Department of Information Engineering
University of Padua

DRT: a General Method for Mixed Integer Problems

Author: Massimo Scantamburlo

Supervisor: prof. Matteo Fischetti

University of Padua

Academic Year: 2001/2002

Abstract

In questo lavoro è stato affrontato il problema della risoluzione di problemi di programmazione lineare intera mista (MIP) per istanze particolarmente impegnative. È noto, infatti, che molti problemi di questo tipo sono NP-hard e quindi non si riesce a ottenere l'ottimo in tempi accettabili se la taglia dell'istanza è elevata. In questi casi si preferisce, quindi, accontentarsi di una soluzione vicina all'ottimo pur di ottenere un risultato in un tempo ragionevole. Gli algoritmi sviluppati in questo ambito vengono detti *euristici* e si contrappongono a metodi esatti già implementati in softwares commerciali. Per quanto riguarda metodi esatti, abbiamo descritto brevemente sia Cplex 7.0 di ILOG S.A., sia il GLPK (GNU Linear Programming Toolkit) distribuito gratuitamente e open source. Entrambi mettono a disposizione delle librerie di programmazione che possono essere usate nei metodi euristici per indagare regioni limitate dello spazio delle soluzioni. In questa sede sono stati trattati poi tre tipi di algoritmi euristici già sviluppati e testati: la *Tabu Search* proposta da F. Glover, il *Variable Neighborhood Search* di N. Mladenovíc e P. Hansen e il *Local Branching* di M. Fischetti e A. Lodi. Per ciascuno di essi viene riportata una breve introduzione teorica e un possibile pseudocodice. Dopo questa parte introduttiva e propedeutica, si passa all'esposizione del metodo DRT (*Diversification, Refining, Tight-Refining*). Esso si basa essenzialmente sulla considerazione che, in alcuni casi, la struttura logica delle variabili binarie di un problema può consentire di ridurre notevolmente gli sforzi computazionali. In particolare, vengono individuati due livelli di variabili binarie: livello 1 e 2. Il primo diremo che controlla il secondo, poiché una determinata configurazione delle variabili di primo livello impone alle variabili di secondo livello un range di configurazioni possibili. Si pensi, ad esempio, al caso dei problemi di *location*: se un concentratore viene disattivato, allora tutti i links che lo connettono ai terminali devono essere annullati. Ci si rende conto, dunque, di quanto il problema si semplifichi in questo modo. Sono stati discussi due algoritmi di costruzione automatica dell'insieme di variabili di primo livello. Il primo è basato proprio sulla struttura dei vincoli appena discussa. Il secondo sulla presenza di coefficienti big M : imponendo un determinato valore per ciascuna di esse, viene eliminato l'indesiderato effetto di generare molti livelli dell'albero di branching causato da un coefficiente molto maggiore degli altri. Il metodo di risoluzione qui proposto consiste nell'alternanza di tre fasi. Nella prima (*diversification*), si restringe l'indagine dello spazio delle soluzioni ad un intorno di primo livello, relativo cioè alle variabili di livello 1. Una volta fissata la configurazione delle variabili di primo livello, si passa a una fase di intensificazione, in cui si esplorano intorni di secondo livello (*refining*). Se il problema così ottenuto dovesse essere ancora troppo difficile da essere risolto con il software a disposizione (nel nostro caso ILOG-Cplex o GLPK), si applica un algoritmo iterativo di ricerca locale (fase di *tight-refining*). Al termine di ciascun ciclo DRT si impone un vincolo *tabu* alle variabili di primo livello, in modo da non visitare nuovamente la stessa

configurazione. Si è poi proposta anche una variante del DRT, detta DRT2, in cui viene aggiunta una fase di *big dievrsification*: nel caso in cui non vengano ottenute significative migliorie dopo un certo numero di diversificazioni, si applica una drastico cambiamento di intorno di primo livello, al fine di cercare regioni più promettenti. Questo procedimento in parte sfrutta il concetto di conoscenza della storia recente delle soluzioni considerta, ereditato dalla tabu search. Viene quindi riportato un dettagliato pseudo-codice del DRT2.

Come test bed è stata utilizzato un insieme di istanze legate alla progettazione di reti UMTS, affrontata da due punti di vista diversi. Altre istanze sono state indicate come possibili candidati, nonché utilizzate per testare gli algoritmi di costruzione dell'insieme di variabili di primo livello. I risultati sono stati sempre molto buoni nel caso del metodo delle variabili forzate. Nel caso del metodo basato sui big M, invece, si sono evidenziati comportamenti influenzati dal settaggio del criterio per dire se un coefficiente è o no big M.

Per quanto riguarda il metodo vero e proprio, si sono fatti dei confronti sia con ILOG-Cplex, uno dei punti di riferimento nell'ambito dei software commerciali, sia con gli altri metodi euristici citati prima. I risultati sono sempre stati più che soddisfacenti, evidenziando come il DRT ottenga soluzioni uguali o migliori rispetto a tutti i concorrenti in quasi tutti i casi. Anche la variante del DRT, il DRT2, ha dati risultati soddisfacenti, seppure non quanto desiderato, poiché un cambiamento totale di regione di indagine obbliga a fare alcuni cicli di DRT prima di arrivare ad una soluzione di qualità paragonabile a quella che si aveva prima della big diversification. A margine sono stati effettuati anche alcuni test per confrontare Cplex con il GLPK al fine di vedere quale dei due solver potesse essere utilizzato come strumento per la risoluzione dei sotto-problemi generati. Se da un punto di vista della velocità e della qualità della soluzione le differenze non sono eclatanti, Cplex si è dimostrato più affidabile sia per quanto riguarda la capacità di leggere i modelli in ingresso, sia per la maggiore stabilità numerica, nonché per la disponibilità di molti più parametri di configurazione a livello di librerie di programmazione.

Contents

Chapter 1

Introduction

In this work we face the task of solving large mixed integer problems (MIPs). Their importance is growing in these times since new kinds of models have been developed to solve optimization tasks such as the planning of a telecommunications network. In most cases we deal with an enormous number of entities, thus commercial solvers that try to find the optimum value would take too long time (these problems are often NP-hard). So, in real cases we may need to find a good (not necessarily the best) solution in an acceptable time. In this thesis we present a general MIP metaheuristic method based on the use of a solver like, for example, ILOG-Cplex or the GNU linear programming kit (GLPK) to investigate subspaces of the starting problem. The aim of this procedure is that we will divide the main problem in many smaller subproblems that will hopefully be easier to solve. Dynamically adding new cuts will help us to investigate subspaces of the solution space using the general purpose solver. A fundamental point is how to generate these neighborhoods. The basic idea is to split the binary variables into two subsets: we build the subproblems changing the configuration of one of these two subsets. Then we apply to each configuration a research divided in two phases: the first based on increasing size neighborhoods and the second on a classical iterative procedure. We will call the whole algorithm DRT standing for the three steps: diversification, refining, tight-refining.

Before the discussion of the main algorithm given in chapter 4, we will introduce two MIP solvers in chapter 2 and then, in chapter 3, three heuristic algorithms to highlight some ideas that will be deeply exploited . For each solution method a theoretical description and a pseudo-code are given. At the end we will present some computational results on a large set of instances of different kinds (a complete description of each test model is provided in appendix). The aim is to show that our method can works with good performance for a significant class of problems.

Chapter 2

MIP and LP solvers

2.1 Introduction to ILOG-Cplex 7.0

2.1.1 Introduction

The commercial software Cplex by ILOG S.A. is a state-of-the-art tool for optimization problems: it can solve linear Mixed Integer Programming, Quadratic programming and Network flow problems. We can choose between three interfaces:

- *Cplex Interactive Optimizer*: an interactive shell that can read the problem from a file or from standard input and solve the model writing the solution on a file or to standard output

- *Concert technology*: a set of libraries to include Cplex optimizers in C++ programs

- *Cplex Callable library*: a library to include Cplex optimizers in C, Visual Basic, Java, and Fortran applications.

We will focus our attention on the Cplex capability of solving linear programming problems with or without integer variables constraints in the form:

$$
\begin{array}{ll}
\text{max/min} & c_1x_1 + c_2x_2 + \ldots + c_nx_n \\
\\
\text{subject to} & a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \sim b_1 \\
& a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \sim b_2 \\
& \ldots \\
& a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \sim b_m \\
\\
\text{bounds} & l_1 \leq x_1 \leq u_1 \\
& l_2 \leq x_2 \leq u_2 \\
& \ldots \\
& l_n \leq x_n \leq u_n
\end{array}
$$

where $\sim$ can be $\leq$, $\geq$ or $=$ while the lower bounds l_i and upper bounds u_i are real numbers, including positive and negative infinity. For each variable (also called column) we can specify the type (integer, binary, general). The functional mode we will use is the Concert Technology since implement our methods using C++.

We pass to the program the c_i costs that are the coefficients of the objective function; the a_{ij} constraints coefficients and b_i constants. We can build the model first reading it from a file in any supported format and then, at run time, by adding constraints. The lp and mps formats are the most common ones: the first let us entering the model in an algebraic, row-oriented form; in other words we specify the problem in terms of its constraints. The latter uses columns-oriented models. For a detailed lp/mps file types description see [**?**].

2.1.2 Programming with Cplex 7.0

Briefly, to solve a problem with Cplex using Concert Technology we have to follows these steps:

- include the `ilcplex/ilocplex.h` header (for detailed instructions concerning Cplex use with programming environments such as Microsoft Visual C++ see file provided with Cplex distribution);
- eventually call the macro `ILOSTLBEGIN` needed for portability;
- declare the environment with `IloEnv env`;
- declare the model, the class used to represent optimization problems, with `IloModel mod(env)`;
- build the model loading it from a file or interactively using `mod.add(objective_function)`, and `mod.add(constraint)`;
- declare a Cplex object and assign to it the model to be solved with `IloCplex cplex(mod)`;

- eventually set the solver parameters with instructions `cplex.setParam(Parameter,value);`
- solve the model with `cplex.solve()`. We can choose between eight solving algorithms that belong to the enumeration type `IloCplex::Algorithm`. We also can set the type of solver with `cplex.setRootAlgorithm(alg);`
- query solution
- end the environment with `env.close()` to destroy implementation objects (it avoids memory leaks).

2.1.3 Solver parameters

As we said, Cplex let us set some parameters: in this way we can have different behaviors of the solver suiting our needs. For a detailed list of these parameters see [**?**]. Here we see just those ones which will be used for a complete understanding of the algorithms implementation.

- `TiLim`: maximum execution time; once it's elapsed the solver stops even if optimality is not reached
- `NodeLim`: maximum number of nodes to be explored before the solver stops even without optimal solution has been found
- `TreLim`: maximum memory amount (in Mega Bytes) that branch and cut algorithm can use to build the tree
- `NodeFileInd`: tells the solver what to do when maximum memory amount (TreLim) has been reached: if set to 0 the solver stops, else writes some nodes to files in the hard disk
- `MIPEmphasis`: if it is set to 0 the solver gives priority to the optimality while value 1 forces the solver to generate as many feasible good intermediate solutions as possible
- `CutUp`: upper cutoff; cuts off any nodes having an objective value above this parameter (assuming a minimization problem). When a mixed integer optimization problem is continued, the smaller of these values and the updated cutoff found during optimization are used during the next mixed integer optimization. A too restrictive value may result in no integer solutions being found.
- `IntSolLim`: maximum number of MIP solutions to be found before stopping

- `EpAGap`: absolute mipgap tolerance; sets an absolute tolerance on the gap between the value of best integer solution and the lower bound of the best node remaining. When this difference falls below this value, the mixed integer optimization is stopped

- `EpGap`: relative mipgap tolerance; sets a relative tolerance on the gap between the value of best integer solution and the lower bound of the best node remaining. When this difference falls below this value the mixed integer optimization is stopped

2.2 Introduction to GLPK 3.2.2

Now we are going to give a brief overview of GLPK (*GNU Linear Programming Kit*), a quite new software for solving MIPs. The package was developed in the 2002 and distributed under the GPL (GNU Public Licence) and so comes for free. The other main issue is that we have access to the whole source code and so we can change it in order to suite our needs.

We have three main ways to use this tool:

- `GLPK/L`: a modeling language which is intended for writing mathematical programming models. Model description written in GLPK/L language consists of a sequence of statements constructed by the user from the language elements. In a process called translation a program called the language processor analyzes the model description statements and translates them into internal data structures, which may be then used either for generating mathematical programming problem data or directly by a program called the solver for obtaining numerical solution of the problem. Note that this language is theoretically not restricted to the linear programming (LP) since it can describe also nonlinear models. However the solver doesn't implement a solution method for these ones and so we will consider only LP.

- `GLPSOL`: a sort of the ILOG Cplex interactive optimizer. We can use it giving at the prompt line the path of the model file in mps or lp format (and also in GLPK/L).

- `GLPK`: the package is basically a set of routines written in ANSI C programming language and organized in the form of a callable library. It is intended for solving linear programming, mixed integer programming and other related problems.

We focus our attention on the third way.

2.2.1 The problem model

GLPK assumes the following formulation of *linear programming (LP)* problem:

minimize (or maximize)

$$Z = c_1x_1 + c_2x_2 + \ldots + c_{m+n}x_{m+n} + c_0$$

subject to linear constraints

$$\begin{aligned} x_1 &= a_{11}x_{m+1} + a_{12}x_{m+2} + \ldots + a_{1n}x_{m+n} \\ x_2 &= a_{21}x_{m+1} + a_{22}x_{m+2} + \ldots + a_{2n}x_{m+n} \\ &\ldots\ldots\ldots\ldots\ldots\ldots \\ x_m &= a_{m1}x_{m+1} + a_{m2}x_{m+2} + \ldots + a_{mn}x_{m+n} \end{aligned}$$

and bounds of variables

$$\begin{aligned} l_1 &\le x_1 \le u_1 \\ l_2 &\le x_2 \le u_2 \\ &\ldots\ldots\ldots \\ l_{m+n} &\le x_{m+n} \le u_{m+n} \end{aligned}$$

where: $x_1, x_2, \ldots, x_m$ — auxiliary variables; $x_{m+1}, x_{m+2}, \ldots, x_{m+n}$ — structural variables; Z — objective function; $c_1, c_2, \ldots, c_{m+n}$ — coefficients of the objective function; c_0 — constant term of the objective function; $a_{11}, a_{12}, \ldots, a_{mn}$ — constraint coefficients; $l_1, l_2, \ldots, l_{m+n}$ — lower bounds of variables; $u_1, u_2, \ldots, u_{m+n}$ — upper bounds of variables.

Auxiliary variables are also called *rows*, because they correspond to rows of the constraint matrix (i.e. a matrix built of the constraint coefficients). Analogously, structural variables are also called *columns*, because they correspond to columns of the constraint matrix.

Bounds of variables can be finite as well as infinite. Besides, lower and upper bounds can be equal to each other. Thus, the following types of variables are possible:

Bounds of variable	Type of variable
$-\infty < x_k < +\infty$	Free (unbounded) variable
$l_k \le x_k < +\infty$	Variable with lower bound
$-\infty < x_k \le u_k$	Variable with upper bound
$l_k \le x_k \le u_k$	Double-bounded variable
$l_k = x_k = u_k$	Fixed variable

Note that the types of variables shown above are applicable to structural as well as to auxiliary variables.

To solve the LP problem is to find such values of all structural and auxiliary variables, which:

a) satisfy to all the linear constraints, and
b) are within their bounds, and
c) provide an optimum value of the objective function.

For solving LP problems GLPK uses a well known numerical procedure called *the simplex method*. The simplex method performs iterations, where on each iteration it transforms the original system of equaility constraints resolving them through different sets of variables to an equivalent system called *the simplex table* (or sometimes *the simplex tableau*), which has the following form:

$$\begin{aligned} Z = \ & d_1(x_N)_1 + \ d_2(x_N)_2 + \ldots + \ d_n(x_N)_n \\ (x_B)_1 = \ & \alpha_{11}(x_N)_1 + \alpha_{12}(x_N)_2 + \ldots + \alpha_{1n}(x_N)_n \\ (x_B)_2 = \ & \alpha_{21}(x_N)_1 + \alpha_{22}(x_N)_2 + \ldots + \alpha_{2n}(x_N)_n \\ & \ldots\ldots\ldots\ldots\ldots \\ (x_B)_m = \ & \alpha_{m1}(x_N)_1 + \alpha_{m2}(x_N)_2 + \ldots + \alpha_{mn}(x_N)_n \end{aligned}$$

where: $(x_B)_1, (x_B)_2, \ldots, (x_B)_m$ — basic variables; $(x_N)_1, (x_N)_2, \ldots, (x_N)_n$ — non-basic variables; $d_1, d_2, \ldots, d_n$ — reduced costs; $\alpha_{11}, \alpha_{12}, \ldots, \alpha_{mn}$ — coefficients of the simplex table. (May note that the original LP problem also has the form of a simplex table, where all equalities are resolved through auxiliary variables.)

From the linear programming theory it is well known that if an optimal solution of the LP problem exists, it can always be written in the second form, where non-basic variables are fixed on their bounds, and values of the objective function and basic variables are determined by the corresponding equalities of the simplex table.

A set of values of all basic and non-basic variables determined by the simplex table is called *basic solution*. If all basic variables are within their bounds, the basic solution is called *(primal) feasible*, otherwise it is called *(primal) infeasible*. A feasible basic solution, which provides a smallest (in case of minimization) or a largest (in case of maximization) value of the objective function is called *optimal*. Therefore, for solving LP problem the simplex method tries to find its optimal basic solution.

