UNIVERSITÁ DEGLI STUDI DI PADOVA
FACOLTÁ DI INGEGNERIA
DIPARTIMENTO DI ELETTRONICA ED INFORMATICA

TESI DI LAUREA

INTERFACING A MIP HEURISTIC BASED ON ILOG CPLEX WITH DIFFERENT LP SOLVERS

Relatore: Prof. Matteo Fischetti

Laureando: Davide Baracco

ANNO ACCADEMICO 2003-2004

Contents

Summary

We have implemented two interfaces for a heuristic MIP solver (Feasibility Pump) based on the commercial solver ILOG CPLEX. These interfaces, without modifying the original code, capture all the calls to the CPLEX simplex solver and redirect them to other solver such as Xpress or GLPK. All the FP functionalities have been preserved. We tested the interfaces on several hard MIP instances. Using the interface for Xpress there is only a loss of speed with respect to CPLEX, while for GLPK some very hard instances cannot be solved with the default tuning. However, also in this case the results are satisfactory.

Sommario

Abbiamo realizzato due interfacce per un algoritmo euristico (Feasibility Pump) capace di risolvere problemi MIP (Mixed Integer Programming). Tale algoritmo sfrutta la libreria di funzioni del software commerciale ILOG CPLEX. Le interfacce, senza modificare il codice originale, catturano le chiamate a CPLEX e le reindirizzano ad altri solver come Dash Xpress o GLPK. I software che abbiamo sviluppato hanno permesso di preservare tutte le funzionalitá della Feasibility Pump. Essi sono stati testati su numerose istanze di problemi MIP anche piuttosto difficili. La Feasibility Pump interfacciata a Xpress accusa un modesto peggioramento delle prestazioni in termini di tempo richiesto per ogni iterazione; l'algoritmo interfacciato a GLPK, oltre a manifestare un calo prestazionale piú consistente, si é dimostrato anche incapace di risolvere molte istanze. Tuttavia i risultati ottenuti si possono considerare soddisfacenti in entrambi i casi.

Introduction

This thesis describes the implementation of two interfaces for a heuristic MIP solver (Feasibility Pump).

Feasibility Pump (FP) is an algorithm, proposed recently by M. Fischetti, A. Lodi and Fred Glover [1], whose target is to provide a feasible solution to NP-hard MIP (Mixed Integer Programming) problems. Since NP-hard problems can be extremely hard in practice, in some important cases, state-of-the-art MIP solvers may spend a very large computational effort before discovering their first solution. A complete comparison between ILOG CPLEX solver and FP can be found in [1].

As FP needs a powerful LP solver, when the algorithm was originally proposed the commercial software ILOG CPLEX was chosen. The main target of this thesis is creating two interfaces which capture the calls to CPLEX and redirect them to other solvers; namely: GLPK (an open-source software), and Xpress-MP by Dash optimizations. This target has been reached without modifying the original code, but creating a sort of emulator which converts all the CPLEX functions in GLPK or Xpress functions. This interface can be applied to other CPLEX based algorithms which uses the same functions.
We have compared the capabilities of the interfaced FP implementations through extensive computational tests.
The thesis is organized as follows:

- In Chapter 1 we introduce the FP algorithm; a more complete description and benchmark can be found in [1].
- Chapters 2, 3 and 4 give a brief description of the three main codes we used: ILOG CPLEX, Xpress-MP, and GLPK. Information is taken from [3], [5], [4].
- Chapter 5 gives a description of the interfaces and a function-by-function

guide to understand the limits of the pieces of software.

- Chapter 6 reports a description of the testbed used, all the computational results and comments upon them.
- In Chapter 7 there are final considerations about the work.

