc01	516	423	81.98%	16	21
misc02	40	13	32.50%	6	8
misc03	1,215	1227	100.99%	7	12
misc07	72,266	68,639	94.98%	6	21
mkp18_3	894,474	894,474	100.00%	0	31
mod008	380	4	1.05%	2	2
mod010	10	10	100.00%	0	6
p0201	131	137	104.58%	10	17
pipex	17	4	23.53%	2	2
stein9	9	10	111.11%	2	5
		geometric mean	48.68%		

Table 4.2: Comparison of tree size for best bound branching

The tables show that, with the (SM) model, the number of nodes processed decreased, sometimes drastically, in 12 instances, and increased in 3 out of 18 instances. On (geometric) average, by using the (SM) model the resulting tree size was 53.35% of that generated with default branching, even applying this model only in the first few levels of the tree. In particular, on instances `mod008` and `mod010`, the disjunctions found always managed to close all the gap (for the right branch), thus reducing the tree size more effectively. In the cases were the tree size increased, this seems to be caused both by the unbalanced nature of this approach and by an higher difficulty in finding feasible solutions (and then update the incumbent solution) when branching on general disjunctions.

The overall performance of the (BM) model was even better, with tree sizes reduced, on average, to 48.68%, and never increased considerably. Here it can be noticed that, for 3 instances, no general disjunctions could be found at any node (so the tree size is unchanged). In the case of instance `enigma` this is because its objective function can't effectively guide the optimization since the LP bound at the root node is equal to the optimal value; hence the (BM) model can never find a disjunction that improves the bound at both sons. For the other 2 cases, it is possible that no disjunctions closing one eighth of the gap existed or the time-limit was to low to find them.

Finally, times are not shown here since, at this stage, our focus was only on the tree size. Clearly both our strategies involved solving MIP models that are quite difficult, thus leading to optimization times in the order of several minutes and artificially limited by the depth and time limits imposed. `ILOG Cplex` could solve most of these instances in few seconds.

4.5 Conclusions

In this chapter we proposed two new strategies for branching on general disjunctions within branch-and-cut algorithms for Mixed Integer Programming problems. Both strategies are independent of specific problem structures and can be applied to any MIP. The first strategy aims at a left son subproblem of lower dimension and a right son subproblem with a large LP bound improvement, while the second one looks for the largest possible bound improvement in both sons.

Both strategies model the problem of choosing the best disjunction as a Mixed Integer Problem, which is then solved through a general purpose MIP solver with a truncated search.

Preliminary tests of these ideas, comparing branching on variables with branching on general disjunctions, over a set 15 instances with quite different structures, show that, on average, branching on general disjunctions can effectively reduce considerably the size of the branching tree.

Future directions of research include both improving the proposed MIP models and their solution scheme, in order to evaluate their effects at deeper levels of the branching tree and on a larger set of test problems. Also, another interesting question that is going to be investigated is, given an optimal MIP solution, how often is it possible to close all the gap between a node LP bound and the optimal objective value with only one disjunction? We believe that these can provide very useful new insight to the fundamental topic of branching.

Bibliography

[1] 3GPP, Technical Specification Group Radio Access Networks; Radio Transmission and Reception. 3G TS 25.102 version 3.6.0, 1999.

[2] T. Achterberg. SCIP - a framework to integrate Constraint and Mixed Integer Programming, Technical Report 04-19, Zuse Institute Berlin, 2004 (available at `http://www.zib.de/Publications/abstracts/ZR-04-19/`).

[3] T. Achterberg and T. Berthold. Improving the feasibility pump. Technical Report Zuse Institute Berlin, September 2005.

[4] T. Achterberg, T. Koch and A. Martin. Branching rules revisited. *Operation Research Letters*, 33:42–54, 2005.

[5] T. Achterberg, T. Koch and A. Martin. MIPLIB 2003. Technical Report 05-28, Zuse Institute Berlin, 2005 (available at `http://www.zib.de/PaperWeb/abstracts/ZR-05-28/`).

[6] E. Balas. Facets of the knapsack polytope. *Mathematical Programming*, 8:146–164, 1975.

[7] E. Balas, S. Ceria, G. Cornuéjols and N. R. Natraj. Gomory cuts revisited. *Operations Research Letters*, 19:1–9, 1996.

[8] E. Balas, S. Ceria, M. Dawande, F. Margot and G. Pataki. OCTANE: A New Heuristic For Pure 0-1 Programs. *Operations Research* 49, 207–225, 2001.

[9] E. Balas and C. H. Martin. Pivot-And-Complement: A Heuristic For 0-1 Programming. *Management Science* 26, 86–96, 1980.

[10] E. Balas, S. Schmieta and C. Wallace. Pivot and Shift-A Mixed Integer Programming Heuristic. *Discrete Optimization* 1, 3–12, 2004.

[11] E. Balas and E. Zemel. Facets of the knapsack polytope from minimal covers. *SIAM Journal of Applied Mathematics*, 34:119–148, 1978.