Primal feasibility of some basic solution may be stated by simple checking if all basic variables are within their bounds. Basic solution is optimal if additionally the following optimality conditions are satisfied for all non-basic variables:

Status of $(x_N)_j$	Minimization	Maximization
$(x_N)_j$ is free	$d_j = 0$	$d_j = 0$
$(x_N)_j$ is on its lower bound	$d_j \geq 0$	$d_j \leq 0$
$(x_N)_j$ is on its upper bound	$d_j \leq 0$	$d_j \geq 0$

In other words, basic solution is optimal if there is no non-basic variable, which changing in the feasible direction (i.e. increasing if it is free or on its lower bound, or decreasing if it is free or on its upper bound) can improve (i.e. decrease in case of minimization or increase in case of maximization) the objective function.

If all non-basic variables satisfy to the optimality conditions shown above (independently on whether basic variables are within their bounds or not), the basic solution is called *dual feasible*, otherwise it is called *dual infeasible*.

It may happen that some LP problem has no primal feasible solution due to incorrect formulation — this means that its constraints conflict with each other. It also may happen that some LP problem has unbounded solution again due to incorrect formulation — this means that some non-basic variable can improve the objective function, i.e. the optimality conditions are violated, and at the same time this variable can infinitely change in the feasible direction meeting no resistance from basic variables. (May note that in the latter case the LP problem has no dual feasible solution.)

2.2.2 The software interface

As we said, this package is mainly devoted to be used as a set of routines in a C program to solve linear programming or mixed integer programming problems. The three aspects we are going to tell something about regard:

- the way variables and constraints are treated
- how to solve a model
- parameters

Note that the API routines provide also some data checking and if something gone wrong display a message and stop the execution. Thus, in order to prevent crashes we should check all data which are suspected to be incorrect before calling GLPK routines.

To all routines we have to pass a pointer to the problem object structure called `LPX`: we can create it from scratch with `LPX *lpx_create_prob()` or reading it from a file with the routines `LPX *lpx_read_mps(char *file_name)` or `LPX *lpx_read_lp(char *file_name)` or `LPX *lpx_read_lpt(char *file_name)` for each supported file type. It is up to the programmer check what type of file has been submitted and so what routine must be called.

Variables and constraints

As we have told in 2.2.1, variables and constraints are basically seen in the same way: in GLPK terms we can call the first columns and the latter rows or auxiliary variables. They all have an identifier made up of an index and a mnemonic name and a couple of bounds. Note that in constraints or variables operation we can't refer to a row or a column with its name but only using its index. The most important operations we can do are:

- add one or more columns/rows

- delete one or more colums/rows
- query colums/rows information such as bounds, coefficients, type (applied only to columns)

It is worth to emphasize the mechanism to add rows and delete them since we will use these ones a lot[1].

To add a new constraint we have to use these routines:

1. `void lpx_add_rows(LPX *lp,int num_rows)`: add at the end of the rows list new `num_rows` empty rows to the problem object pointed by lp;
2. `void lpx_set_row_bnds(LPX *lp,int row_index,int type, double LB,double UB)`: set the bounds of the `row_index`-th row;
3. `void set_mat_row(LPX *lp, int row_index, int len, int ndx[], double val[])`: sets (replaces) the `row_index`-th row of the constraint matrix for the problem object, which the parameter `lp` points to. Column indices and numerical values of new non-zero coefficients of the `row_index`-th row should be placed in the locations `ndx[1]`, ..., `ndx[len]` and `val[1]`, ..., `val[len]`, respectively, where $0 \leq$ `len` $\leq n$ is the new length of the `row_index`-th row, n is number of columns.
4. `void lpx_set_row_name(LPX *lp, int index, char *name)` : set a mnemonic name for the `index`-th row
5. `void lpx_mark_row(LPX *lp, int i, int mark)` : assigns an integer `mark` to the `i`-th row. The reason to mark a row is that is the only way we can delete it after.

To delete a constraint we use the routine `void lpx_del_items(LPX *lp)` which will erase *all* marked rows. Again, since we can't refer to indexes or names to delete a constraint, the programmer has to use the markers in a clever way.

Solving a problem

To solve a linear programming problem we use the simplex method calling the routine `int lpx_simplex(LPX *lp)` which is an interface to the LP problem solver based on the two-phase revised simplex method. This routine obtains problem data from the problem object, which the parameter `lp` points to, calls the solver to solve the LP problem, and stores the found solution and other relevant information back in the problem object. Generally, the simplex solver does

[1]The GLPK uses to start arrays index from 1 instead of 0. We have to keep it in mind to avoid abnormal behaviors.

the following:
1 - "warming up" the initial basis;
2 - searching for (primal) feasible basic solution (phase I);
3 - searching for optimal basic solution (phase II)
4 - storing the final basis and found basic solution back in the problem object.
Since large scale problems may take a long time, the solver reports some information about the current basic solution, which is sent to the standard output. This information has the following format:

```
*nnn:   objval = xxx   infeas = yyy (ddd)
```

where: '`nnn`' is the iteration number, '`xxx`' is the current value of the objective function (which is unscaled and has correct sign), '`yyy`' is the current sum of primal infeasibilities (which is scaled and therefore may be used for visual estimating only), '`ddd`' is the current number of fixed basic variables. If the asterisk '`*`' precedes to '`nnn`', the solver is searching for an optimal solution (phase II), otherwise the solver is searching for a primal feasible solution (phase I). The routine `lpx_simplex` returns one of the following exit codes:

`LPX_E_OK`	the LP problem has been successfully solved. (Note that, for example, if the problem has no feasible solution, this exit code is reported.)
`LPX_E_FAULT`	unable to start the search because either the problem has no rows/columns, or the initial basis is invalid, or the initial basis matrix is singular or ill-conditioned.
`LPX_E_OBJLL`	the search was prematurely terminated because the objective function being maximized has reached its lower limit and continues decreasing (the dual simplex only).
`LPX_E_OBJUL`	the search was prematurely terminated because the objective function being minimized has reached its upper limit and continues increasing (the dual simplex only).
`LPX_E_ITLIM`	the search was prematurely terminated because the simplex iterations limit has been exceeded.
`LPX_E_TMLIM`	the search was prematurely terminated because the time limit has been exceeded.
`LPX_E_SING`	the search was prematurely terminated due to the solver failure (the current basis matrix got singular or ill-conditioned).

Sometimes numerical instability troubles can arise; we can try to eliminate them using the routine `lpx_scale_prob` which performs scaling problem data for the specified problem object. The purpose of scaling is to replace the original constraint matrix A by the scaled matrix $A' = RAS$, where R and S are diagonal scaling matrices, in the hope that A' has better numerical properties than A. On API level the scaling effect is almost invisible, since all data entered into the problem object (say, constraint coefficients or bounds of variables) are automati-

cally scaled by API routines using the scaling matrices R and S, and vice versa, all data obtained from the problem object (say, values of variables or reduced costs) are automatically unscaled. However, round-off errors may involve small distortions (of order `DBL_EPSILON`) of the original problem data.

When we face MIPs we have to use the routine `int lpx_integer(LPX *lp)`, an interface to the MIP problem solver based on the *branch-and-bound* method. As the simplex one, this routine obtains problem data from the problem object, which the parameter `lp` points to, calls the solver to solve the MIP problem, and stores the found solution and other relevant information back in the problem object. On entry to this routine the problem object should contain an optimal basic solution for LP relaxation, which can be obtained by means of the simplex-based solver. So, remember that to solve a mixed integer programming problem we have to:

1. find a solution for LP relaxation with `lpx_simplex`

2. solve the problem with `lpx_integer`

Again, the solver reports some information about the best known solution, which is sent to the standard output. This information has the following format:

```
+nnn: mip = xxx; lp = yyy (mmm; nnn)
```

where `nnn` is the simplex iteration number, `xxx` is a value of the objective function for the best known integer feasible solution (if no integer feasible solution has been found yet, `xxx` is the text `not found yet`), `yyy` is an optimal value of the objective function for LP relaxation (this value is not changed during all the search), `mmm` is number of subproblems in the active list, `nnn` is number of subproblems which have been solved (considered).

Note that the branch-and-bound solver implemented in GLPK uses easiest heuristics for branching and backtracking, and therefore it is not perfect. Even if the GLPK author states that most probably this solver can be used for solving MIP problems with one or two hundreds of integer variables, we have tested it on some very large problems with good results (see the results section). The routine `lpx_integer` returns one of the following exit codes:

`LPX_E_OK` the MIP problem has been successfully solved. (Note that, for example, if the problem has no integer feasible solution, this exit code is reported.)

`LPX_E_FAULT` unable to start the search because either:
the problem is not of MIP class, or
the problem object doesn't contain optimal solution for LP relaxation, or
some integer variable has non-integer lower or upper bound, or
some row has non-zero objective coefficient.

`LPX_E_ITLIM` the search was prematurely terminated because the simplex iterations limit has been exceeded.

`LPX_E_TMLIM` the search was prematurely terminated because the time limit has been exceeded.

`LPX_E_SING` the search was prematurely terminated due to the solver failure (the current basis matrix got singular or ill-conditioned).

Parameters

The GLPK provides a set of control parameters, real or integer. To access them we can use `int lpx_get_int_parm(LPX *lp, int parm)`; to change their value we call `void lpx_set_int_parm(LPX *lp, int parm, int val)` (or the real correspondent). The most important parameters are:

`LPX_K_MSGLEV` type: integer, default: 3
Level of messages output by solver routines:
0 — no output
1 — error messages only
2 — normal output
3 — full output (includes informational messages)

`LPX_K_SCALE` type: integer, default: 3
Scaling option:
0 — no scaling
1 — equilibration scaling
2 — geometric mean scaling, then equilibration scaling

`LPX_K_DUAL` type: integer, default: 0
Dual simplex option:
0 — do not use the dual simplex
1 — if initial basic solution is dual feasible, use the dual simplex

`LPX_K_PRICE` type: integer, default: 1
Pricing option (for both primal and dual simplex):
0 — textbook pricing
1 — steepest edge pricing

`LPX_K_ROUND` type: integer, default: 0
Solution rounding option:
0 — report all primal and dual values "as is"
1 — replace tiny primal and dual values by exact zero

`LPX_K_OBJLL` type: real, default: `-DBL_MAX`
Lower limit of the objective function. If on the phase II the objective function reaches this limit and continues decreasing, the solver stops the search. (Used in the dual simplex only.)

`LPX_K_OBJUL` type: real, default: `+DBL_MAX`
Upper limit of the objective function. If on the phase II the objective function reaches this limit and continues increasing, the solver stops the search. (Used in the dual simplex only.)

`LPX_K_ITLIM` type: integer, default: -1
Simplex iterations limit. If this value is positive, it is decreased by one each time when one simplex iteration has been performed, and reaching zero value signals the solver to stop the search. Negative value means no iterations limit.

`LPX_K_ITCNT` type: integer, initial: 0
Simplex iterations count. This count is increased by one each time when one simplex iteration has been performed.

`LPX_K_TMLIM` type: real, default: -1.0
Searching time limit, in seconds. If this value is positive, it is decreased each time when one simplex iteration has been performed by the amount of time spent for the iteration, and reaching zero value signals the solver to stop the search. Negative value means no time limit.

`LPX_K_OUTFRQ` type: integer, default: 200
Output frequency, in iterations. This parameter specifies how frequently the solver sends information about the solution to the standard output.

`LPX_K_OUTDLY` type: real, default: 0.0
Output delay, in seconds. This parameter specifies how long the solver should delay sending information about the solution to the standard output. Non-positive value means no delay.

`LPX_K_BRANCH` type: integer, default: 2
Branching heuristic option (for MIP only):
0 — branch on the first variable
1 — branch on the last variable
2 — branch using a heuristic by Driebeck and Tomlin

`LPX_K_BTRACK` type: integer, default: 2
Backtracking heuristic option (for MIP only):
0 — depth first search
1 — breadth first search
2 — backtrack using the best projection heuristic

`LPX_K_TOLINT` type: real, default: 10^{-6}
Absolute tolerance used to check if the current basic solution is integer feasible. (Do not change this parameter without detailed understanding its purpose.)

`LPX_K_TOLOBJ` type: real, default: 10^{-7}
Relative tolerance used to check if the value of the objective function is not better than in the best known integer feasible solution. (Do not change this parameter without detailed understanding its purpose.)

The number of parameters is quite smaller than the one provided by Cplex. However they may change in future, especially with respect to the solvers since their implementations could be modified. Other parameters can be added in a quite easy way such as the MIP absolute or relative gaps (see Cplex EpAGap and EpGap)

Chapter 3

MIP general methods

3.1 Introduction

Mixed-integer linear programming plays a central role in modelling difficult-to-solve (NP-hard) combinatorial problems. So it's not surprising that the availability of effective exact or heuristic solution methods for mixed-integer problems (MIPs) is of paramount importance for practical applications. The main problem is that even if we know that the optimal solution can be reached, in most of common real cases the problem size is too large to be solved in a reasonable time. Several "ad hoc" heuristic algorithms have been developed for specific classes of problems; that often can give a solution with a low gap with respect to the optimal one in a fixed time. We have also commercial or free MIP solvers that let us to configure some parameters in order to improve the performance using a clever tuning. However in some very large size problems they can take too much time.

In this thesis we develop a general method that can exploit the use of a general-purpose MIP solver as a black-box "tactical" tool (in our case the state-of-the-art commercial software ILOG-Cplex 7.0). With it we can explore effectively suitable solution subspaces defined and controlled at a "strategic" level by a simple external framework. This allows one to work within a perfectly general MIP environment, and to take advantage of the impressive research and implementation effort that nowadays is devoted to the design of MIP solvers. The new solution strategy is exact in nature, though it is designed to improve the heuristic behavior of the MIP solver at hand. A simple but very powerful variant is to fix some variables obtaining simple subproblems that we can solve in a short time.

3.2 The problem formulation

We consider a generic MIP with 0-1 variables of the form:

$$
\begin{aligned}
(P) \qquad & \min c^T x && && (3.1)\\
& Ax \geq b && && (3.2)\\
& x_j \in \{0,1\} && \forall j \in \mathcal{B} \neq \emptyset && (3.3)\\
& x_j \geq 0, \text{ integer} && \forall j \in \mathcal{G} && (3.4)\\
& x_j \geq 0 && \forall j \in \mathcal{C} && (3.5)
\end{aligned}
$$

Here, the variable index set $\mathcal{N} := \{1, \ldots, n\}$ is partitioned into $(\mathcal{B}, \mathcal{G}, \mathcal{C})$, where $\mathcal{B} \neq \emptyset$ is the index set of the 0-1 variables, while the possibly empty sets $\mathcal{G}$ and $\mathcal{C}$ index the general integer and the continuous variables, respectively.

Given a *reference solution* $\bar{x}$ of (P), let $\overline{S} := \{j \in \mathcal{B} : \bar{x}_j = 1\}$ denote the binary support of $\bar{x}$. For a given positive integer parameter k, we define the *k-OPT neighborhood* $\mathcal{N}(\bar{x}, k)$ of $\bar{x}$ as the set of the feasible solutions of (P) satisfying the additional *local branching constraint*:

$$\Delta(x, \bar{x}) := \sum_{j \in \overline{S}} (1 - x_j) + \sum_{j \in \mathcal{B} \setminus \overline{S}} x_j \leq k \tag{3.6}$$

where $\Delta(x, \bar{x})$ is the distance between x and $\bar{x}$ and the two terms in the left-hand side count the number of binary variables flipping their value (with respect to $\bar{x}$) either from 1 to 0 or from 0 to 1, respectively.

In the relevant case in which the cardinality of the binary support of any feasible solution of (P) is a constant, this constraint can more conveniently be written in its equivalent "asymmetric" form

$$\sum_{j \in \overline{S}} (1 - x_j) \leq k' \ (= k/2) \tag{3.7}$$

3.3 Tabu search

3.3.1 The method

The tabu search method was developed essentially by Fred Glover (see [**?**]) and is deeply discussed in [**?**]. It is a general algorithm that can solve problems on the form

$$\begin{aligned} &minimize\ f(x) \\ &x \in X \end{aligned}$$

with a very low programming effort.

We start from an initial solution $x_0 \in X$ and take its neighborhood $N(x_0) \subseteq X$ built with some pre-defined metrics. Solving the new problem with the added constraint that $x \in N(x_0)$ leads to the new solution $x_1 = argmin(f(x) \ : x \in N(x_0))$. We set x_1 as the current best solution if is better than the previous one. Now take $x \in N(x_1)$ as additional constraint and solve the problem finding x_2 and so on. This iterative procedure will end when we find a solution x_n that is a local optimum, so no other $x \in N(x_n)$ is better.

As we said, the current solution is just a local optimum, hence we could try to explore other solutions subspaces. One could take another solution x_{n+1} worse than x_n. Obviously, we can take the solution that is nearest to x_n and so we have $x_{n+1} = argmin(f(x))$ with $x \in N(x_n) \setminus \{x_n\}$. From this new point we start again our local search with the hope that a new better solution will be found.

Changing the neighborhood can lead to visit twice a solution, with the risk of loops. We can avoid this problem keeping track of the last visited solutions and checking if we are going to consider one of them again. In practice we can put a certain number of solutions in a *FIFO* list of forbidden (*tabu*) solutions: the oldest solution in the list will be the first deleted, since we hope that in the meanwhile we went far enough from that point to avoid finding that solution again. Since for most problems the size of a solution (that is the number of variables) can be very large, managing a list of them could be time and space expensive. What we do is to keep track of *moves* instead of complete solutions: they are defined as the set of basic operations to be done to pass from a solution to another. So, for each solution $i \in S$, we define $M(i)$ as the set of moves m that can be applied to i in order to reach a solution $j \in S$ (the notation is $j = i \oplus m$; usually m is reversible, that is $(i \oplus m) \oplus m^{-1} = i$). The improvement of this method is basically done by the implementation of the way we keep track of previous solutions and by the choice of the moves. This last is obviously strictly related to the problem at hand.