In appendix:

- A - Source code: interface for GLPK
- B - Source code: interface for Xpress-MP

Chapter 1

The Feasibility Pump

1.1 Introduction

We are introducing the Feasibility Pump, an algorithm developed by M. Fischetti, F. Glover and A. Lodi [1].
This chapter is focused on the problem of finding a feasible solution of a generic MIP problem of the form

$$(MIP) \quad \min c^T x \tag{1.1}$$

$$Ax \geq b \tag{1.2}$$

$$x_j \text{ integer} \quad \forall j \in \mathcal{I} \tag{1.3}$$

where A is an $m \times n$ matrix. This NP-hard problem can be extremely hard in practice—in some important practical cases, state-of-the-art MIP solvers may spend a very large computational effort before discovering their first solution. Therefore, heuristic methods to find a feasible solution for hard MIPs are highly important in practice. This is particularly true in recent years where successful local-search approaches for general MIPs such as local branching [2] and RINS/guided dives [12] are used that can only be applied if an initial feasible solution is known. Heuristic approaches to general MIP problems have been proposed by several authors, including [7, 8, 9, 12, 2, 14, 15, 16, 17, 18, 22, 21, 24, 25, 28].

In this chapter we show a new approach to compute heuristic MIP solutions, that is called the *Feasibility Pump*. The chapter is organized as follows. In the remaining part of this section the `FP` method is described in more detail, and then an implementation for 0-1 MIPs is shown. Computational results are presented in Section 1.2, where we have reported a comparison, taken from [1], between the

FP performance and that of the commercial software ILOG-CPLEX 8.1 on a set of 83 hard 0-1 MIPs. The possibility of reducing the computing time involved in the various LP solutions is reported in Section 1.3, where the use of approximate LP solutions is investigated. In the same section we have also reported from [1] a way to produce a sequence of feasible solutions of better and better quality using the FP scheme.

Let $P := \{x : Ax \geq b\}$ denote the polyhedron associated with the LP relaxation of the given MIP. With a little abuse of notation, we say that a point x is *integer* if x_j is integer for all $j \in \mathcal{I}$ (no matter the value of the other components). Analogously, the rounding $\widetilde{x}$ of a given x is obtained by setting $\widetilde{x}_j := [x_j]$ if $j \in \mathcal{I}$ and $\widetilde{x}_j := x_j$ otherwise, where $[\cdot]$ represents scalar rounding to the nearest integer.

We will consider the L_1-norm distance between a generic point $x \in P$ and a given integer point $\widetilde{x}$, defined as

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

Notice that the continuous variables x_j ($j \notin \mathcal{I}$), if any, do not contribute to this function. Assuming without loss of generality that the MIP constraints include the variable bounds $l_j \leq x_j \leq u_j$ for all $j \in \mathcal{I}$, we can write

$$\Delta(x, \widetilde{x}) := \sum_{j \in \mathcal{I}: \widetilde{x}_j = l_j} (x_j - l_j) + \sum_{j \in \mathcal{I}: \widetilde{x}_j = u_j} (u_j - x_j) + \sum_{j \in \mathcal{I}: l_j < \widetilde{x}_j < u_j} (x_j^+ + x_j^-)$$

where the additional variables x_j^+ and x_j^- require the introduction into the MIP model of the additional constraints:

$$x_j = \widetilde{x}_j + x_j^+ - x_j^-, \quad x_j^+ \geq 0, \; x_j^- \geq 0, \quad \forall j \in \mathcal{I} : l_j < \widetilde{x}_j < u_j \tag{1.4}$$

Given an integer point $\widetilde{x}$, the closest point $x^* \in P$ can therefore be determined by solving the LP

$$\min\{\Delta(x, \widetilde{x}) : Ax \geq b\} \tag{1.5}$$

If $\Delta(x^*, \widetilde{x}) = 0$, then $x_j^* (= \widetilde{x}_j)$ is integer for all $j \in \mathcal{I}$, so x^* (but not necessarily $\widetilde{x}$) is a feasible MIP solution. Conversely, given a point $x^* \in P$, the integer point $\widetilde{x}$ closest to x^* is easily determined by rounding x^*. These observations suggest the following Feasibility Pump (FP) heuristic to find a feasible MIP solution, in which a pair of points $(x^*, \widetilde{x})$ with $x^* \in P$ and $\widetilde{x}$ integer is iteratively updated with the aim of reducing as much as possible their distance $\Delta(x^*, \widetilde{x})$.