[12] N. Benvenuto, G. Carnevale and S. Tomasin. Joint Power Control and Receiver Optimization of CDMA Transceivers using Successive Interference Cancelation. Technical Report DEI, Department of Information Engineering, University of Padova, 2004.

[13] N. Benvenuto and S. Tomasin. On the Comparison Between OFDM and Single Carrier Modulation With a DFE Using a Frequency-Domain Feedforward Filter. *IEEE Transactions on Communications* 50(6), 947–955, 2002.

[14] D. Bertsimas and J. N. Tsitsiklis. *Introduction to Linear Optimization.* Athena Scientific, Belmont, Massachusetts, 1997.

[15] J. W. Chinneck and J. Patel. Faster MIP Solutions Through Better Variable Ordering, ISMP 2003, Copenhagen, August 2003.

[16] V. Chvátal. Edmonds Polytopes and a Hierarchy of Combinatorial Problem. *Discrete Mathematics* 4, 305–337, 1973.

[17] W. Cook, R. Kannan and A. Schrijver. Chvátal closures for mixed integer programs. *Mathematical Programming*, 47:155–174, 1990

[18] Dash Xpress-Optimizer 16.01.05: Getting Started and Reference Manual, Dash Optimization Ltd, `http://www.dashoptimization.com/`, 2004.

[19] E. Danna, E. Rothberg and C. Le Pape. Exploring relaxation induced neighborhoods to improve MIP solutions. *Mathematical Programming* 102, 71–90, 2005.

[20] G. B. Dantzig, D. R. Fulkerson and S. Johnson. Solution of a Large Scale Traveling Salesman Problem. *Journal of the Operations Research Society of America* 2, 393–410, 1954.

[21] DIMACS Second Challenge. `http://mat.gsia.cmu.edu/challenge.html`.

[22] S. Fang and S. Puthenpura. *Linear Optimization and Extensions: Theory and Algorithms.* Prentice Hall College Div, 1993.

[23] M. Fischetti, F. Glover and A. Lodi. The Feasibility Pump. *Mathematical Programming* 104, 91–104, 2005.

[24] M. Fischetti and A. Lodi. Local Branching. *Mathematical Programming* 98, 23–47, 2003.

[25] M. Fischetti and A. Lodi. Optimizing over the first Chvátal closure. To appear in *Mathematical Programming*, 2005.

[26] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, 1979.

[27] F. Glover and M. Laguna. General Purpose Heuristics For Integer Programming: Part I. *Journal of Heuristics* 2, 343–358, 1997.

[28] F. Glover and M. Laguna. General Purpose Heuristics For Integer Programming: Part II. *Journal of Heuristics* 3, 161–179, 1997.

[29] F. Glover and M. Laguna. *Tabu Search.* Kluwer Academic Publisher, Boston, Dordrecht, London, 1997.

[30] R. E. Gomory. Outline of an Algorithm for Integer Solutions to Linear Programs. *Bulletin of The American Mathematical Society* 64, 275–278, 1958.

[31] R. E. Gomory. An algorithm for the mixed integer problem. Technical Report RM-2597, The Rand Corporation, 1960.

[32] M. Grötschel, M. Jünger and G. Reinelt. A Cutting Plane Algorithm for the Linear Ordering Problem. *Operations Research*, 32, 1195–1220, 1984.

[33] M. Grötschel, M. Jünger and G. Reinelt. On the acyclic subgraph polytope. *Mathematical Programming*, 33, 28–42, 1985.

[34] M. Grötschel, M. Jünger and G. Reinelt. Facets of the Linear Ordering Polytope. *Mathematical Programming*, 33, 43–60, 1985.

[35] M. Grötschel, L. Lovász and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and Combinatorics.* Springer, 1988.

[36] Z. Gu, G. L. Nemhauser and M. W. P. Savelsbergh. Lifted cover inequalities for 01 integer programs: Computation. *INFORMS Journal on Computing*, 10:427–437, 1998.

[37] Z. Gu, G. L. Nemhauser and M. W. P. Savelsbergh. Lifted flow cover inequalities for mixed 01 integer programs. *Mathematical Programming*, 85:439–467, 1999.

[38] G. Gutin and A. Punnen (editors) *The Traveling Salesman Problem and its Variations*, Kluwer, 2002.

[39] P. L. Hammer, E. L. Johnson and U. N. Peled. Facets of regular 01 polytopes. *Mathematical Programming*, 8:179–206, 1975.

[40] F. S. Hillier. Effcient Heuristic Procedures For Integer Linear Programming With An Interior. *Operations Research* 17, 600–637, 1969.

[41] H. Holma and A. Toskala. *WCDMA for UMTS: Radio Access for Third Generation Mobile Communications*, Wiley, New York, 2000.

[42] T. Ibaraki, T. Ohashi and H. Mine. A Heuristic Algorithm For Mixed-Integer Programming Problems. *Mathematical Programming Study* 2, 115–136, 1974.