A good idea should be to apply a *diversification* phase: when we see that no better solution (with respect to the current one) can be found after a fixed

number of steps, we go from the current neighborhood to a completely different subspace and continue the search starting from there. So we have two phases: the *intensification* phase when we apply the basic algorithm, and the *diversification* phase when we jump to another solution.

3.3.2 The pseudo-code

Briefly, an idea of how the Tabu Search method can be implemented. We report the pseudo-code fragment related to a generic iteration of the algorithm.

Algorithm 1 : Tabu Search pseudo-code

```
1: function TS()
2: choose a starting solution i from S set of solutions
3: i*=i
4: k:=1
5: while not stop condition do
6:    k:=k+1
7:    add the move to the tabu list T(k)
8:    build N(i,k):=N(i)\T(k)
9:    build the reduced neighborhood N̂(i,k) ⊆ N(i,k)
10:   find the best j ∈ N̂(i,k)
11:   i:=j
12:   if f(i) is better than f(i*) then
13:      i*:=i
14:   end if
15: end while
```

As already said, the main problems are the choice of new moves and of the data structure T to keep track of them while the starting solution can be obtained for example by a simple greedy algorithm. It is worth emphasizing that, no matter the way to manage and implement T, a cycle of length smaller than $|T|$ can take place since we loose some information by keeping memory of the moves instead of the whole solution. The stop condition can be defined by the user considering a time limit or a maximum number of iterations, or the fact that the next neighborhood is empty. Now we see the diversification method presented in [**?**]. We can assign an high priority to solutions that are "similar" to the current one by introducing in the objective function a new term (called f_{int}) to penalize the "distant" solutions (that is, the ones we can reach with a huge number of moves) and which is different from zero for all the iterations we want to perform the *intensification*. On a very similar way we obtain a *diversification* by introducing a correcting component in the objective function called f_{div}. The new global function to be minimized will be:

$$\bar{f} := f + f_{int} + f_{div}$$

3.4 Variable Neighborhood Search (VNS)

3.4.1 The method

The Variable Neighborhood Search (VNS) is a quite recent metaheuristic technique introduced by Mladenovíc and Hansen in [**?**]. In its first version this algorithm implementation is dependent on the problem type we have to solve. The VNS we are going to expose is the one generalized by Polo in [**?**]. It explores neighborhoods at growing distance from the current best solution; we move from this one only if we find a new solution that leads to a better value of the objective function. The solution subspace exploration ends under a condition that can be a time limit or a maximum number of iterations without finding a better solution.

We define N_k as a set of preselected neighborhood structures with $k \in \{1, \ldots, k_{max}\}$, and denote by $N_k(x)$ the set of solutions of the k-th neighborhood of x.

We follow these steps:

Initialization:

- build the structure of neighborhoods N_k with $k \in \{1, \ldots, k_{max}\}$
- find an initial solution x
- fix the condition for the exploration ending

Do until ending condition is reached:

1. $k := 1$
2. do until $k \leq k_{max}$
   (a) *shaking (diversification)*: generate an $x' \in N_k$ at random
   (b) *local search (intensification)*: apply a local search method starting from x' and find solution x''
   (c) *moving*: if the optimum value founded in step 2.b is better than the current one x, fix $x := x''$ and continue the search starting from N_1 ($k := 1$), else increase k by setting $k := k + 1$

Figure 3.1: a VNS method example in two dimensions

3. since we have reached a value of k greater than k_{max}, fix $x := x''$ even if x'' is worse than x and continue the search starting from here

We can notice how the two phases idea exposed in 3.3 has been used in this algorithm at steps 2.a and 2.b. Figure 3.4.1 illustrates an example for a single variable problem. We start from the local optimum x: consider the neighborhood N_1 and randomly select a solution called $x_1^{'} \in N_1$. Apply the iterative algorithm falling on x, that is $x_1^{''} = x$. Since we would like to find a better solution searching in another region we consider a second neighborhood N_2 at increasing distance from x. Again we randomly choose $x_2^{'} \in N_2$ and again we arrive at the initial solution without an improvement. So we move to N_3 using the same procedure as above but this time we get a new local minimum $x_3^{''} \in N_3$ which is also the global one. At this time we consider this new solution as the best current one, that is $x := x_3^{''}$ and repeat the previous steps.

3.4.2 VNS extensions

A well known variant of this method is the *Variable Neighborhood Descent (V.N.D.)*. In this case we apply the increasing distance neighborhood concept also to the local search (step 2.b). The idea is that a local minimum for a neighborhood, i.e. $N_1'(x)$, is not necessary minimum for another neighborhood. So, while the VNS ends its local search as soon as the first minimum is reached in $N_1'(x)$, the V.N.D. tries to find a better solution in $N_2'(x)$ or $N_3'(x)$ and so on until finds a solution better than the current one or an ending condition is verified. In this last case the solution will obviously be the same x.

The steps that the V.N.D. follows in the i-th *intensification* phase are:

Initialization:

- build up the structure of neighborhoods N_k' with $k \in \{1, \ldots, k'_{max}\}$
- take as initial solution the random solution $x \in N_i(x^*)$ given by the *diversification* phase (with x^* current solution)

Iterative algorithm:

1. $k := 1$

2. do until $k \leq k'_{max}$
   (a) *exploration*: explore the neighborhood finding the best solution $x' \in N'_k(x)$
   (b) *moving*: if the optimum value found in the previous step is better than the current one x, fix $x := x''$ and continue the search starting from N'_1 ($k := 1$), else increase k by setting $k := k + 1$
3. consider x as the local optimum

Another variant of the VNS that will be used is the *Reduced Variable Neighborhood Search R.VNS*.

The first issue is that in the diversification phase (step 2.a of the VNS), instead of taking a random solution $x' \in N_k(x)$ we take the best solution between l random solutions, where l is a parameter of the algorithm. So the choice of the new solution is more clever than the blind one and leads to a new starting point that will hopefully be not too bad, avoiding a lot of computational extra time.

The second point is that we choose two parameters: k_{min} and k_{step} to control the neighborhood changing. So we will fix $k := k_{min}$ instead of $k := 1$ at the 1.a step of the VNS and $k := k + k_{step}$ at the 1.c. The advantage is that we can implements the intensification and diversification phases in a very similar way by simply choosing different values of these two parameters. With high values we do the diversification forcing the algorithm to visit regions far from the current one. With low values we do the intensification since we look at near possible solutions subsets.

These two parameters can be chosen considering the problem at hand. For example, in some network problems the minimum change we can have between two solutions is 2 components, so $k_{step} := 2$.

3.4.3 The pseudo-code

Now take a look to the pseudo-code of the algorithm. We are going to use ILOG-Cplex 7.0 as a black-box in the spirit of what we said in paragraph 3.1. See section 2.1 for a brief introduction to this tool.

Algorithm 2 : VNS pseudo-code

```
1: function VNS()
2: set Cplex parameters for general search;
3: build the initial model and solve it with Cplex, stopping at the first feasible
   solution;
4: best_solution:=first_feasible_solution;
5: start_solution:=best_solution;
6: /* use R.VNS extension */
```

```
7: k_1 := k_min;
8: k_2 := k_1 + k_step;
9: while execution_time ≤ maximum_time do
10:   /* diversification */
11:   add the diversification constraint k_1 ≤ △(x, start_solution) ≤ k_2;
12:   set Cplex parameters for diversification;
13:   solve the current model stopping at the l-th feasible solution (or at the
      maximum allowed time);
14:   /* local search algorithm */
15:   k' := k_min;
16:   while (k' ≤ k'_max) and (execution_time≤maximum_time) do
17:     add to the model the constraint △(x, current_solution) = k' and set
        Cplex parameters for intensification;
18:     solve the current model;
19:     if new_current_solution is better than current_solution then
20:       current_soluton:=new_current_solution;
21:       k' := k'_min;
22:     else
23:       k' := k' + k'_step;
24:     end if
25:     remove the intensification constraint;
26:   end while
27:   remove the diversification constraint;
28:   new_local_solution:=current_solution;
29:   if new_local_solution is better than best_solution then
30:     best_solution:=new_local_solution;
31:     start_solution:=best_solution;
32:     add the tabu constraint △(x, start_solution) ≥ 1;
33:     k_1 := k_min;
34:     k_2 := k_1 + k_step;
35:   else
36:     if k_1 < k_max then
37:       k_1 := k_1 + k_step;
38:       k_2 := k_1 + k_step;
39:     else
40:       start_solution:=new_local_solution;
41:       add the tabu constraint △(x, start_solution) ≥ 1;
42:       k_1 := k_min;
43:       k_2 := k_1 + k_step;
44:     end if
45:   end if
46: end while
47: output best_solution;
```

3.5 Local branching method

3.5.1 The method

Local branching is a recent method proposed by M. Fischetti and A. Lodi in 2002 in [**?**]. As its name suggests, the local branching constraint, on which the method is based, can be used as a branching criterion within an enumerative scheme for the model (P) exposed in 3.2. Indeed, given the incumbent solution $\bar{x}$, the solution space associated with the current branching node can be partitioned by means of the disjunction

$$\Delta(x, \bar{x}) \leq k \quad \text{(left branch)} \quad \textbf{or} \quad \Delta(x, \bar{x}) \geq k + 1 \quad \text{(right branch)} \tag{3.8}$$

As to the neighborhood-size parameter k, it should be chosen as the largest value producing a left-branch subproblem which is likely to be much easier to solve than the one associated with its father. The idea is that the neighborhood $\mathcal{N}(\bar{x}, k)$ corresponding to the left branch must be "sufficiently small" to be optimized within short computing time, but still "large enough" to likely contain better solutions than $\bar{x}$. According to computational experience, the choice of k is seldom a problem by itself, in that values of k in range $[10, 20]$ proved effective in most cases. The local branching philosophy is quite different from the standard one: here we do not want to force the value of a fractional variable as happens in the classic branching, but we rather instruct the solution method to explore first some promising regions of the solution space. The expected advantage of the local-branching scheme is an early (and more frequent) update of the incumbent solution. In other words, we expect to find quickly better and better solutions until we reach a point where local branching cannot be applied anymore, hence we have to resort to tactical branching to conclude the enumeration. Note that the algorithm leads to an overall structure with shape of a tree, as depicted in figure 3.2. Every node represents a subproblem solved with a commercial general purpose solver such as ILOG Cplex or the GNU GLPK described in sections 2.1 and 2.2 respectively.

In order to enhance the basic idea we could modify it with a couple of features. The first improvement is related to the fact that, in some cases, the exact solution of the left-branch node can be very time consuming for the value of the parameter k at hand. Hence, from the point of view of a heuristic, it is reasonable to impose a time limit for the left-branch computation. In case the time limit is exceeded, we have two cases.

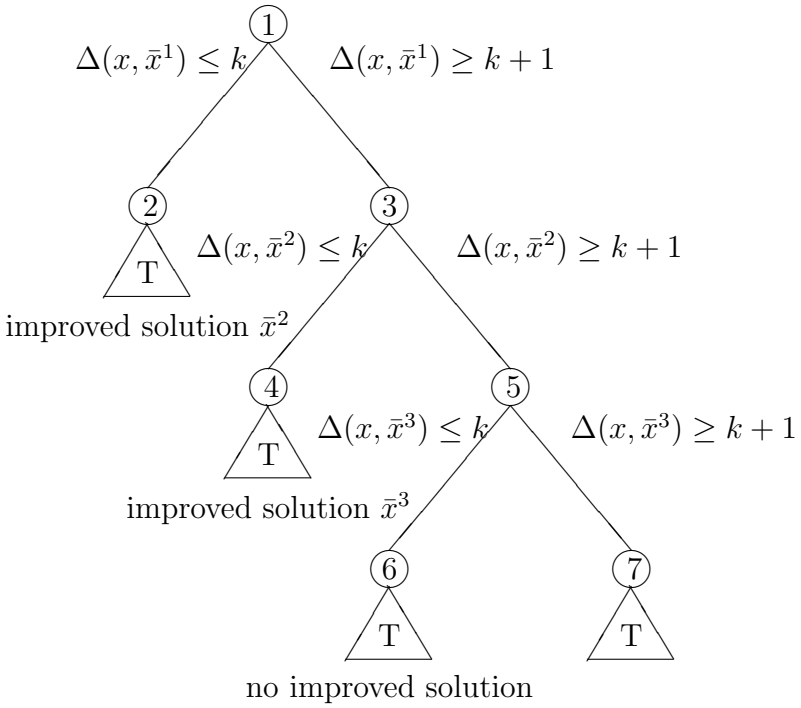

Figure 3.2: The basic local branching scheme.

(a) If the incumbent solution has been improved, we backtrack to the father node and create a new left-branch node associated with the new incumbent solution, without modifying the value of parameter k.

(b) If the time limit is reached with no improved solution, instead, we reduce the size of the neighborhood in the attempt of speeding-up its exploration. This is obtained by dividing the value of k by a given factor $\alpha > 1$.

A further improvement of the heuristic performance of the method can be obtained by exploiting well-known diversification mechanisms borrowed from local search metaheuristics. In the local branching scheme, diversification is worth applying whenever the current left-node is proved to contain no improving solutions. In order to keep a strategic control on the enumeration even in this situation, we use two different diversification mechanisms. We first apply a "soft" diversification consisting in *enlarging* the current neighborhood by increasing its size by a factor $\beta > 1$. Diversification then produces a left-branch node which is processed by tactical branching within a certain time limit. In case no improved solution is found even in the enlarged neighborhood, we apply a "strong" diversification

step, in the spirit of *Variable Neighborhood Search* described in section 3.4. Here, we look for a solution (typically worse than the incumbent one) which is not "too far" from $\bar{x}^2$, e.g., a feasible solution x such that

$$\Delta(x, \bar{x}^2) \leq \lfloor \beta^2 k \rfloor \tag{3.9}$$

3.5.2 The pseudo-code

Algorithm 3 : Local Branching pseudo-code

```
1: function locBra(k,time_limit,node_time_limit, dv_max,x*)
2: rhs:=bestUB:=UB:=TL:=+∞; x*:=undefined
3: opt:=true  /*optimization status*/
4: first:=true  /*stop at first feasible sol*/
5: diversify:=true /*perform strong diversification*/
6: repeat
7:    if (rhs< ∞) then
8:       add the local branching constraint △(x, x̃) ≤rhs
9:    end if
10:   TL:=min{TL,total_time_limit-elapsed_time}
11:   stat:=MIP_SOLVE(TL,UB,first,x̃)
12:   TL:=node_time_limit
13:   if (stat=opt_sol_found) then
14:      /*optimum found for current MIP*/
15:      if cost(x̃) <bestUB then
16:         bestUB:=cost(x̃)
17:         x*:=x̃
18:      end if
19:      if (rhs≥ +∞) then
20:         return opt
21:      end if
22:      reverse the local branching constraint into △(x, x̃) ≥rhs+1
23:      diversify:=first:=false
24:      x̄ := x̃; UB:=cost(x̃); rhs:=k
25:   end if
26:   if (stat=proven_infeasible) then
27:      /*infeasible problem: MIP current has no feasible solution bet-
         ter than UB*/
28:      if (rhs≥ +∞) then
29:         return opt
30:      end if
31:      reverse the local branching constraint into △(x, x̃) ≥rhs+1
32:      if (diversify) then
```

```
33:         UB:=TL:=+∞; dv++; first:=true
34:       end if
35:       rhs:=rhs + k/2; diversify:=true
36:     end if
37:     if (stat=feas_sol_found) then
38:       /*feasible solution found: improving the reference solution*/
39:       if (rhs< ∞) then
40:         if (first) then
41:           delete the last local branching constraint △(x, x̃) ≤rhs
42:         else
43:           replace the last local branching constraint △(x, x̃) ≤rhs by △(x, x̃) ≥1
44:         end if
45:       end if
46:       /*compute the optimal solution with the MIP solver*/
47:       REFINE(x̃)
48:       if (cost(x̃)<bestUB) then
49:         bestUB:=cost(x̃); x*:=x̃
50:       end if
51:       first:=diversify:=false; x̄ := x̃; UB:=cost(x̃); rhs:=k
52:     end if
53:     if (stat=no_feas_sol_found) then
54:       /*no feasible solution found for current node*/
55:       if (diversify) then
56:         /*strong diversification*/
57:         replace the last local branching constraint △(x, x̃) ≤rhs by △(x, x̃) ≥1
58:         UB:=TL:=+∞; dv++; rhs:=rhs+k/2; first:=true
59:       else
60:         /*strong diversification*/
61:         delete the last local branching constraint △(x, x̃) ≤rhs
62:         rhs:=rhs-k/2
63:       end if
64:       diversify:=true
65:     end if
66: until (elapsed_time>total_time_limit) or (dv > dv_max))
67: TL:=total_time_limit - elapsed_time; first:=false
68: stat:=MIP_SOLVE(TL,bestUB,first,x*)
69: opt:=(stat=opt_sol_found) or (stat=proven_infeasible)
70: return opt
```

Function `LocBra` receives on input the neighborhood size (k), the overall time limit (`total_time_limit`), the time limit for each tactical branching exploration (`node_time_limit`), and the maximum number of diversifications allowed

(`dv_max`). It returns on output the best/optimal feasible solution found (x^*) along with the final optimization status (opt).

Chapter 4

Diversification-Refining-Tight refining method

4.1 Introduction

So far we have seen some important ideas that can help us to solve large MIPs. We have learn how to explore the solutions space without considering a solution more than once (section 3.3) and how to build neighborhoods for a more clever search using a two phases procedure (section 3.4). Now we introduce another important ingredient to obtain a metaheuristic algorithm that will use also all these features.