We start from any $x^* \in P$, and initialize a typically infeasible integer point $\widetilde{x}$ as the rounding of x^*. At each FP iteration, called a *pumping cycle*, we fix $\widetilde{x}$ and find through linear programming the point $x^* \in P$ which is as close as possible to $\widetilde{x}$. If $\Delta(x^*, \widetilde{x}) = 0$, then x^* is a MIP feasible solution, and we are done. Otherwise, we replace $\widetilde{x}$ by the rounding of x^* so as to further reduce $\Delta(x^*, \widetilde{x})$, and repeat. (This basic scheme will be slightly elaborated, as we indicate subsequently, so as to overcome possible stalling and cycling issues.)

From a geometric point of view, the FP generates two (hopefully convergent) trajectories of points x^* and $\widetilde{x}$ that satisfy feasibility in a complementary but partial way—one satisfies the linear constraints, the other the integer requirement. An important feature of the method is related to the infeasibility measure used to guide $\widetilde{x}$ towards feasibility: instead of taking a weighted combination of the degree of violation of the single linear constraints, as customary in MIP heuristics, we use the distance $\Delta(x^*, \widetilde{x})$ of $\widetilde{x}$ from polyhedron P, as computed at each pumping cycle[1]. This distance can be interpreted as a sort of "difference of pressure" between the two complementary types of infeasibility of x^* and $\widetilde{x}$, that we try to reduce by "pumping" the integrality of $\widetilde{x}$ into x^*—hence the name of the method. FP can be interpreted as a strategy for producing a sequence of roundings that leads to a feasible MIP point.

The FP can also be viewed as modified *local branching* strategy [2]. Indeed, at each pumping cycle we have an incumbent (infeasible) solution $\widetilde{x}$ satisfying the integer requirement, and we face the problem of finding a feasible solution (if any exists) within a small-distance neighborhood, i.e., changing only a small subset of its variables. In the local branching context, this subproblem would have been modeled by the MIP

$$\min\{c^T x : Ax \geq b, x_j \text{ integer } \forall j \in \mathcal{I}, \Delta(x, \widetilde{x}) \leq k\}$$

for a suitable value of parameter k, and solved through an enumerative MIP method. In the FP context, instead, the same subproblem is modeled in a relaxed way through the LP (1.5), where the "small distance" requirement is translated in terms of the objective function. (Notice that (1.5) can be viewed as a relaxed model for the problem: "Change a minimum number of variables so as to convert the current $\widetilde{x}$ into a feasible MIP solution x^*".) The working hypothesis here is that the objective function $\Delta(x, \widetilde{x})$ will discourage the optimal solution x^* of the

[1] A similar infeasibility measure for nonlinear problems was recently investigated in [11].

relaxation from being "too far" from the incumbent $\widetilde{x}$, hence we expect a large number of the integer-constrained variables in $\widetilde{x}$ will retain their (integer) values also in the optimal x^*.

In the remainder of this chapter we will focus on the important case where all integer-constrained variables are binary, i.e., we assume constraints $Ax \geq b$ include the variable bounds $0 \leq x_j \leq 1$ for all $j \in \mathcal{I}$. As a consequence, no additional variables x_j^+ and x_j^- are required in the definition of the distance function (1.4), which attains the simpler form

$$\Delta(x, \widetilde{x}) := \sum_{j \in \mathcal{I}: \widetilde{x}_j = 0} x_j + \sum_{j \in \mathcal{I}: \widetilde{x}_j = 1} (1 - x_j) \tag{1.6}$$

An outline of the FP algorithm for 0-1 MIPs is reported in Figure 1.1. The algorithm receives on input two parameters: the time limit TL and the number T of variables to be flipped (i.e., changed with respect to their current 0-1 value) at each iteration—the use of this latter parameter will be clarified later on.

```
The Feasibility Pump (basic version):
  1. initialize nIT := 0 and x* := argmin{c^T x : Ax ≥ b};
  2. if x* is integer, return(x*);
  3. let x~ := [x*] (= rounding of x*);
  4. while (time < TL) do
  5.   let nIT := nIT + 1 and compute x* := argmin{Δ(x, x~) : Ax ≥ b};
  6.   if x* is integer, return(x*);
  7.   if ∃ j ∈ I : [x*_j] ≠ x~_j then
  8.      x~ := [x*]
       else
  9.      flip the TT = rand(T/2,3T/2) entries x~_j (j ∈ I) with highest |x*_j − x~_j|
  10.  endif
  11. enddo
```