[43] ILOG Cplex 9: User's Manual and Reference Manual, ILOG, S.A., `http://www.ilog.com/`, 2004.

[44] ILOG Concert Technology 2: User's Manual and Reference Manual, ILOG, S.A., `http://www.ilog.com/`, 2004.

[45] R. Kannan and L. Lovász. Covering minima and lattice-point-free convex bodies. *Annals of Mathematics*, 128:577–602, 1988.

[46] M. Karamanov and G. Cornuéjols. Branching on General Disjunctions To appear in *Mathematical Programming*, 2005.

[47] A. H. Land and A. G. Doig. An Automatic Method for Solving Discrete Programming Problems. *Econometrica* 28, 497–520, 1960.

[48] H. W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8(4):538–548, 1983.

[49] A. K. Lenstra, H. W. Lenstra Jr. and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 1982.

[50] J. K. Lenstra, A. H. G. Rinnooy Kan and D. Shmoys (editors). *The Traveling Salesman Problem: A Guided Tour to Combinatorial Optimization,* Wiley, 251-305, 1985.

[51] J. Linderoth and M. Savelsbergh. A computational study of search strategies for mixed integer programming. Report LEC-97-12, Georgia Institute of Technology, 1997.

[52] A. Løkketangen. Heuristics for 0-1 Mixed-Integer Programming. In P.M. Pardalos and M.G.C. Resende (ed.s) *Handbook of Applied Optimization*, Oxford University Press, 474–477, 2002.

[53] A. Løkketangen and F. Glover. Solving Zero/One Mixed Integer Programming Problems Using Tabu Search. *European Journal of Operational Research* 106, 624–658, 1998.

[54] H. Marchand and L. A. Wolsey. Aggregation and mixed integer rounding. *Operations Research*, 49:363–371, 2001.

[55] A. Martin. Integer Programs with Block Structure. Habilitationsschrift, Technische Universität Berlin.

[56] H. D. Mittelmann. Benchmarks for Optimization Software: Testcases. `http://plato.asu.edu/topics/testcases.html`.

[57] MIPLIB website of Zuse Institute Berlin. `http://miplib.zib.de/miplib3/miplib.html`.

[58] M. Nediak and J. Eckstein. Pivot, Cut, and Dive: A Heuristic for 0-1 Mixed Integer Programming. Research Report RRR 53-2001, RUTCOR, Rutgers University, October 2001.

[59] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* Wiley-Interscience. John Wiley & Sons, New York, 1988.

[60] G. L. Nemhauser and L. A. Wolsey. A recursive procedure for generating all cuts for 01 mixed integer programs. *Mathematical Programming*, 46:379–390, 1990.

[61] J. Owen and S. Mehrotra. Experimental results on using general disjunctions in branch-andbound for general-integer linear program. *Computational Optimization and Applications*, 20:159–170, 2001.

[62] M. W. Padberg. On the facial structure of set packing polyhedra. *Mathematical Programming*, 5:199–215, 1973.

[63] M. W. Padberg. Covering, packing and knapsack problems. *Annals of Discrete Mathematics*, 4:265–287, 1979.

[64] M. W. Padberg and G. Rinaldi. Optimization of a 532-city symmetric traveling salesman problem by branch and cut. *Operations Research Letters*, 6, 1–7, 1987.

[65] M. W. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. SIAM Rev., 33, 60–100, 1991.

[66] M. W. Padberg, T. J. Van Roy and L. A. Wolsey. Valid linear inequalities for fixed charge problems. *Operations Research*, 33:842–861, 1985.

[67] P. Patel and J. Holzman. Analysis of a Simple Successive Interference Cancellation Scheme in a DS/CDMA System. IEEE J. Select. Areas Commun., 12, 727–807, 1994.

[68] L. Peeters. Cyclic Railway Timetable Optimization. ERIM PhD Series, Erasmus University Rotterdam, June, 2003.

[69] J. G. Proakis. *Digital Communications 4th edition*, McGraw Hill, New York, 2004.

[70] G. Reinelt. *The Linear Ordering Problem: Algorithms and Applications.* Research and Exposition in Mathematics 8, Heldermann Verlag, Berlin, 1985.

[71] M. W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. ORSA Journal on Computing, 6:445–454, 1994.

[72] A. Schrijver. On cutting planes. *Annals of Discrete Mathematics* 9, 291–296, 1980.

[73] S. S. Srivastava, S. Kumar, R. C. Garg and P. Sen. Generalized Travelling Salesman Problem through Sets of Nodes. *CORS Journal*, 7, 97–101, 1969.

[74] T. J. Van Roy and L. A. Wolsey. Solving mixed integer programming problems using automatic reformulation. *Operations Research*, 35:45–57, 1987.

[75] D. Warrier and U. Madhow. On the Capacity of Cellular CDMA with Successive Decoding and Controlled Power Disparities. in Proc. Vehic. Tech. Conf. (VTC), vol. 3, 1873–1877, 1998.

[76] L. A. Wolsey. *Integer Programming*. John Wiley & Sons, New York, 1998.