Trying to solve a very-large problem taking it as a whole can be very time expensive, hence the interest in dividing it into smaller subproblems. Each one of them will take a typically short computational time, so as to get a good solution in a very fast way. The underlaying idea is in effect quite simple: we split the set of binary variables into two subsets. We call each subset a level and so we have *first level* and *second level* variables. The set of subproblems is made by setting the first level variables in different configurations. So, if L_1 denotes the first level subset, we have up to $2^{|L_1|}$ subproblems. It is clear that the cardinality of first level set should be sufficiently small to have a reasonable number of subproblems. This division can be left to the user which can exploit its knowledge of the problem to split the variables in a clever way, or to an algorithm that bases its decision upon information given by the constraints.

The method we are going to describe is very general and can solve problems in the form of (P) introduced in section 3.2. We will see that if no first level set is provided (or cannot be automatically detected) the algorithm enters VNS mode while if no binary variables are given the Cplex solver is directly applied to the whole model. We will test the algorithm using instances related to telecommunication networks design.

4.2 Splitting of the variables

Now we are going to see some methods to build up the 1st level variables set in an automated way. All algorithms let the user force some variables to belong to the first level by specifying them in a text file on the form

```
var1
var2
  ⋮
varn
\end
```

The algorithm also checks if each variable belongs to the model and is binary, so as to prevent some user's error. For practical reasons we will denote with y the first level variables and with x the second level ones. Note that since all constraints treatment is done by Cplex using the extractable objects (such as constraints and variables), the methods are file-type independent, so we can process any file type supported by Cplex. Another advantage in using Cplex interface towards the model is that a lot of effort in file parsing is left to the ILOG application, while the programmer can focus his attention on the logical structure.

4.2.1 The forced-variables method

A very important characteristic of the 1st level variables is that they can force the values of some 2nd level ones. For example, take $x_{1,1}$ and $x_{2,1}$ two 2nd level variables and y_1 at the first level. In many location problems we can see the x's as links from terminals or users that request a service from some provider. If provider 1 is not activated ($y_1 = 0$) both links can't be used, so this choice for y_1 forces to zero also the other two variables. In terms of constraints we can formulate this condition as

$$x_{1,1} + x_{2,1} \leq y_1$$

Generalizing this example we have a constraint like:

$$\sum_{i \in N} c_{ij} x_{ij} \leq \sum_{j \in M} T_j y_j \tag{4.1}$$

Figure 4.2.1 represents the situation with one 1st level variable y_1 and n 2nd level variables.

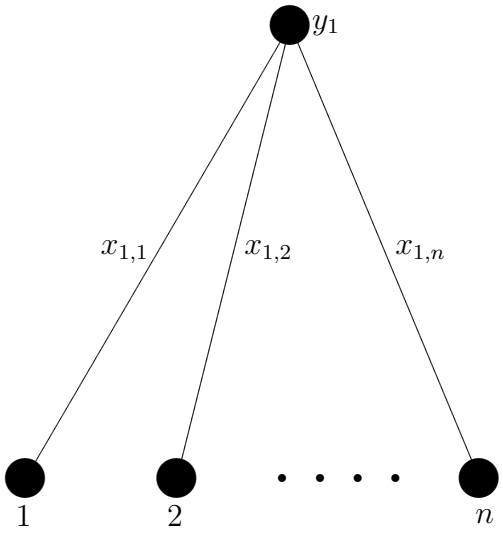

Figure 4.1: n 2nd level variables forced by one 1st level.

We base our method on an equivalent form of the constraint 4.1:

$$\sum_{j \in M} T_j y_j - \sum_{i \in N} c_{ij} x_{ij} \geq 0 \tag{4.2}$$

The algorithm is quite simple and very efficient and is made of two nested for loops. The first scans all constraints while the second is done inside each constraint, considering the coefficient of the variables involved:

Algorithm 4 : Splitting method based on forced-variables

```
1: initialize L_1 reading variables from a file (if given);
2: for all constraints C in form 4.2 do
3:   for all variables var in C  do
4:     number_variables = no_binary = num_second_lev = 0;
5:     L'_1 := ∅
6:     if var∈ L_1 then
7:       continue;
8:     end if
9:     number_variables++;
10:    if var is not binary then
11:      no_binary++;
```

```
12:         break;
13:       end if
14:       if coeff(var) > 0 then
15:         add var to L'_1
16:       else
17:         num_second_lev++;
18:       end if
19:     end for
20:     if number_variables=no_binary then
21:       continue;
22:     end if
23:     if |L'_1|+num_second_lev > 1 then
24:       add variables in L'_1 to L_1
25:     end if
26: end for
27: return L_1
```

After the initialization of the first level set L_1, we scan each constraint in the model. For each constraint, we consider the variables var with nonzero coefficient. If var is already in L_1, we jump to the next one since no new information can be extracted. If all variables involved in the constraint aren't binary, again jump to the next constraint because we are sure that here we can't find new 1st level variables. We put potential 1st level variables in L'_1 basing on the coefficient sign: from equation 4.2 we have positive sign for the first level. If the number of variables is greater than 1 (to prevent constraints which are just bounds), we put all L'_1 elements in L_1. The function returns the L_1 set.

A more theoretical approach to the task is the following. We consider constraints in the form:

$$\sum_j \alpha_j x_j \leq \alpha_0 \qquad (\text{or} = \alpha_0) \tag{4.3}$$

Assuming that each variable x_j is bounded by $LB_j \leq x_j \leq UB_j$, the minimum value for the left-hand side term in (**??**) is obtained when all variables with a positive coefficient α_j are set to their lower bound LB_j, whereas all variables with a negative coefficient α_j are set to their upper bound UB_j. With this variable setting, the total slack for the constraint attains its maximum value:

$$\delta_{max} := \alpha_0 - (\sum_{j:\alpha_j>0} \alpha_j LB_j + \sum_{j:\alpha_j<0} \alpha_j UB_j) \tag{4.4}$$

Therefore, our condition to insert a binary variable x_j into the first level set $\mathcal{B}_1$ is:

$$\alpha_j < 0 \ : \ |\alpha_j|(UB_j - LB_j) \geq \delta_{max} \tag{4.5}$$

as in this case setting $x_j = 0(= LB_j)$ implies that all other variables with a nonzero coefficient α_j are forced to keep their (upper or lower bound) value as in (**??**).

4.2.2 The big M method

We are going to show a method for variables splitting based on the presence of coefficients much higher than the others (big M). We can find them in bad modeled problems or in linearized models. In fact, if there are no-linear constraints, we can often linearize them by introducing big M coefficients. There are two ways to say if a coefficient is a big M or not: the first is *absolute*, that is we have a coefficient which is much larger than all the others. This case can occur especially in linearizations. The pseudo-code is:

Algorithm 5 : Splitting method based on absolute big M

```
1: initialize L_1 reading variables from a file (if given);
2: for all constraints C do
3:    for all variables var in C  do
4:       if |coeff(var)| > bigM and var is binary and var ∉ L_1 then
5:          add var to L_1
6:       end if
7:    end for
8: end for
```

The main disadvantage of this method is that it not considers the different kind of values: for example 1000 meters can be viewed as a big value while 1 Km would be taken as a small one but clearly they are the same thing. Again, we have different classes of constraints inside a model and so a value can be a big M for a class but not for another one. We can supply to this last trouble with a second method which can be called *relative*. Fix a gap in percentage (call it for example *bigMperc*); find a reference value which captures the trend of all coefficients inside a constraint (for example the average) and then compare all coefficients to it. If someone is higher than the *bigMperc* of it we say that have found a big M coefficient. The way we choose the reference value is the key for a good identification of variables with big M value. The pseudo-code is:

Algorithm 6 : Splitting method based on the average

```
1: initialize L_1 reading variables from a file (if given);
2: for all constraints C do
3:    for all variables var in C  do
4:       if var is not binary then
5:          no_binary++
6:       end if
7:       tot+=|coeff(var)|
8:       num_variables++
9:    end for
10:   if num_variabels==no_binary then
11:      /* no 1st level can be found here; jump to next constraint */
12:      continue
13:   end if
14:   aver:=tot/num_variables
15:   for all variables var in C  do
16:      if |coeff(var)|>(1+bigMperc)·aver AND var ∉ L_1  then
17:         add var to L_1
18:      end if
19:   end for
20:   tot = num_variables = 0
21: end for
```

The reason why we put these variables at the first level is that they can cause a slow down in the branching procedure forcing to a very deep investigation in the branch tree. In fact, consider for example a constraint like:

$$\sum_{i \in S} c_i x_i \leq My \tag{4.6}$$

with $c_i \ll M$. During the relaxation the integrality constraint is not applied and so this cut is satisfied even by very low y values. So, several subproblems may be added to the branching tree to find out an integer solution.

4.3 The resolution method

Let's recall the MIP formulation that we have seen in paragraph 3.2:

$$(P) \qquad \min c^T x \tag{4.7}$$

$$Ax \geq b \tag{4.8}$$

$$x_j \in \{0,1\} \qquad \forall j \in \mathcal{B} \neq \emptyset \tag{4.9}$$

$$x_j \geq 0, \text{ integer} \quad \forall j \in \mathcal{G} \tag{4.10}$$
$$x_j \geq 0 \quad \forall j \in \mathcal{C} \tag{4.11}$$

We split the binary variables in two subsets named L_1 for the first level and L_2 for the second level (as seen in section 4.2). So we partition $\mathcal{B}$ respectively $\mathcal{B}_1$ and $\mathcal{B}_2$. For each possible first level variables configuration we have:

$$(P_k) \qquad \min c^T x$$
$$A_2 x_2 \geq \tilde{b} \tag{4.12}$$
$$x_j \in \{0,1\} \quad \forall j \in \mathcal{B}_2 \tag{4.13}$$
$$x_j = \bar{x}_j \quad \forall j \in \mathcal{B}_1 \tag{4.14}$$
$$x_j \geq 0, \text{ integer} \quad \forall j \in \mathcal{G} \tag{4.15}$$
$$z_j \geq 0 \quad \forall j \in \mathcal{C} \tag{4.16}$$

where A_2 and x_2 are the components of coefficients matrix A and variables vector x related to all non first level variables and $\tilde{b} = b - A_1\bar{x}$ if A_1 denotes the matrix coefficients part regarding the first level variables. Each problem is characterized by

We start from a solution, in the hope that we can obtain a better solution keeping the current first variables configuration. During this phase, called *refining*, we add the constraint[1]

$$\sum_{j \in \mathcal{B}_1(1)} (1 - x_j) + \sum_{j \in \mathcal{B}_1(0)} x_j = 0 \tag{4.17}$$

imposing that configuration of $\bar{x}_j \ \forall j \in \mathcal{B}_1$ never changes. The structure of this inequality is very similar to that we have seen for the distance definition (see 3.2): in effect we are imposing that the number of the changes between two configurations is zero. Now we proceed with the solution of this subproblem using Cplex.

If the problem is still too large and its solution takes more than a given time, we enter in the *tight-refining* phase. This is an iterative algorithm that investigates the current subspace adding constraints like

$$\sum_{j \in \mathcal{B}_2(1)} (1 - x_j) + \sum_{j \in \mathcal{B}_2(0)} x_j = k \tag{4.18}$$

so we follow the same idea of increasing neighborhoods seen in the VNS for the intensification by adding k_{step} to the k value at each iteration. It stops when the termination condition arises that is when k has reached a maximum allowed value k_{max} or the maximum time for this phase is elapsed. If Cplex can't find a

[1] we define $\mathcal{B}_i(w)$ as the set of the variables of level i fixed at the w value. In our case $i \in \{1,2\}$ and $w \in \{0,1\}$

solution which leads to a better objective function value, the current solution will be marked as the best one for the current configuration. We remove constraints 4.14 and 4.15.

Now we have to consider another configuration for the first level variables so we apply a *diversification* phase as seen in the VNS method adding the diversification constraint

$$k_1 \leq \sum_{j \in \mathcal{B}_1(1)} (1 - x_j) + \sum_{j \in \mathcal{B}_1(0)} x_j \leq k_2 \tag{4.19}$$

As the tabu search method tells us, a random choice of the new configuration can lead to solving twice a yet visited scenario. To avoid this risk we add a tabu constraint like

$$\sum_{j \in \mathcal{B}_1(1)} (1 - x_j) + \sum_{j \in \mathcal{B}_1(0)} x_j \geq 1 \tag{4.20}$$

From here we start again with the refining phase in order to find the best solution for the current settings. If we find a solution cost better than the previous one we consider this solution as the new global optimum.

We can summarize the procedure in this scheme:

1: **repeat**
2: fix the first level variables adding 4.14
3: solve the subproblem with Cplex adding constraints 4.15 for increasing neighborhoods investigation
4: **if** current solution is better than the previous global one **then**
5: global solution:=current solution
6: **end if**
7: remove previous added constraints
8: build the new subproblem with the diversification 4.16 and the tabu 4.17 constraint
9: **until** ending condition is verified

4.3.1 DRT variation

As we will see in the computational section 5, the way we choose the parameters (especially the time limit for each phase) can cause the method to emphasize one of the two search sides. The local searching is preferred if we have a small number of diversifications exploring a little set of 1st level variable configurations in a very deep way. If we allow for a great number of diversifications, we expect to examine as much configurations as possible and so we should spend less time in the local search phase. This case can be very useful, for example, when we have very large cardinality sets L_1 .

Applying the same reasoning as above, we can control the subspaces visited by the algorithm to prevent that for a too long time we spend our resources ex-

ploring configurations that lead to not significant results, i.e., the improvement at each diversification is not significant. When for a too long time we have not improvement in solution objective better than a given percentage of the last improvement, we can force the algorithm to make a *big diversification* step changing a great number of 1st level set variables. Obviously, we have to add a tabu constraint also for this phase to avoid a new search for this configuration. We can decide the maximum number of big diversifications, the number of 1st level variables to change, the gap to consider a solution as an improvement or not and the maximum number of no improvement before the big diversification starts.

4.3.2 The pseudo-code

Now we take a look to the pseudo-code of the algorithm with the variation introduced in the previous section.

Algorithm 7 : DRT2 pseudo-code

```
1: function DRT()
2: read algorithm parameters defined by the user from a file
3: /* search for start solution */
4: set the solver parameters for initial search
5: start_solution=solve()
6: if solver can't find a start solution then
7:    print(No solution has been found)
8:    return
9: end if
10: if start_solution is optimal then
11:    output(start_solution)
12:    return
13: end if
14: current_solution=best_solution=start_solution;
15: while (execution_time < time_limit) and (num_div < max_div_number) and
    (num_big_div < max_big_div_number) do
16:    if num_div_with_no_improv==max_div_with_no_improv then
17:       /***** big diversification *****/
18:       num_div_with_no_improv=0;
19:       num_big_div++;
20:       add the tabu constraint for big diversification;
21:       set solver parameters for diversification;
22:       k^big_1,div = k^big_min,div
23:       k^big_2,div = k^big_1,div + k^big_step,div
24:       repeat
25:          add the constraint for diversification with k^big_1,div and k^big_2,div;
26:          if new_sol then
```

current_solution=solver_solution;
if current_solution is better than best_solution **then**
best_solution=current_solution
end if
else
$k^{big}_{1,div} += k^{big}_{step,div} + 1$
$k^{big}_{2,div} = k^{big}_{1,div} + k^{big}_{step,div}$
end if
remove the constraint for big diversification;
num_big_div++;
until (execution_time=time_limit) **or** (new_sol)
/* end of big diversification */
if execution_time>time_limit **then**
break
end if
else
/*** diversification *****/**
add the tabu constraint for diversification;
set the solver parameters for diversification;
$k_{1,div} = k_{min,div}$
$k_{2,div} = k_{1,div} + k_{step,div}$;
repeat
add the constraint for diversification with $k_{1,div}$ and $k_{2,div}$;
if new_sol **then**
current_solution=solver_solution
if current_solution is better than best_solution **then**
best_solution=current_solution
end if
else
$k_{1,div} += k_{step,div} + 1$
$k_{2,div} = k_{1,div} + k_{step,div}$
end if
remove the constraint for diversification;
num_div++;
until (execution_time=time_limit) **or**(new_sol)
end if
/* now 1st level variables are fixed */
if execution_time>time_limit **then**
break
end if
/*** refining *****/**
add the constraint for refining;

```
69:  set the solver parameters for refining;
70:  solve the model;
71:  if no new solution found then
72:    /* try refining with MIPEmphasis=1 */
73:    set the solver MIPEmphasis=1;
74:    if no new solution found then
75:      current_solution.status=solver.getStatus()
76:    end if
77:  end if
78:  if new solution found then
79:    current_solution=solver_solution;
80:    if current_solution is better than the best one then
81:      best_solution=current_solution
82:    end if
83:  end if
84:  if current_solution is infeasible then
85:    /* Solver has not found a solution because best solution (cut
       upper/lower limit) was better than lower/upper bound */
86:    remove the constraint for refining
87:    /* skip to next diversification */
88:    continue
89:  end if
90:  if there aren't 2nd level variables then
91:    /*no tight refining can be done*/
92:    remove the refining constraint
93:    continue;
94:  end if
95:  if execution_time>time_limit then
96:    break
97:  end if
98:  /***** tight refining *****/
99:  repeat
100:   add the tight refining constraint;
101:   set the solver parameters for tight refining;
102:   if a solution better than the current one has been found then
103:     current_solution=solver_solution;
104:     k_tight = k_min,tight
105:   else
106:     k_tight+ = k_step,tight
107:   end if
108:   remove the tight refining constraint;
109:  until (execution_time=time_limit) or (k_tight > k_max,tight)
110:  if current_solution is better than the best one then
```

```
111:      best_solution=current_solution
112:    end if
113:    remove the tight refining constraint;
114:    /* end of refining and tight refining */
115:    if best_solution is better of gap_percent than the one of the previous diversification then
116:      num_div_with_no_improv=0
117:    else
118:      num_div_with_no_improv++
119:    end if
120: end while
121: output(best_solution)
```

`solve()` denotes that we invoke the solver used as a black-box and the result is stored in `solver_solution` or in `current_solution`. The keyword **break** as usual tells that we have to exit from the current loop; on the other hand **continue** forces to jump to the next cycle iteration.