Figure 1.1: The basic FP implementation for 0-1 MIPs

At step 1, x^* is initialized as a minimum-cost solution of the LP relaxation, a choice intended to increase the chance of finding a small-cost feasible solution. At each pumping cycle, at step 5 we redefine x^* as a point in P with minimum

distance from the current integer point $\widetilde{x}$. We then check whether the new $x^* \in P$ is integer. If this is not the case, the current integer point $\widetilde{x}$ is replaced at step 8 by $[x^*]$, so as to reduce even further the current distance $\Delta(x^*, \widetilde{x})$. In order to avoid stalling issues, in case $\widetilde{x} = [x^*]$ (with respect to the integer-constrained components) we flip, at step 9, a random number $\mathtt{TT} \in \{\frac{1}{2}\mathtt{T}, \cdots, \frac{3}{2}\mathtt{T}\}$ of integer-constrained entries of $\widetilde{x}$, chosen so as to minimize the increase in the total distance $\Delta(x^*, \widetilde{x})$.

The procedure terminates as soon as a feasible integer solution x^* is found, or when the time-limit `TL` has been exceeded. In this latter case, the `FP` heuristic has to report a failure–which is not surprising, as finding a feasible 0-1 MIP solution is an NP-hard problem in general.

A main problem with the basic `FP` implementation described above is the possibility of *cycling*: after a certain number of iterations, the method may enter a loop where a same sequence of points x^* and $\widetilde{x}$ is visited again and again. In order to overcome this drawback, the following straightforward perturbation mechanism is implemented. As soon as a cycle is heuristically detected by comparing the solutions found in the last 3 iterations, and in any case after `R` (say) iterations, steps 7-10 are skipped and a random perturbation move is applied. To be more specific, for each $j \in \mathcal{I}$ a uniformly random value $\rho_j \in [-0.3, 0.7]$ is generated and, in case $|x^*_j - \widetilde{x}_j| + \max\{\rho_j, 0\} > 0.5$, $\widetilde{x}_j$ is flipped.

1.2 Computational experiments

In this section we report computational results taken from the article [1], comparing the performance of the `FP` method with that of the commercial software `ILOG-CPLEX` 8.1. The testbed is made by 44 0-1 MIP instances collected in MIPLIB 2003 [6] and described in Table 1.1, plus an additional set of 39 hard 0-1 MIPs described in Table 1.2. The two tables report the instance names and the corresponding number of variables (n), of 0-1 variables ($|\mathcal{I}|$) and of constraints (m).

The results of the initial `FP` implementation described above are reported in Tables 1.3 and 1.4, with a comparison with the state-of–the-art MIP solver `ILOG-CPLEX` 8.1. The focus of this experiment was to measure the capability of the compared methods to converge to an initial feasible solution, hence both `FP` and `ILOG-CPLEX` were stopped as soon as the first feasible solution was found.

Name	n	$\|\mathcal{I}\|$	m	Name	n	$\|\mathcal{I}\|$	m
10teams	2025	1800	230	mod011	10958	96	4480
A1C1S1	3648	192	3312	modglob	422	98	291
aflow30a	842	421	479	momentum1	5174	2349	42680
aflow40b	2728	1364	1442	net12	14115	1603	14021
air04	8904	8904	823	nsrand_ipx	6621	6620	735
air05	7195	7195	426	nw04	87482	87482	36
cap6000	6000	6000	2176	opt1217	769	768	64
dano3mip	13873	552	3202	p2756	2756	2756	755
danoint	521	56	664	pk1	86	55	45
ds	67732	67732	656	pp08a	240	64	136
fast0507	63009	63009	507	pp08aCUTS	240	64	246
fiber	1298	1254	363	protfold	1835	1835	2112
fixnet6	878	378	478	qiu	840	48	1192
glass4	322	302	396	rd-rplusc-21	622	457	125899
harp2	2993	2993	112	set1ch	712	240	492
liu	1156	1089	2178	seymour	1372	1372	4944
markshare1	62	50	6	sp97ar	14101	14101	1761
markshare2	74	60	7	swath	6805	6724	884
mas74	151	150	13	t1717	73885	73885	551
mas76	151	150	12	tr12-30	1080	360	750
misc07	260	259	212	van	12481	192	27331
mkc	5325	5323	3411	vpm2	378	168	234

Table 1.1: The 44 0-1 MIP instances collected in MIPLIB 2003 [6]

Computing times are expressed in CPU seconds, and refer to a Pentium M 1.6 Ghz notebook with 512 MByte of main memory. Parameters `T` and `TL` were set to 20 and 1,800 CPU seconds, respectively, while the perturbation-frequency parameter `R` was set to 100.

In the `FP` implementation, the `ILOG-CPLEX` function `CPXoptimize` is preferred to solve each LP (thus leaving to `ILOG-CPLEX` the choice of the actual LP algorithm to invoke) with the default parameter setting.

As to `ILOG-CPLEX`, after extensive experiments and contacts with `ILOG-CPLEX` staff [27] the authors found that, as far as the time and quality of the root node solution is concerned, the best results are obtained (perhaps surprisingly) when the MIP preprocessing/presolve is not invoked, and the default "balance optimal-

Name	n	$\|\mathcal{I}\|$	m	source
biella1	7328	6110	1203	[2]
NSR8K	38356	32040	6284	[2]
dc1c	10039	8380	1649	[13]
dc1l	37297	35638	1653	[13]
dolom1	11612	9720	1803	[13]
siena1	13741	11775	2220	[13]
trento1	7687	6415	1265	[13]
rail507	63019	63009	509	[2]
rail2536c	15293	15284	2539	[2]
rail2586c	13226	13215	2589	[2]
rail4284c	21714	21705	4284	[2]
rail4872c	24656	24645	4875	[2]
A2C1S1	3648	192	3312	[2]
B1C1S1	3872	288	3904	[2]
B2C1S1	3872	288	3904	[2]
sp97ic	12497	12497	1033	[2]
sp98ar	15085	15085	1435	[2]
sp98ic	10894	10894	825	[2]
bg512142	792	240	1307	[23]
dg012142	2080	640	6310	[23]

Name	n	$\|\mathcal{I}\|$	m	source
blp-ar98	16021	15806	1128	[22]
blp-ic97	9845	9753	923	[22]
blp-ic98	13640	13550	717	[22]
blp-ir98	6097	6031	486	[22]
CMS750_4	11697	7196	16381	[19]
berlin_5_8_0	1083	794	1532	[19]
railway_8_1_0	1796	1177	2527	[19]
usAbbrv.8.25_70	2312	1681	3291	[19]
manpower1	10565	10564	25199	[26]
manpower2	10009	10008	23881	[26]
manpower3	10009	10008	23915	[26]
manpower3a	10009	10008	23865	[26]
manpower4	10009	10008	23914	[26]
manpower4a	10009	10008	23866	[26]
ljb2	771	681	1482	[12]
ljb7	4163	3920	8133	[12]
ljb9	4721	4460	9231	[12]
ljb10	5496	5196	10742	[12]
ljb12	4913	4633	9596	[12]

Table 1.2: The additional set of 39 0-1 MIP instances

ity and integer feasibility" strategy for the exploration of the search tree is used. Indeed, the number of root-node failures for `ILOG-CPLEX` was 19 with the setting used in the experiments. By contrast, when the preprocessing/presolve was activated `ILOG-CPLEX` could not find any feasible solution at the root node in 25 cases (with the default "balance optimality and integer feasibility" strategy) or in 41 cases (with the "emphasize integrality" strategy). In case the preprocessing/presolve is deactivated but the "emphasize integrality" strategy was used, instead, no solution was found at the root node in 33 cases.