Basically we can divide the pseudo-code in five parts: start search, big diversification, diversification, refining and tight-refining.

1. (lines 2 to 13) search for a starting solution. If no solution can be found or we reached an optimal one, exit;

2. (line 15) the condition for the algorithm termination: it is based on time limitation and maximum number of big/small diversifications

3. (lines 16 to 38) perform the big diversification if no significant improvements took place; this phase, very similar to the diversification, ends as soon as when the first feasible solution is reached, checking the boolean variable `new_sol`; the (big) diversification constraint is the 4.16

4. (lines 39 to 41) check the running time: quit if it is greater than the limit;

5. (lines 43 to 61) perform the diversification; as in the big diversification phase;

6. (lines 43 to 61) perform the refining: once we have fixed the first level variables we try to catch the optimal solution for this configuration adding constraints like 4.14. If we can't, we try to take a feasible solution setting the MIPEmphasis[2] parameter to 1. If there isn't a feasible solution jump to next cycle. Always keep track of the solution status;

[2]this step can be done only if the solver has a parameter with this function. Note that for example the GLPK has not.

7. (lines 90 to 96) check if there are 2nd level variables: if is not so, tight-refining can't be done and so we jump to the next iteration. If we can continue with next phase, check if time limit is reached;

8. (lines 98 to 112) perform tight-refining with an iterative procedure. The tight-refining constraint is 4.15.

9. (lines 114 to 118) check if a significant improvement took place; if not, increase the number of diversifications with no improvement for the big diversification test.

Since the ending condition is based on the time limit we could not try all the possible y configurations. We have to check this clause also after every phase to avoid the reaching of an execution time too greater than the maximum allowed one. Note that during the refining phase we set MIPEmphasis to 1 after no feasible solution has been found since this value forces Cplex to generate as many good intermediate feasible solutions as possible (see section 2.1.3). Every time we perform a comparison between two solutions we have to check also the direction of the objective function (i.e. maximize or minimize): in this way the algorithm can be used for both cases.

4.3.3 The overall scheme

As we have noticed before, the DRT method can be applied only when 1st level variables can be found. If it is not the case but we have however binary variables (that go obviously to the 2nd level) we can perform the VNS method. If the problem has no binary variables at all we have to rely on our MIP commercial solver giving to it the whole model. The scheme surrounding the whole solution process can be depicted like this:

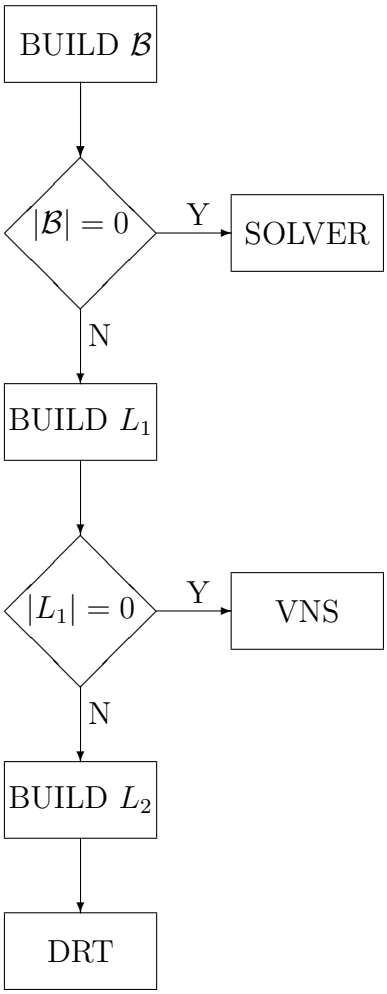

Figure 4.2: The overall scheme.

4.3.4 The implementation

The algorithm has been implemented in C++ using the ILOG Cplex Concert Technology 1.0 as described in section 2.1. The package comes as a set of files:

- `DRT.h`: the header with all variable definitions. There are also the input and output file names definitions (see forward).
- `DRTsolve.h`: a set of routines for the DRT implementation. So we have `DRT_method`, `VNS_method` and `NO_NEIGH_method`. The latter is the simple application of Cplex to the whole problem when no binary variables are

present. The synopsis for these routines is:

void DRT_method(IloCplex& cplex, IloModel& model, char *FILE_LEVEL1)
void VNS_method(IloCplex& cplex, IloModel& model)
void NO_NEIGH_method(IloCplex& cplex, IloModel& model)

- `split.h`: contains the two methods for splitting variables. To switch from the forced variables method to the big M one, we have just to tell to the splitting routine what we want to use. Loosely speaking, the routine declaration is

```
NumVarArray
split_var(IloEnv env,IloModel model,IloCplex cplex&,
NumVarArray variables,int type,char *file_first_lev);.
```

  The `type` argument specifies the splitting method (1 = forecd variables, 2 = big M); the last parameter is optional and tells what is the file where the user's defined first level set is stored.

- `input.cpp`: read parameters for each method from the respective files.

The files where all parameters are written (read by input.cpp) are stored in the DRT.exe directory. Their names are: `_cplexpar_DRT.dat`, `_cplexpar_VNS.dat` and `_cplexpar_NO_NEIGH.dat`. By default, files `_out.dat` and `_finalsol.dat` are created in the same directory. The first reports a log of the last run of the algorithm and the second displays information about the solution found (objective value, variables, time and so on). If we want to change this scenario we may modify the settings in the `DRT.h` file.

4.4 Applications and developments

As we can see, there are several possible variants to the basic algorithm. For example, starting form the one shown is section 4.3.1, we could modify the way we define the increasing judgement criterium taking as the new reference solution the best found in the last n iterations without improvements. Another extension with the goal of minimizing the influence of wrong settings made by the user could be an auto-tuning algorithm. The idea is that during the first m phases we keep track of the elapsed time for example for a diversification to reach a certain quality of the solution (say the relative gap from the best bound or others). We calculate the average of these times and if it is less than the parameter given by the user, we set it as the new value. Obviously there can be some troubles with it. For example, a too low number of diversifications can occur due to the long time they take to close the gap (which can be even too restrictive) and so no new

setting can be done, or our choice for the bound gap can be a too pessimistic. So this method could be more appropriated for problems with fast phases, closer to the spirit of the DRT A fascinating work could be to try to relate the parameters of the algorithm to some properties of different classes of problems. In this way we could develop some pre-defined settings to achieve the best performance once we know the nature of the problem at hand. For the splitting method, we could try to refine the big M method adopting reference values which differ from a simple average. Also we could perform a kind of sensitive analysis to find out what are the variables to put to the 1st level or an analysis at relaxation time.

From the applications point of view, we could exploit the first level fixing method to solve bilinear problems. They are MIP with constraints and objective function which contain also expressions like

$$\sum_{i \in I} \sum_{j \in J} y_i \cdot x_j \tag{4.21}$$

As we said, these kinds of models are usually solved with linearization. This method is useful to exploit the power of MIP solvers like Cplex and to prove in a easier way their optimality. On the other hand, this procedure can increase in a dramatic way the number of constraints, leading to a more difficult problem. For an example of linearization applied to a telecommunications network planning see [**?**]. What we can do with our DRT is to choose the y_i's of a bilinear constraint like 4.18 as the 1st level ones. So, for each configuration we have to solve a linear problem. We haven't added new constraints at all except the temporary one for diversification, refining and tight-refining.

Chapter 5

Tests results

5.1 Instances

We have performed tests on three different problem classes. A detailed description and formulation of each model is provided in appendix. Now we are just going to give a brief introduction to these instances.

The first problem we deal with is the *Facility Location Problem* (FLP) in its capacitated version which is a classical NP-hard problem in Operative Research. A set of warehouses (or concentrators) have to provide some goods or services to a given set of terminals (N): we want to decide where to place the warehouses choosing between candidates sites (set M) minimizing the cost of the links between them and the terminals, satisfying capacity constraints. The decisions variables are:

$$y_j = \begin{cases} 1 & \text{if a concentrator is installed at location } j \\ 0 & \text{otherwise} \end{cases}$$

for all $j \in M$

$$x_{ij} = \begin{cases} 1 & \text{if terminal } i \text{ is assigned to a concentrator installed at location } j \\ 0 & \text{otherwise} \end{cases}$$

for all $i \in N$, $j \in M$

The second problem concerns the planning of an UMTS network. The entities are BTS's (which can be seen as the terminals of the FLP), CSS's and LE's (both play the role of concentrators). We consider a certain number of potential CSS and LE sites among which the planner has to choose those to be actually activated. We consider a three level star-type UMTS architecture (star/star network), defined by an upper layer made of active LE's (chosen in the given set of potential LE's), a middle layer of active CSS's (also chosen in the given set of potential CSS's), and a lower layer of the given BTS's (each of which is required to play the role of a leaf in the star-type structure. Since we have to satisfy

technical constraints not all links are allowed. BTS's can be connected either to CSS's or LE's. Each activated LE plays the role of the root of a tree spanning a different connected component. Moreover, the problem cannot be decomposed in two independent subproblems consisting of assigning LE's to CSS's and CSS's to BTS's, respectively, in that the choice of the active CSS's and of their traffic load creates a tight link between the two subproblems. Obviously, we need to build up the minimum cost network respecting traffic requirements and technical constraints. Again, the problem is NP-hard (note how it is close to the FLP). The most important decision (binary) variables are:

$$y_j^{CSS-h} = \begin{cases} 1 & \text{if a CSS is installed at location } j \\ 0 & \text{otherwise} \end{cases}$$

for h=1,2 (CSS of type simplex or complex respectively)

$$y_k^{LE} = \begin{cases} 1 & \text{if a LE is installed at location } k \\ 0 & \text{otherwise} \end{cases}$$

$$x_{i,j}^{BTS-CSS} = \begin{cases} 1 & \text{if the BTS in location } i \text{ is assigned to a CSS in location } j \\ 0 & \text{otherwise} \end{cases}$$

$$x_{i,k}^{BTS-LE} = \begin{cases} 1 & \text{if the BTS in location } i \text{ is assigned to a LE in location } k \\ 0 & \text{otherwise} \end{cases}$$

$$x_{i,k}^{CSS-LE} = \begin{cases} 1 & \text{if the CSS in location } i \text{ is assigned to a LE in location } k \\ 0 & \text{otherwise} \end{cases}$$

The third problem is the QCL-C (*Quadratic Capacity Concentrator Location Problem with Complete Routing*), another one very close to the classical FLP. We need to build a two level network: we have the access network for terminals-terminals links and backbone network for concentrators-concentrators connections. The goal is again to build up the minimum cost global network (access plus backbone). Note that no assumptions on the nature of the two networks are taken, so we can face different situations such as star/star or complete/star configurations. This model (complete routing) encompasses them all. It is easy to guess that again we are facing a NP-hard problem. The main decision variables in the formulation we consider (called QCL-C3, proposed by Ernst and Krishnamoorthy in [**?**]) are:

$$y_j = \begin{cases} 1 & \text{if a concentrator is installed at location } j \\ 0 & \text{otherwise} \end{cases}$$

$$x_{i,j} = \begin{cases} 1 & \text{if the concentrator in location } i \text{ is assigned to a the } i\text{-th terminal} \\ 0 & \text{otherwise} \end{cases}$$

For each problem we considered different instances: in table 5.1 report the name of each instance, a reference and some data to characterize the size of the problem. For Yaman's problems, we have the names on the form *ft,N,M,Q*, where N is the number of nodes, M is the capacity of each concentrator and Q is the traffic demand.

NAME	REF.	num. rows	num. cols	$\|L_1\|$	$\|L_2\|$	best known
cap41	[?]	866	816	16	800	1040444.37
cap81	[?]	2575	1275	25	1250	838499.29
cap111	[?]	5100	2550	50	2500	826124.71
A	[?]	473	334	10	308	10760529
B	[?]	681	498	13	455	11619817
C	[?]	809	614	12	562	12393173
D	[?]	972	738	14	676	13376659
E	[?]	1741	1316	20	1232	19734790
F	[?]	2119	1628	23	1515	21950121
G	[?]	2394	1836	24	1704	22971252
H	[?]	2660	2046	26	1900	23977171
I	[?]	3739	2946	30	2772	30088388
L	[?]	4376	3406	33	3191	31096388
M	[?]	4710	3706	34	3464	31977972
N	[?]	4984	4002	36	3742	32833226
O	[?]	8271	6650	50	6200	39182672
ft17101div2	[?]	867	5185	17	272	26274.5
ft17101div3	[?]	867	5185	17	272	18726.6
ft17101div4	[?]	867	5185	17	272	15520.8
ft17101div5	[?]	867	5185	17	272	13902.2*
ft17101div6	[?]	867	5185	17	272	13035.7*
ft17151div2	[?]	867	5185	17	272	23446.0
ft17151div3	[?]	867	5185	17	272	17313.5
ft17151div4	[?]	867	5185	17	272	14899.4*
ft17151div5	[?]	867	5185	17	272	13690.7*
ft17151div6	[?]	867	5185	17	272	12837.7*
ft17152div3	[?]	867	5185	17	272	30671.0
ft17201div2	[?]	867	5185	17	272	21213.0
ft17201div3	[?]	867	5185	17	272	16416.5*
ft17201div4	[?]	867	5185	17	272	14566.5*
ft17201div5	[?]	867	5185	17	272	13661.8*
ft17201div6	[?]	867	5185	17	272	12837.7*
ft17201	[?]	867	5185	17	272	42453.2
ft17202div3	[?]	867	5185	17	272	27760.8

Table 5.1: Instances resume. The * means proven optimal solution.

Other possible models for our method may be the ones related to the planning of UMTS network from the point of view of quality and health requirements as described in [**?**], or again the index selection problem in a database project.

5.2 Comparing the methods

We are going now to show two tables concerning the performance of the three algorithms we have seen in relation of the UMTS network planning as described in section **??**. A pure Cplex application is provided too, as reference. We have four classes of instances for the same problem, which differ for their size. For each class, in the table we give the number of BTS, CSS and LE in the form (num_BTS,num_CSS,num_LE) and the objective function value (the minimum one for each example is in bold face) calculated in *ECU* basing on the costs of devices installation and links activation. We report some significant parameters; for an explanation about Cplex settings see 2.1.3.

Parameters:

TABU SEARCH METHOD[1]:

- time limit: 6, 8, 10, 12 hours for classes 1, 2, 3 and 4 respectively
- Penalty cost for exploration: 2500
- Penalty cost for intensification: 500000

PURE CPLEX 7.0 METHOD:

- time limit (`TiLim`): 6, 8, 10, 12 hours for classes 1, 2, 3 and 4 respectively
- `MIPEmphasis` = 1[2]
- `EpAGap` = 10^{-9}
- `EpGap` = 10^{-9}
- `TreLim` = 128 MB

VNS METHOD:

Global parameters

- time limit: 6, 8, 10, 12 hours for classes 1, 2, 3 and 4 respectively
- maximum number of diversifications = 1000
- memory limit = 128 MB

[1] in the implementation proposed in [**?**] Cplex is not used so all constraints and variables management is handled by the programmer

[2] with this setting we have results better than with the zero one since the solver is forced to generate as many feasible solution as possible during the generation of the branching tree and so there is an high chance to fall in a solution which is better than with MIPEmphasis=0

- `NodeFileInd=1`(save nodes on a file when the tree size exceeds the memory limit)

Parameters for initial research

- `TiLim` = 60'
- `NodeLim` = 5000000 (not bounded)
- `IntSolLim` = 1
- `MIPEmphasis` = 1

Parameters for diversification:

- `TiLim` = 3', 5', 7', 10' for classes 1, 2, 3 and 4 respectively
- `NodeLim` = 5000000 (not bounded)
- `MIPEmphasis` = 0
- `EpAGap` = 10^{-9}
- `EpGap` = 10^{-9}
- `CutUp` = cost of current solution
- $k'_{min} = 2$
- $k'_{max} = 6$
- $k'_{step} = 2$

Parameters for intensification:

- `TiLim` = 30'
- `NodeLim` = 5000000 (not bounded)
- `IntSolLim` = 7
- `MIPEmphasis` = 1
- $k_{min} = 50$
- $k_{max} = 200$
- $k_{step} = 50$

DRT METHOD:

Global parameters

- time limit: 6, 8, 10, 12 hours for classes 1, 2, 3 and 4 respectively
- maximum number of diversifications = 1000
- memory limit = 128 MB
- `NodeFileInd=`1(save nodes on a file when the tree size exceeds the memory limit)

Parameters for initial research

- `TiLim` = 60'
- `NodeLim` = 5000000 (not bounded)
- `IntSolLim` = 1
- `MIPEmphasis` = 1

Parameters for diversification:

- `TiLim` = 30'
- `NodeLim` = 5000000 (not bounded)
- `IntSolLim` = 7
- `MIPEmphasis` = 1
- $k_{min} = 1$
- $k_{step} = 2$

Parameters for refining:

- `TiLim` = 5', 10', 20', 30' for classes 1, 2, 3 and 4 respectively
- `NodeLim` = 5000000 (not bounded)
- `MIPEmphasis` = 0 (set to 1 if the attempt with the other setting fails)
- `CutUp` = cost of best solution at hand

Parameters for tight refining:

- `TiLim` = 3'

- `NodeLim` = 5000000 (not bounded)
- `MIPEmphasis` = 0
- `EpAGap` = 10^{-9}
- `EpGap` = 10^{-9}
- `CutUp` = cost of current solution
- $k'_{min} = 1$
- $k'_{max} = 4$

The tests were performed on a PC with Pentium II cpu at 450 MHz with 128 MB ram and Microsoft Windows 98. The whole set of parameters is read from a text file which must be placed on the same folder of the executable. The user can easy modify every parameter changing this file. As we can see the DRT method (developed for UMTS network planning) is always the best one except in example C where however the difference is very small (see the percentage table **??**). Due to its general nature we are encouraged from these results to test the generalized version with the other instances.

(BTS,CSS,LE)	**DRT**	**VNS**	**Cplex**	**TS**
A(50,4,2)	**10760529**	**10760529**	**10760529**	**10760529**
B(55,5,3)	**11619817**	**11619817**	**11619817**	**11619817**
C(60,5,4)	12398907	**12393173**	12396436	12395078
D(65,6,4)	**13376659**	**13376659**	**13376659**	**13445309**
E(100,8,4)	**19734790**	19740675	**19734790**	20006714
F(105,9,5)	**21950121**	21951866	21952231	22151922
G(110,9,6)	**22971252**	22973105	22971543	23100774
H(115,10,6)	**23977171**	23979111	23996482	23999151
I(150,12,6)	**30088388**	30137934	30142363	30139211
L(155,13,7)	**31096388**	31137306	31098048	31152784
M(160,13,8)	**31997972**	32035223	32045632	32070411
N(165,14,8)	**32833226**	32856421	32866855	32896514
O(200,20,10)	**39182672**	39247134	39282420	39241845

Table 5.2: Comparisons between MIP's algorithms.

	VNS		**Cplex**		**TS**	
problem	δ	δ %	δ	δ %	δ	δ %
A	0	0.00	0	0.00	0	0.00
B	0	0.00	0	0.00	0	0.00
C	5734	0.05	2471	0.02	3829	0.03
D	0	0.00	0	0.00	-68650	-0.51
E	-5885	-0.03	0	0.00	-271924	-1.38
F	-1745	-0.01	-2110	-0.01	-201801	-0.92
G	-1853	-0.01	-291	0.00	-129522	-0.56
H	-1940	-0.01	-19311	-0.08	-21980	-0.09
I	-49546	-0.16	-53975	-0.18	-50823	-0.17
L	-40918	-0.13	-1660	-0.01	-56396	-0.18
M	-37251	-0.12	-47660	-0.15	-72439	-0.23
N	-23195	-0.07	-33629	-0.10	-63288	-0.19
O	-64462	-0.16	-99748	-0.25	-59173	-0.15

Table 5.3: comparisons of DRT with VNS, Cplex 7.0 and TS. We define $\delta := DRT\_obj - algo\_obj$ and $\delta(\%) := \delta \cdot 100/DRT\_obj$.

5.3 Comparing the splitting algorithms

We have tested the two types of algorithms described in section 4.2. Table **??** and **??** provide, for each instance, the variables at first level found with the *forced variables* method. Note that results are always the ones we expect. Computation times are not relevant since this algorithm always takes from some cents to a few seconds (at worst a couple of minutes).

INSTANCE	L_1	$\|L_1\|$	**RESULT**
cap41	$y_i\ \forall i = 1, \ldots, 16$	16	OK
cap81	$y_i\ \forall i = 1, \ldots, 25$	25	OK
cap111	$y_i\ \forall i = 1, \ldots, 50$	50	OK
A	$y_i^{CSS-h}\ h = 1, 2\ i = 0, \ldots, 3$ $y_k^{LE}\ k = 0, \ldots, 1$	10	OK
B	$y_i^{CSS-h}\ h = 1, 2\ i = 0, \ldots, 4$ $y_k^{LE}\ k = 0, \ldots, 2$	13	OK
C	$y_i^{CSS-h}\ h = 1, 2\ i = 0, \ldots, 3$ $y_k^{LE}\ k = 0, \ldots, 3$	12	OK
D	$y_i^{CSS-h}\ h = 1, 2\ i = 0, \ldots, 4$ $y_k^{LE}\ k = 0, \ldots, 3$	14	OK
E	$y_i^{CSS-h}\ h = 1, 2\ i = 0, \ldots, 7$ $y_k^{LE}\ k = 0, \ldots, 3$	20	OK
F	$y_i^{CSS-h}\ h = 1, 2\ i = 0, \ldots, 8$ $y_k^{LE}\ k = 0, \ldots, 4$	23	OK
G	$y_i^{CSS-h}\ h = 1, 2\ i = 0, \ldots, 8$ $y_k^{LE}\ k = 0, \ldots, 5$	24	OK
H	$y_i^{CSS-h}\ h = 1, 2\ i = 0, \ldots, 9$ $y_k^{LE}\ k = 0, \ldots, 5$	26	OK
I	$y_i^{CSS-h}\ h = 1, 2\ i = 0, \ldots, 11$ $y_k^{LE}\ k = 0, \ldots, 5$	30	OK
L	$y_i^{CSS-h}\ h = 1, 2\ i = 0, \ldots, 12$ $y_k^{LE}\ k = 0, \ldots, 6$	33	OK
M	$y_i^{CSS-h}\ h = 1, 2\ i = 0, \ldots, 12$ $y_k^{LE}\ k = 0, \ldots, 7$	34	OK
N	$y_i^{CSS-h}\ h = 1, 2\ i = 0, \ldots, 13$ $y_k^{LE}\ k = 0, \ldots, 7$	36	OK
O	$y_i^{CSS-h}\ h = 1, 2\ i = 0, \ldots, 19$ $y_k^{LE}\ k = 0, \ldots, 9$	50	OK

Table 5.4: Results of the splitting based on the forced variables method.

INSTANCE	L_1	$\|L_1\|$	**RESULT**
ft17101div2	$y_i\ \forall i = 1, \ldots, 17$	17	OK
ft17101div3	$y_i\ \forall i = 1, \ldots, 17$	17	OK
ft17101div4	$y_i\ \forall i = 1, \ldots, 17$	17	OK
ft17101div5	$y_i\ \forall i = 1, \ldots, 17$	17	OK
ft17101div6	$y_i\ \forall i = 1, \ldots, 17$	17	OK
ft17151div2	$y_i\ \forall i = 1, \ldots, 17$	17	OK
ft17151div3	$y_i\ \forall i = 1, \ldots, 17$	17	OK
ft17151div4	$y_i\ \forall i = 1, \ldots, 17$	17	OK
ft17151div5	$y_i\ \forall i = 1, \ldots, 17$	17	OK
ft17151div6	$y_i\ \forall i = 1, \ldots, 17$	17	OK
ft17152div3	$y_i\ \forall i = 1, \ldots, 17$	17	OK
ft17201	$y_i\ \forall i = 1, \ldots, 17$	17	OK
ft17201div2	$y_i\ \forall i = 1, \ldots, 17$	17	OK
ft17201div3	$y_i\ \forall i = 1, \ldots, 17$	17	OK
ft17201div4	$y_i\ \forall i = 1, \ldots, 17$	17	OK
ft17202div3	$y_i\ \forall i = 1, \ldots, 17$	17	OK

Table 5.5: Results of the splitting based on the forced variables method.

For the FLP we catch the fist level variables from the constraint

$$x_{ij} \leq y_j\ \forall i \in N, j \in M \tag{5.1}$$

stating that a terminal can be assigned to a site only if there is an activated concentrator. The variation of this constraint for the UMTS planning model regards both LE's and CSS's since they all play the role of concentrators. So we have:

$$\sum_{i=1}^{n} T_i^{BTS} x_{ij}^{BTS-CSS} \leq \sum_{h=1,2} T_j^{CSS-h} y_j^{CSS-h}\ \forall j = 1, \ldots, m \tag{5.2}$$

or

$$\sum_{i=1}^{n} x_{ij}^{BTS-CSS} \leq \sum_{h=1,2} N_j^{CSS-h} y_j^{CSS-h}\ \forall j = 1, \ldots, m \tag{5.3}$$

or

$$\sum_{i=1}^{n} e_i^{BTS} x_{ij}^{BTS-CSS} \leq \sum_{h=1,2} E_j^{CSS-h} y_j^{CSS-h}\ \forall j = 1, \ldots, m \tag{5.4}$$

as far as the y_j^{CSS-h} concerns and

$$\sum_{j=1}^{m} w_{jk}^{CSS-LE} + \sum_{i=1}^{n} T_i^{BTS} x_{ik}^{BTS-LE} \leq T_k^{LE} y_k^{LE}\ \forall k = 1, \ldots, p \tag{5.5}$$

or

$$\sum_{j=1}^{m} z_{jk}^{CSS-LE} + \sum_{i=1}^{n} e_i^{BTS} x_{ik}^{BTS-LE} \leq E_k^{LE} y_k^{LE} \; \forall k = 1, \ldots, p \tag{5.6}$$

for y_k^{LE}.

The QCL-C counterpart is:

$$\sum_{i \in I} n_i x_{ij} + \sum_{l \in I \setminus \{j\}} (z_{lj} + z_{jl}) \leq M y_j \quad \forall j \in I \tag{5.7}$$

where n_i and z_{jl} are auxiliary variables on traffic demands used to linearize the model.

The second method based on *BIG M coefficients* leads to results very variable with the choice of percentage used to state if a coefficient is big M or not.

INSTANCE	L_1^*	$\lvert L_1 \rvert$	**RESULT**
A	$x_{i,j}^{CSS-LE}\ i = 0, \ldots, 3\ j = 0, \ldots, 2$	32	EXCEED
B	$x_{i,j}^{CSS-LE}\ i = 0, \ldots, 4\ j = 0, \ldots, 2$	28	EXCEED
C	$x_{i,j}^{CSS-LE}\ i = 0, \ldots, 4\ j = 0, \ldots, 3$ $x_{34,k}^{BTS-CSS}\ k = 0, \ldots, 4$	37	EXCEED
D	$x_{i,j}^{CSS-LE}\ i = 0, \ldots, 5\ j = 0, \ldots, 3$	38	EXCEED
E	$x_{i,j}^{CSS-LE}\ i = 0, \ldots, 7\ j = 0, \ldots, 3$	52	EXCEED
F	$x_{i,j}^{CSS-LE}\ i = 0, \ldots, 8\ j = 0, \ldots, 4$	68	EXCEED
G	$x_{i,j}^{CSS-LE}\ i = 0, \ldots, 8\ j = 0, \ldots, 5$	78	EXCEED
H	$x_{i,j}^{CSS-LE}\ i = 0, \ldots, 9\ j = 0, \ldots, 5$	86	EXCEED
I	$x_{i,j}^{CSS-LE}\ i = 0, \ldots, 11\ j = 0, \ldots, 5$	102	EXCEED
L	$x_{i,j}^{CSS-LE}\ i = 0, \ldots, 12\ j = 0, \ldots, 6$	124	EXCEED
M	$x_{i,j}^{CSS-LE}\ i = 0, \ldots, 12\ j = 0, \ldots, 7$	138	EXCEED
N	$x_{i,j}^{CSS-LE}\ i = 0, \ldots, 13\ j = 0, \ldots, 7$	148	EXCEED
O	$x_{i,j}^{CSS-LE}\ i = 0, \ldots, 19\ j = 0, \ldots, 9$	250	EXCEED

Table 5.6: results of the splitting based on the big M method with $bigMperc = 0.9$.

: the L_1 set is made up by all variables reported in table* **??. *Here, for practical reasons, we report just the new variables added by this method*

INSTANCE	L_1	$\|L_1\|$	**RESULT**
ft17101div2	$x_{i,j}\ \forall i,j = 1,\ldots,17$	289	EXCEED
ft17101div3	$x_{i,j}\ \forall i,j = 1,\ldots,17$	289	EXCEED
ft17101div4	$x_{i,j}\ \forall i,j = 1,\ldots,17$	289	EXCEED
ft17101div5	$x_{i,j}\ \forall i,j = 1,\ldots,17$	289	EXCEED
ft17101div6	$x_{i,j}\ \forall i,j = 1,\ldots,17$	289	EXCEED
ft17151div2	$x_{i,j}\ \forall i,j = 1,\ldots,17$	289	EXCEED
ft17151div3	$x_{i,j}\ \forall i,j = 1,\ldots,17$	289	EXCEED
ft17151div4	$x_{i,j}\ \forall i,j = 1,\ldots,17$	289	EXCEED
ft17151div5	$x_{i,j}\ \forall i,j = 1,\ldots,17$	289	EXCEED
ft17151div6	$x_{i,j}\ \forall i,j = 1,\ldots,17$	289	EXCEED
ft17152div3	$x_{i,j}\ \forall i,j = 1,\ldots,17$	289	EXCEED
ft17201	$x_{i,j}\ \forall i,j = 1,\ldots,17$	289	EXCEED
ft17201div2	$x_{i,j}\ \forall i,j = 1,\ldots,17$	289	EXCEED
ft17201div3	$x_{i,j}\ \forall i,j = 1,\ldots,17$	289	EXCEED
ft17201div4	$x_{i,j}\ \forall i,j = 1,\ldots,17$	289	EXCEED
ft17202div3	$x_{i,j}\ \forall i,j = 1,\ldots,17$	289	EXCEED

Table 5.7: Results of the splitting based on the big M method with $bigMperc = 0.9$. Recall that $x_{i,i}$ is equivalent to y_i.

For Polo's instances (from A to O) y_j^{CSS-h}'s are derived from

$$\sum_{i=1}^{n} T_i^{BTS} x_{ij}^{BTS-CSS} \leq \sum_{h=1,2} T_j^{CSS-h} y_j^{CSS-h} \tag{5.8}$$

or

$$\sum_{i=1}^{n} x_{ij}^{BTS-CSS} \leq \sum_{h=1,2} N_j^{CSS-h} y_j^{CSS-h} \tag{5.9}$$

or

$$\sum_{i=1}^{n} e_i^{BTS} x_{ij}^{BTS-CSS} \leq \sum_{h=1,2} E_j^{CSS-h} y_j^{CSS-h} \tag{5.10}$$

where $T_i^{BTS} \sim 10$ while $T_j^{CSS-h} \sim 100$ and so $T_i^{BTS} \ll T_j^{CSS-h}$. On the other hand, we catch y_k^{LE}'s from

$$\sum_{j=1}^{m} w_{jk}^{CSS-LE} + \sum_{i=1}^{n} T_i^{BTS} x_{ik}^{BTS-LE} \leq T_k^{LE} y_k^{LE} \tag{5.11}$$

or

$$\sum_{j=1}^{m} z_{jk}^{CSS-LE} + \sum_{i=1}^{n} e_i^{BTS} x_{ik}^{BTS-LE} \leq E_k^{LE} y_k^{LE} \tag{5.12}$$

again,$T_i^{BTS} \ll T_k^{LE}$ ($T_k^{LE} \sim 1000$, $T_i^{BTS} \sim 10$) and $e_i^{BTS} \ll E_k^{LE}$. Note that we don't perform any test about nature of all variables inside a constraint like we did in the previous method, since there is no logical relationship between variables. We just consider the size of their coefficients. Wrong variable splitting can be obtained as we can see in table **??** from constraints like

$$z_{jk}^{CSS-LE} \leq M_{jk} x_{jk}^{CSS-LE} \tag{5.13}$$

or

$$w_{jk}^{CSS-LE} \leq F_{jk} x_{jk}^{CSS-LE} \tag{5.14}$$

because coefficients are on the order of 10^9. Changing $bigMperc$ from 0.9 to 2.0 cuts them away.

For class 2 problems (Yaman and Labbé's) the method leads to no significant results since we don't have great difference between coefficients and so all binary variables are put at 1st level, even the ones regarding links of access network. Even if we increase the $bigMperc$ value we don't get an improvement.

Finally, it is not surprising that both methods lead to correct results for class 1 problem. In fact, the forced variable constraint is at the base of FLP:

$$x_{ij} \leq y_j \tag{5.15}$$

and in case of capacity limitation usually this value is very high (some hundreds) while all other coefficients are 1 or 0.

5.4 Comparing the algorithms

First of all, we will compare GLPK with ILOG-Cplex 7.0. There aren't particular settings for the GLPK, except the time limit which is, as always, 3 hours (10800 seconds). For Cplex, we also set MIPEmphasis to 1, to force the solver to generate as many feasible solutions as possible. The differences are expressed as:

- $\triangle := GLPK\_time - CPLEX\_time$
- $\delta := GLPK\_obj - CPLEX\_obj$
- $\delta\% := \frac{\delta}{GLPK\_obj} \cdot 100$

FILE	**GLPK obj**	**GLPK time (s)**	$\triangle$	δ	δ (%)
ft17101div2	26614.6	10800	0	9.5	0.04
ft17101div3	19484.8	8051	-2749	149.1	0.77
ft17101div4	15654.5	8616	-2185	133.7	0.85
ft17151div2	23601.9	10800	0	-701.5	-2.97
ft17151div3	17580.3	4572	-6228	266.8	1.52
ft17151div4	14899.4	10800	5187	0.0	0.00
ft17151div6	12837.7	971	722	0.0	0.00
ft17152div3	32279.4	10800	0	734.8	2.28
ft17201div2	20480.7	10800	0	-1195.4	-5.84
ft17201div3	16662	10800	1377	245.5	1.47

Table 5.8: GLPK solver performances on a instances subset.

From a strictly quality and speed solution point of view, table 5.4.1 shows that the GNU solver is quite good, approaching Cplex. However, the main blind spot is the reliability. In fact, in some cases the model parser module doesn't manage to read the file properly or the numerical instability cause the solver module to fail. Many errors have occurred with instances of class 2. These are the main reasons why we will provide in the following sections results regarding the DRT implementation based on the ILOG Cplex.

Now, we pass to the test of the main methods. The reference value is given by the basic DRT implementation. We compare it with its variation, the DRT2, with a pure DRT version without first level variables (so, in this case we don't use the scheme seen in figure **??**), with the Local Branching and Cplex 7.0. The main settings follow:

Cplex 7.0 settings:

- time limit: 3 hours
- MIPEmphasis = 1 (emphasis on feasibility)

DRT settings: the DRT has been tested using basically the same parameters seen in section 5.2 but we have changed:

- time limit: 3 hours
- maximum number of diversifications: 5

In this way we give priority to the search with fixed configurations of first level variables since we use very long time for each refining or tight refining phase.

DRT2 settings:

- time limit: 3 hours
- maximum number of big diversifications: 4
- maximum number of diversifications without improvement: 5
- minimum gap of improvement: 5%
- time limit for big and small diversification: 10 min.
- time limit for refining: 5 min.
- time limit for tight refining: 3 min.

Local Branching settings:

- time limit: 3 hours
- time for neighborhood: 600 seconds
- emphasis: 1 (feasibility)
- k = 20
- heuristic yes (no for LocBra2[3])

[3]if heuristic is set to no algorithm enters in exact mode and doesn't perform diversifications.

problem	DRT	Cplex	LocBra	LocBra2	DRT2	DRT($L_1 = \emptyset$)
A	10760529	10760984	10760529	10760529	10760529	10761424
B	11595231	11619817	11595038	11595038	11595231	11599598
C	12393173	12393173	12402144	12396570	12393173	12397113
D	13376204	13378018	13394693	13378152	13376204	13380362
E	19731006	19731735	19738110	19731497	19737006	19738132
F	21950121	21950121	21964673	21950775	21950121	21950121
G	22969378	22973372	23081822	22972861	22969666	22973771
H	23978221	23990620	24019075	23993010	23978331	24032049
I	30093168	30138793	30120059	30164210	30089580	30197225
L	31081092	31094535	32506306	31118151	31081804	31167321
M	31986285	32065697	32097378	32078994	31984753	32106332
N	32842527	32894205	34310477	32979278	32834623	32952467
O	39181447	39282420	40696120	39478615	39183489	39457778
ft17101div2	26640.0	26605.1	26289.2	26274.5	26447.6	26289.2
ft17101div3	19702.3	19335.7	18726.6	19973.4	19582.1	19147.4
ft17101div4	15520.8	15520.8	15520.8	15520.8	15520.8	15520.8
ft17101div5	13902.2	13902.2	13902.2	13902.2	13902.2	13902.2
ft17101div6	13035.7	13035.7	13035.7	13035.7	13035.7	13035.7
ft17151div2	25303.4	24303.4	24201.4	27323.6	25626.0	24724.8
ft17151div3	17313.5	17313.5	17465.0	17465.0	17313.5	17921.8
ft17151div4	14899.4	14899.4	14899.4	14899.4	14899.4	15520.8
ft17151div5	13690.7	13690.7	13690.7	13690.7	13690.7	13902.2
ft17151div6	12837.7	12837.7	12837.7	12837.7	12837.7	13035.7
ft17152div3	32042.7	31544.6	30671.0	31786.2	32827.6	32089.8
ft17201div2	21213.0	21676.1	22471.8	23765.9	22232.8	23219.2
ft17201div3	16416.5	16416.5	16538.4	16516.0	16416.5	16885.0
ft17201div4	14646.0	14566.5	14566.5	14566.5	14566.5	14646.0
ft17201div5	13661.8	13661.8	13661.8	13661.8	13661.8	13661.8
ft17201div6	12837.7	12837.7	12837.7	12837.7	12837.7	12837.7
ft17201	43317.6	43361.1	43241.2	43107.5	45153.7	43444.8
ft17202div3	27950.2	29491.3	29659.6	30668.0	28609.9	29519.0

Table 5.9: Objective function values for all considered algorithms.

Now, table **??** reports a comparison between DRT and other methods, recalling that

- $\delta := DRT\_obj - algo\_obj$
- $\delta(\%) := \frac{\delta \cdot 100}{DRT\_obj}$

so, a negative value states that DRT is better than the current reference algorithm.

	Cplex		**LocBra**		**LocBra2**		**DRT2**		**DRT**($L_1 = \emptyset$)	
problem	δ	δ %	δ	δ %	δ	δ %	δ	δ %	δ	δ %
A	-455	0.00	0	0.00	0	0.00	0	0.00	-895	-0.01
B	-24586	-0.21	193	0.00	193	0.00	0	0.00	-4367	-0.04
C	0	0.00	-8971	-0.07	-3397	-0.03	0	0.00	-3940	-0.03
D	-1814	-0.01	-18489	-0.14	-1948	-0.01	0	0.00	-4158	-0.03
E	-729	0.00	-7104	-0.04	-491	0.00	6000	-0.03	-7126	-0.04
F	0	0.00	-14552	-0.07	-654	0.00	0	0.00	0	0.00
G	-3994	-0.02	-112444	-0.49	-3483	-0.02	-288	0.00	-4393	-0.02
H	-12399	-0.05	-40854	-0.17	-14789	-0.06	-110	0.00	-53828	-0.22
I	-45625	-0.15	-26891	-0.09	-71042	-0.24	3588	0.01	-104057	-0.35
L	-13443	-0.04	-1425214	-4.59	-37059	-0.12	-712	0.00	-86229	-0.28
M	-79412	-0.25	-111093	-0.35	-92709	-0.29	1532	0.00	-120047	-0.38
N	-51678	-0.16	-1467950	-4.47	-136751	-0.42	7904	0.02	-109940	-0.33
O	-100973	-0.26	-1514673	-3.87	-297168	-0.76	-2042	-0.01	-276331	-0.71
ft17101div2	34.9	0.13	350.8	1.32	365.5	1.37	192.4	0.72	351.0	1.32
ft17101div3	366.6	1.86	975.7	4.95	-271.1	-1.38	120.2	0.61	555.0	2.82
ft17101div4	0.0	0.00	0.0	0.00	0.0	0.00	0.8	0.01	0.0	0.00
ft17101div5	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00
ft17101div6	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00
ft17151div2	1000.0	3.95	1102.0	4.35	-2020.2	-7.98	-322.6	-1.28	579.0	2.29
ft17151div3	0.0	0.00	-151.4	-0.87	-151.4	-0.87	0.0	0.00	-608.0	-3.51
ft17151div4	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	-621.0	-4.17
ft17151div5	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	-212.0	-1.54
ft17151div6	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	-198.0	-1.54
ft17152div3	498.1	1.55	1371.6	4.28	256.4	0.80	-785.0	-2.45	-47.0	-0.15
ft17201div2	-463.1	-2.18	-1258.8	-5.93	-2552.9	-12.03	-1019.8	-4.81	-2006.0	-9.46
ft17201div3	0.0	0.00	-121.9	-0.74	-99.4	-0.61	0.0	0.00	-468.0	-2.85
ft17201div4	79.5	0.54	79.5	0.54	79.5	0.54	79.5	0.54	0.0	0.00
ft17201div5	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00
ft17201div6	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00
ft17201	-43.5	-0.10	76.4	0.18	210.1	0.49	-1836.1	-4.24	-127.0	-0.29
ft17202div3	-1541.1	-5.51	-1709.4	-6.12	-2717.8	-9.72	-659.7	-2.36	-1569.0	-5.61

Table 5.10: Difference between DRT and other methods.

Figure 5.1: graphical representation of δ %. The upper plot concerns instances A to O. The lower is about instances ft.

Figure 5.2: graphical comparison of algorithms. *better* means that the current algorithm is better than DRT.

Figure **??** shows a graphical representation of the $\delta(\%)$'s, while the pie charts of figure **??** provide a view of the percentage of cases in which DRT is better than the other methods. It is clear that in most cases the DRT is better than or equal to the compared algorithm, even if the differences are not so dramatic. The DRT2 doesn't give a very significant improvement with respect to the basic DRT as we could expect, since every time we perform a big diversification the algorithm needs some complete diversification-refining-tight-refining cycles to reach a solution that is qualitatively comparable with the one left before the big diversification. If we don't give to the DRT, the results are very bad: in this case the DRT method turns to a 2-phases algorithm that is not a VNS-like. For the Local Branching, we can see that the algorithm is more efficient when we use the *heuristic* option. It is also quite clear that the DRT behaves better with instances provided by Polo, while the trend for Yaman's instances is very variable with the problem at hand.

5.4.1 Two detailed examples

Finally, in figures **??** and **??** we show a graphical comparison between the three main heuristic methods (DRT, DRT2 and Local Branching) solving two problems: L and ft17201div2. Note that every time tight-refining or refining are performed giving no new solution we don't write them even if they use some computational time. DRT and DRT2 present some dramatic excursions from a cycle to cycle, especially in case of big diversification. Moreover, here we can concretely see what we said before: after each big diversification the DRT2 needs some complete cycles to find a solution which is comparable with the one of the previous big diversification. So, the improvement is not so high as we could expect. Cplex and DRT with no first level variables have a monotonically decreasing behavior. For the second one, we can explain this phenomenon considering that now we perform only two phases, with no diversification. Every refining (and tight-refining) is done improving the previous solution and so we do not have excursions as we can see in true heuristic algorithms. A very similar behavior can be seen for Local Branching with exact option: as soon as the algorithm have to perform the first diversification the heuristic procedure is left and an exact procedure is called until the end of the given time. Otherwise, in the (true) Local Branching plots we can clearly see the peaks which represent a diversification.

Figure 5.3: Plots of the methods solving problem L.

Figure 5.4: Plots of the methods solving problem ft17201div2.

Chapter 6

Conclusions

In this work we have discussed a generic framework to solve large mixed integer problems in an heuristic way. We have provided a totally automated procedure which has the task to automatically split the binary variables set individuating the variables to be fixed. The way this decision is taken has been deeply discussed as well. After that, the algorithm evolves investigating different subspaces basing its way on the user's settings. We have seen two ways of controlling this evolution; the second one is based on the concept of past memory introduced by the Tabu Search method. The whole procedure has been implemented in C++ with Cplex 7.0 and in C with the GLPK and provided as a software library. The algorithm has been tested with good results on different instances mainly based on telecommunications network planning. All these results are compared with the reference commercial solver Cplex 7.0 and discussed. A comparison with the GNU solver GLPK and with the local branching heuristic method are given too. The global results let us state that our method is very efficient, nearly always better than the concurrent.

We have seen how much the settings can drive the performances of our algorithm. An interesting future development could be try to find out a method of auto-tuning for at least the main important ones. Other ways of controlling the algorithm from "inside", such as the variation of the basic method provided, can be developed as well. The application of the method to other classes of problems could lead to emphasize the generality of this procedure.

Appendix A

Test models

This section describe the models of the tests used in this work. They are all related to telecommunication networks design since this problem is actually very hot and many research efforts have been done to solve in an heuristic manner problems that usually are very difficult, mainly for their very big size. The first model we will study is a very classical one and it is at the base of all network models: the uncapacitated facility location problem. Then we will face two kinds of problems encountered in planning an UMTS network, the new mobile communication standard.

A.1 UFLP

The NP-hard problem UFLP (*uncapacitated facility location problem*) is a core problem for the design of telecommunication networks so it will be discussed as introduction to the topic. Given a set N of terminals and a set M of possible locations for the service providers (also called concentrators), we have to determinate the number and the location of these providers and the way to assign them to the terminals. The goal is to minimize the cost for installing the concentrators and for the terminals requests satisfaction. The model is:

$$minimize \sum_{i \in N} \sum_{j \in M} c_{ij} x_{ij} + \sum_{j \in M} F_j y_j$$

subject to

$$\sum_{j \in M} x_{ij} = 1 \quad \forall i \in N \tag{A.1}$$

$$x_{ij} \leq y_j \quad \forall i \in N, j \in M \tag{A.2}$$

$$x_{ij} \in \{0, 1\} \quad \forall i \in N, j \in M \tag{A.3}$$

$$y_j \in \{0,1\} \quad \forall j \in M \tag{A.4}$$

$$y_j = \begin{cases} 1 & \text{if a concentrator is installed at location } j \\ 0 & \text{otherwise} \end{cases}$$

for all $j \in M$

$$x_{ij} = \begin{cases} 1 & \text{if terminal } i \text{ is assigned to a concentrator installed at location } j \\ 0 & \text{otherwise} \end{cases}$$

for all $i \in N$, $j \in M$. c_{ij} is the cost to connect terminal i to provider j, F_j is the fixed cost to install a provider at the j-th site. Constraints **??** and **??** state that each terminal can be assigned to exactly one concentrator and constraints **??** state that terminals can be assigned only to locations where a concentrator have been installed. We could say that the y's control the x's. This is a very relevant observation which will be exploited by the DRT method. Variations of this model are the capacitated one (CFLP), the p-median problem where we fix the number of y's, and so on. Beasley provides an extensive collection of instances of these problems in his OR library [**?**].

A.2 QCL-C

The QCL-C (*Quadratic Capacity Concentrator Location Problem with Complete Routing*) is a problem very close to the CFLP and so it's easy to show that this is a NP-hard problem too. The goal is to build in an optimal way the interconnection network between concentrators called *backbone network* and the one between terminals and concentrators (the *access network*).

For our model we refer to the one suggested by Ernst and Krishnamoorthy called QCL-C3 (see [**?**]), with $O(n^3)$ variables and $O(n^2)$ constraints (n is the number of terminals). Let I denote the set of terminals, with $|I| = n$ and K the set of sites where concentrators can be installed $|K| = m$. Figure **??** illustrates a possible configuration.

We define

$$A := \{(j,l) : j,l \in I, j \neq l\}$$

$$x_{ij} = \begin{cases} 1 & \text{if terminal } i \text{ is assigned to a concentrator installed at location } j \\ 0 & \text{otherwise} \end{cases}$$

for all $i,j \in I$. x_{ii} stands for concentrators, that is $x_{ii} = 1$ if a concentrator is installed at location i. z_{jl} is the total amount of traffic on the arc (j,l) for all $(j,l) \in A$. B_{jl} is the cost of routing a unit of traffic on the arc (j,l) if it belongs to the backbone network, that is if both j and l are concentrators. c_{ij} are the

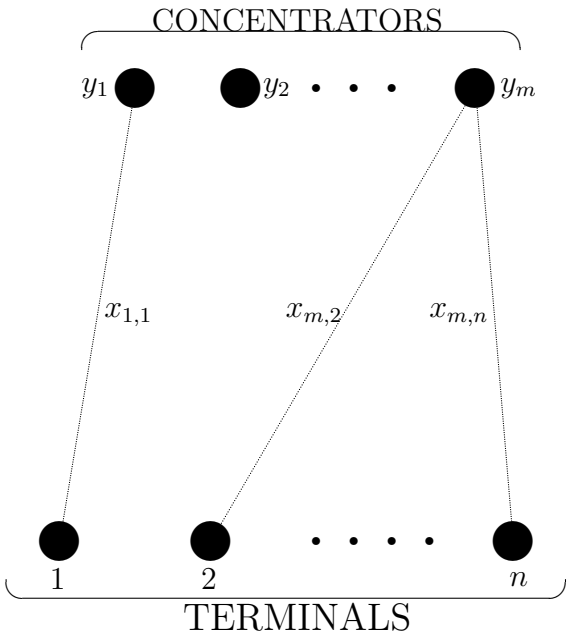

Figure A.1: connections between concentrators and n terminals

cost to connect node i to node j. M is the capacity for each concentrator and represents the upper bound for the total amount of traffic passing through the concentrator. f_{jl}^{i} is the flow of commodities originating at node i and travelling along the arc (j, l), for all $i \in I$ and $(j, l) \in A$. t_{ij} is the amount of traffic between node i and node j and define

$$n_i = \sum_{m \in I} (t_{mi} + t_{im})$$

So the model is:

$$minimize \sum_{i \in I} \sum_{j \in I} c_{ij} x_{ij} + \sum_{(j,l) \in A} B_{jl} z_{jl}$$

subject to

$$\sum_{j \in I} x_{ij} = 1 \quad \forall i \in I \tag{A.5}$$

$$\sum_{l \in I} f_{jl}^{i} - \sum_{l \in I} f_{lj}^{i} = \sum_{m \in I} t_{im} (x_{ij} - x_{mj}) \quad \forall i, j \in I \tag{A.6}$$

$$z_{jl} \geq \sum_{i \in I} f_{jl}^{i} \quad \forall (j,l) \in A \tag{A.7}$$

$$\sum_{i \in I} n_i x_{ij} + \sum_{l \in I \setminus \{j\}} (z_{lj} + z_{jl}) \leq M x_{jj} \quad \forall j \in I \tag{A.8}$$

$$x_{ij} \in \{0,1\} \quad \forall i,j \in I \tag{A.9}$$

$$z_{ij} \geq 0 \quad \forall (j,l) \in A \tag{A.10}$$

$$f_{jl}^{i} \geq 0 \quad \forall (j,l) \in A, i \in I \tag{A.11}$$

A.3 UMTS network planning

The UMTS (standing for *Universal Mobile Telecommunication System*) planning is a very hot argument at this time. UMTS is the third generation European standard for mobile telecommunications systems. Since the scope of this standard is to provide to users an advanced service letting them to exchange large amount of data in an efficient way, there are several technical constraints to be fulfilled. There are several aspects we can face in a network planning: we are going to consider its construction with respect to the nodes activation looking at the capacities, the traffic amount and the type of each one. A model based on the signal power control is discussed deeply in [**?**].

A mobile radio telephone system aims at ensuring secure communications between mobile terminals and any other type of user device, either mobile or fixed. A mobile customer should be reachable at any time and in any location where the radio coverage is granted.

The connection among mobile terminals (i.e., the user's handheld terminals) and fixed radio base stations is obtained by means of radio waves. However, a single antenna system cannot cover the whole service area. In fact, that choice would require high irradiation power both from the fixed and the mobile stations, with consequent possible damage due to the generated electromagnetic field.

The above limitations lead to the implementation of "cellular systems", constituted by several fixed radio base stations and related antennas systems. Each single radio base station coverage area is called "cell" and it serves a small region of variable size ranging from 10-100 meters (high user density inside business buildings) to 1-20 kilometers (low user density areas in the country).

Every fixed radio base station, usually called Base Transceiver Station (BTS), is both transmitting and receiving signals on a variable number of frequencies. Depending upon the type of system considered and the radio access scheme, each

frequency (or carrier) permits the allocation of a variable number of channels; in the GSM case, each frequency carries 8 channels.

Whenever a user moves from a cell to an adjacent one during a communication, a new channel is assigned inside the cell just entered. This feature is commonly called handover. Covering the served region with several cells allows for "frequency reuse", i.e., for the use of the same frequency inside two or more non- interfering cells.

The users' mobility causes issues related to the user location detection and to cell change, which are managed by equipment implementing the interface between the BTS and the fixed network.

Third generation mobile telecommunication systems are currently in course of standardization in Europe under the name of Universal Mobile Telecommunication System (*UMTS*). The basic architecture of a UMTS network include the following devices:

- Mobile Terminal (*MT*) of different types (e.g., phone, fax, video, computer).
- Base Transceiver Station (*BTS*) interfacing mobile users to the fixed network; a BTS handles users' access and channel assignment. Due to the inherent flexibility featured by next generation BTS's, different network topologies can be undertaken: the BTS can be either directly connected to the switching equipment (smart BTS) or linked to a BTS controller (CSS).
- Cell Site Switch (*CSS*), which is a switch connected to several BTS's on one side and to a single Local Exchange (see below) on the other side; each CSS is devoted to the management of local traffic inside its controlled area, as well as to the connection of the controlled BTS's to the Local Exchange.
- Local Exchange (*LE*), which is a switch connecting the BTS's to the network, either directly or through CSS's.
- Mobility and Service Data Point (*MSDP*), which is a data base where information about users is registered; it may be located either together with an LE or with a CSS, according to a centralized or distributed connection management.
- Mobility and Service Control Point (*MSCP*), which is a controller to manage mobility; it can access the data base to read, write or erase information about users, and is generally associated with LE's and MSDP's.

BTS's are also called *terminals* while CSS's and LE's are called *concentrators*. The network with BTS-CSS or BTS-LE links is known as *access network* and the one with CSS-LE arcs is the *backbone network*. They both can be star or complete network leading to different configurations with different associated problems. The design of a UMTS network can be divide into three phases:

1. decide about number and locations of concentrators and the assignment of terminals to these concentrators

2. design access network (terminals-concentrators)

3. design backbone network (concentrators-concentrators)

We now want to apply the DRT method to the project of a UMTS interconnection network having a multilevel star-type architecture (both access and backbone network are star type). This is a difficult-to-solve (NP-hard) optimization problem of crucial importance in the design of effective and low-cost networks.

In the problem we consider, a certain number of potential CSS and LE sites is given, among which the planner has to choose those to be actually activated. We consider a three level star-type UMTS architecture, defined by an upper layer made up of active LE's (chosen in the given set of potential LE's), a middle layer made up of active CSS's (also chosen in the given set of potential CSS's), and a lower layer made up of the given BTS's (each of which is required to play the role of a leaf in the star-type structure). The problem then consists of choosing the CSS and LE to be activated, and the way to connect them to the BTS's and between each other, so as to produce a feasible three-level network of minimum cost. Figure **??** depicts an example of star/star UMTS network with six LE's, six CSS's and seventeen BTS's. We have to choose what devices have to be activated (gray ones) and the way they are interconnected.

Figure A.2: the star/star structure of UMTS network with a possible configuration for the given devices

The variables we will use are the following:

$$y_j^{CSS-h} = \begin{cases} 1 & \text{if a CSS of } h \text{ type is installed in location } j \text{ assigned to} \\ & \text{a concentrator installed at location } j \\ 0 & \text{otherwise} \end{cases}$$

$$y_k^{LE} = \begin{cases} 1 & \text{if a LE is installed in location } k \\ 0 & \text{otherwise} \end{cases}$$

$$x_{ij}^{BTS-CSS} = \begin{cases} 1 & \text{if a BTS in location } i \text{ isassigned to} \\ & \text{a CSS installed at location } j \\ 0 & \text{otherwise} \end{cases}$$

$$x_{ik}^{BTS-LE} = \begin{cases} 1 & \text{if a BTS in location } i \text{ isassigned to} \\ & \text{a LE installed at location } k \\ 0 & \text{otherwise} \end{cases}$$

$$x_{jk}^{CSS-LE} = \begin{cases} 1 & \text{if a CSS in location } j \text{ isassigned to} \\ & \text{a LE installed at location } k \\ 0 & \text{otherwise} \end{cases}$$

Additional integer non negative variables are:

z_{jk}^{CSS-LE} which is the number of modules from a CSS in location j to a LE in location k

w_{jk}^{CSS-LE} which is the traffic flow between CSS in location j and a LE in location k

The mathematical model is:

$$\min \sum_{j=1}^{m} \sum_{h=1,2} f_j^{CSS-h} y_j^{CSS-h} + \sum_{k=1}^{p} f_k^{LE} y_k^{LE} +$$

$$+ \sum_{i=1}^{n} \sum_{j=1}^{m} \left(c_{ij}^{BTS-CSS} e_i^{BTS} + f_i^{BTS-CSS} \right) x_{ij}^{BTS-CSS} +$$

$$+ \sum_{i=1}^{n} \sum_{k=1}^{p} \left(c_{ik}^{BTS-LE} e_i^{BTS} + f_i^{BTS-LE} \right) x_{ik}^{BTS-LE} + \sum_{j=1}^{m} \sum_{k=1}^{p} c_{jk}^{CSS-LE} z_{jk}^{CSS-LE}$$

subject to:

$$\sum_{j=1}^{m} x_{ij}^{BTS-CSS} + \sum_{k=1}^{p} x_{ik}^{BTS-LE} = 1 \qquad \forall i = 1, \ldots, n \qquad \text{(A.12)}$$

$$\sum_{i=1}^{n} T_i^{BTS} x_{ij}^{BTS-CSS} \leq \sum_{h=1,2} T_j^{CSS-h} y_j^{CSS-h} \qquad \forall j = 1, \ldots, m \qquad \text{(A.13)}$$

$$\sum_{j=1}^{m} x_{ij}^{BTS-CSS} \leq \sum_{h=1,2} N_j^{CSS-h} y_j^{CSS-h} \qquad \forall j = 1, \ldots, m \qquad \text{(A.14)}$$

$$\sum_{j=1}^{m} e_i^{BTS} x_{ij}^{BTS-CSS} \leq \sum_{h=1,2} E_j^{CSS-h} y_j^{CSS-h} \qquad \forall j = 1, \ldots, m \qquad \text{(A.15)}$$

$$\sum_{i=1}^{n} e_i^{BTS} x_{ij}^{BTS-CSS} \leq Q \sum_{k=1}^{p} z_{jk}^{CSS-LE} \qquad \forall j = 1, \ldots, m \qquad \text{(A.16)}$$

$$z_{jk}^{CSS-LE} \leq M_{jk} x_{jk}^{CSS-LE} \quad \forall j = 1, .., m \, ; k = 1, .., p \qquad \text{(A.17)}$$

$$\sum_{j=1}^{m} w_{jk}^{CSS-LE} + \sum_{i=1}^{n} T_i^{BTS} x_{ik}^{BTS-LE} \leq T_k^{LE} y_k^{LE} \qquad \forall j = 1, \ldots, m \qquad \text{(A.18)}$$

$$\sum_{i=1}^{n} T_i^{BTS} x_{ij}^{BTS-CSS} = \sum_{k=1}^{p} w_{jk}^{CSS-LE} \qquad \forall j = 1, \ldots, m \qquad \text{(A.19)}$$

$$w_{jk}^{CSS-LE} \leq F_{jk} x_{jk}^{CSS-LE} \quad \forall j = 1, .., m \, ; k = 1, .., p \qquad \text{(A.20)}$$

$$\sum_{j=1}^{m} z_{jk}^{CSS-LE} + \sum_{i=1}^{n} e_i^{BTS} x_{ik}^{BTS-LE} \leq E_k^{LE} y_k^{LE} \qquad \forall k = 1, \ldots, p \qquad \text{(A.21)}$$

$$\sum_{h=1,2} y_j^{CSS-h} \leq 1 \qquad \forall j = 1, \ldots, m \tag{A.22}$$

$$\sum_{k=1}^{p} x_{jk}^{CSS-LE} = \sum_{h=1,2} y_j^{CSS-h} \qquad \forall j = 1, \ldots, m \tag{A.23}$$

$$x_{ij}^{BTS-CSS} \leq \sum_{h=1,2} y_j^{CSS-h} \quad \forall i = 1, .., n;\ j = 1, .., m \tag{A.24}$$

$$x_{ik}^{BTS-LE} \leq y_k^{LE} \quad \forall i = 1, .., n; k = 1, .., p \tag{A.25}$$

$$x_{jk}^{CSS-LE} \leq y_k^{LE} \quad \forall j = 1, .., m; k = 1, .., p \tag{A.26}$$

$$\sum_{h=1,2} y_j^{CSS-h} \leq \sum_{k=1}^{p} z_{jk}^{CSS-LE} \qquad \forall j = 1, \ldots, m \tag{A.27}$$

$$\sum_{k=1}^{p} y_k^{LE} \geq 1 \tag{A.28}$$

Variables as follows:

$$\begin{aligned}
y_j^{CSS-h} &\in \{0,1\} \quad \forall j = 1, .., m; h = 1, 2 \\
y_k^{LE} &\in \{0,1\} \quad \forall k = 1, \ldots, p \\
x_{ij}^{BTS-CSS} &\in \{0,1\} \quad \forall i = 1, .., n; j = 1, .., m \\
x_{ik}^{BTS-LE} &\in \{0,1\} \quad \forall i = 1, .., n; k = 1, .., p \\
x_{ik}^{CSS-LE} &\in \{0,1\} \quad \forall i = 1, .., n; k = 1, .., p \\
z_{ik}^{CSS-LE} &\in \mathcal{N} \quad \forall i = 1, .., n; k = 1, .., p \\
w_{jk}^{CSS-LE} &\geq 0 \quad \forall j = 1, .., m; k = 1, .., p
\end{aligned}$$

Looking at the formulation it is quite clear that the most suitable variables for the first level are the ones regarding concentrators. In fact they are binary and in a very smaller number with respect to the BTS's. Also while the terminals are all activated, we can choose a subset of CSS's and LE's to be used and so we can build different access networks for each fixed backbone one.

The central expression for the DRT method is the distance definition between two solutions to build the neighborhoods.

$$\triangle'(y, y^*) := \sum_{j \in Y_{CSS}^*} \left[1 - \sum_{h=1,2} \left(y_j^{CSS-h^*} y_j^{CSS-h} \right) \right] + \tag{A.29}$$

$$+ \sum_{k \in Y_{LE}^*} \left(1 - y_k^{LE} \right) + \sum_{j \notin Y_{CSS}^*} \sum_{h=1,2} y_j^{CSS-h} + \sum_{k \notin Y_{LE}^*} y_k^{LE}$$

where the * denotes the current configuration for a certain set of devices.

Appendix B

Algorithms settings

Now, we report the settings for all heuristic algorithms. DRT and DRT2 take these parameters from a plain text file on the same format shown below. All settings for Local Branching are given as arguments of the command line.

DRT settings

```
/*****************************************************************
_cplexpar_DRT.dat - general information on Cplex parameters for
DRT method.  Setting of Cplex parameters in relation to the four
phases of DRT research (general, diversification, refining, tight ref)
*****************************************************************/


   Total time limit (hours) :  3
   Maximum number of diversifications :  1000

# Global algorithm Cplex parameters
   Tree memory limit :  128
   Node storage file indicator :  1

# Parameters for initial general research
   Time limit (minutes) :  60
   Maximum number of nodes :  5000000
   Precedence to optimality (0) or to feasibility (1) :  1
   Which number of admissible solutions before stopping :  4

# Parameters for diversification
   Time limit (minutes) :  30
   Maximum number of nodes :  5000000
   Minimum number of level 1 variables' changes of status :  1
```

```
   Step increment :  2

# Parameters for refining
   Time limit (minutes) :  5
   Maximum number of nodes :  5000000

# Parameters for tight refining
   Time limit (minutes) :  3
   Maximum number of nodes :  5000000
   Abs tolerance on gap between best solution and lower bound :  1e-009
   Rel tolerance on gap between best solution and lower bound :  1e-009
   Minimum number of level 2 variables' changes of status :  1
   Maximum number of level 2 variables' changes of status :  4
   Step increment :  2
```

DRT2 settings

```
/*********************************************************************
_cplexpar_DRT.dat - general information on Cplex parameters for DRT2
method.  Setting of Cplex parameters in relation to the five phases
of DRT2 research (general, big-diversification, diversification,
refining, tight ref)
*********************************************************************

Total time limit (hours) :  3
   Maximum number of diversifications :  10000
   Maximum number of big diversifications :  4

# Parameters for big diversification
   Maximum number of diversifications without improvement :  5
   Minimum Gap of improvement (percentage 0-100) :  5
   Minimum number of level 1 variables' changes of status :  5
   Step increment :  2

# Global algorithm Cplex parameters

Tree memory limit :  128
   Node storage file indicator :  1

# Parameters for initial general research
   Time limit (minutes) :  60
   Maximum number of nodes :  5000000
   Precedence to optimality (0) or to feasibility (1) :  1
```

```
   Which number of admissible solutions before stopping :  4

# Parameters for big and small diversification
   Time limit (minutes) :  10
   Maximum number of nodes :  5000000
   Minimum number of level 1 variables' changes of status :  1
   Step increment :  2

# Parameters for refining
   Time limit (minutes) :  5
   Maximum number of nodes :  5000000

# Parameters for tight refining
   Time limit (minutes) :  3
   Maximum number of nodes :  5000000
   Abs tolerance on gap between best solution and lower bound :  1e-009
   Rel tolerance on gap between best solution and lower bound :  1e-009
   Minimum number of level 2 variables' changes of status :  1
   Maximum number of level 2 variables' changes of status :  4
   Step increment :  2
```

Local Branching settings

```
TimeLimit :  10800
time for Neighborhood :  600
emphasis:  0 (OPT)
presolve:  1 (yes)
heuristic frequency :  10
precision:  0
print MIP interval :  10
number of time intervals :  100
k size :  20
original:  0 (new version:  new corresponds to the symmetric version
of the locbra cons.)
exact:  0/1 (1 exact, 0 heuristic in exact version, the code does not
perform diversifications i.e., dmax = 0)
video:  1 (YES)
```

ILOG-Cplex 7.0 settings

```
TimeLimit :  10800
MipEmphasis :  1 (emphasis on feasibility)
```

Bibliography

[1] E. Amaldi, A. Capone, F. Malucelli. *Planning UMTS Base Station Location: Optimization Models with Power Control and Algorithms.* Technical report, University of Milano, 2002

[2] E. Balas *Some thoughts on the developement of integer programming during my research career.* European Journal of Operational Research 141 pp. 1-7, Rotterdam, 2001

[3] Beasley. *OR-library. internet site: http://mscmga.ms.ic.ac.uk/info.html* 2002

[4] C. Beccari. *LaTeXGuida a un sistema per l'editoria elettronica.* Hoepli, Milano, 1991

[5] R. E. Bixby et al. *MIP: Theory and practice - Closing the gap.* Technical report, 2002.

[6] M. Dell'Amico, F. Maffioli, S. Martello. *Annotated bibliographies in combinatorial optimization.* Wiley Interscience, New York, 1997

[7] M. Fischetti, A. Lodi. *Local branching.* Technical report, University of Padova, 2002

[8] M. Fischetti, G. Romanin Jacur, J. J. Salazar Gonzáles,. *Optimization of the interconnecting network of a UMTS radio mobile telephone system.* , Technical report, University of Padova and Tenerife, 2001

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

[10] GLPK internet site:*http://www.gnu.org/directory/libs/glpk.html* 2002

[11] GNU project *GNU linear programming kit version 3.2.2. Reference manual.* Draft version october, 2002

[12] GNU project *GNU linear programming kit version 3.2.2. Modeling language GLPK/L version 3.2.* Draft version june, 2002

[13] ILOG S.A. *ILOG Concert Technology 1.0 User's Manual and Reference Manual*. ILOG, S.A., 2001 (http://www.ilog.com)

[14] ILOG S.A. *ILOG Cplex 7.0 User's Manual and Reference Manual*. ILOG, S.A., 2001 (http://www.ilog.com)

[15] A. Laspertini. *Third-generation mobile telephone systems: Optimal design of the interconnecting network.* Master Dissertation, University of Padova, Italy, 1997 (in Italian).

[16] S. B. Lippman. *C++. Corso di programmazione.* Addison-Wesley, Milano, 1996

[17] N. Mladenovíc, P. Hansen *Variable Neighborhood Search.* Computers and Operations Research 24, pp. 1097-1100, 1997

[18] T. Oetiker, H. Partl, I. Hyna, E. Schlegl. *The not so short introduction to LaTeX2ε version 3.7.* 1999

[19] C. Polo. *Algoritmi Euristici per il Progetto Ottimo di una Rete di Interconnessione.* Master Dissertation, University of Padova, Italy, 2002 (in Italian).

[20] H. Yaman *Concentrator location in telecommunication network.* Master Dissertation, University of Bruxelles, 2002