Tables 1.3 and 1.4 report the results for the instances in Tables 1.1 and 1.2, respectively. For each instance and for each algorithm (`FP` and `ILOG-CPLEX`) the value of the first feasible solution found ("value" for `FP`, and "root value/first value" for `ILOG-CPLEX`) and the corresponding computing time are reported. In case of failure, "N/A" is reported. Moreover, for `FP` you find the number of iterations performed by the algorithm ("nIT"), while, for `ILOG-CPLEX` you find the

	feasibility pump			ILOG-CPLEX 8.1			
name	value	nIT	time	root value	first value	nodes	time
10teams	992.00	53	7.5	N/A	924.00	14	5.2
A1C1S1	18,377.24	5	3.8	N/A	14,264.61	120	8.6
aflow30a	4,545.00	18	0.1	N/A	1,574.00	40	1.4
aflow40b	6,859.00	7	0.5	1,786.00		0	1.8
air04	58,278.00	4	12.5	57,640.00		0	6.2
air05	29,937.00	2	3.4	29,590.00		0	2.0
cap6000	-2,354,320.00	2	0.6	-2,445,344.00		0	0.6
dano3mip	756.62	4	77.7	768.37		0	161.2
danoint	77.00	3	0.2	73.00		0	1.7
ds	N/A	81	1,800.0	5,418.56		0	81.6
fast0507	181.00	4	34.0	209.00		0	33.1
fiber	1,911,617.79	2	0.0	570,936.07		0	0.0
fixnet6	9,131.00	4	0.0	12,163.00		0	0.0
glass4	4,650,037,150.00	23	0.1	N/A	3,500,034,900.00	162	0.3
harp2	-43,856,974.00	654	4.5	-73,296,664.00		0	0.1
liu	6,262.00	0	0.0	6,262.00		0	0.0
markshare1	1,064.00	11	0.0	710.00		0	0.0
markshare2	1,738.00	7	0.0	1,735.00		0	0.0
mas74	52,429,700.59	1	0.0	19,197.47		0	0.0
mas76	194,527,859.06	1	0.0	44,877.42		0	0.0
misc07	4,515.00	123	0.5	3,060.00		0	0.0
mkc	-164.56	2	0.3	-195.97		0	0.5
mod011	-49,370,141.17	0	1.0	-42,902,314.08		0	1.9
modglob	35,147,088.88	0	0.0	20,786,787.02		0	0.0
momentum1	455,740.91	520	1478.4	N/A	N/A	75	1,800.0
net12	337.00	346	55.4	N/A	214.00	480	1,593.7
nsrand_ipx	340,800.00	3	0.7	699,200.00		0	0.3
nw04	19,882.00	1	2.9	17,306.00		0	5.1
opt1217	-12.00	0	0.0	-14.00		0	0.0
p2756	N/A	163435	1,800.0	3,485.00		0	0.1
pk1	57.00	1	0.0	89.00		0	0.0
pp08a	11,150.00	2	0.0	14,800.00		0	0.0
pp08aCUTS	10,940.00	2	0.0	13,540.00		0	0.0
protfold	-10.00	367	493.8	N/A	N/A	637	1,800.0
qiu	389.36	3	0.3	1,691.14		0	0.1
rd-rplusc-21	N/A	900	1,800.0	N/A	N/A	372	1,800.0
set1ch	76,951.50	2	0.0	109,759.00		0	0.0
seymour	452.00	9	3.4	469.00		0	5.1
sp97ar	1,398,705,728.00	6	4.3	734,171,023.04		0	2.6
swath	18,416.00	109	4.7	N/A	826.66	1609	38.6
t1717	826,848.00	42	644.9	N/A	N/A	1397	1,800.0
tr12-30	277,218.00	9	0.1	N/A	143,586.00	200	2.1
van	8.21	4	245.0	6.59		0	100.3
vpm2	19.25	3	0.0	15.25		0	0.0

Table 1.3: Convergence to a first feasible solution

	feasibility pump			ILOG-CPLEX 8.1			
name	value	nIT	time	root value	first value	nodes	time
biella1	3,537,959.54	5	7.9	3,682,135.10		0	8.4
NSR8K	5,111,376,832.18	5	1,751.4	4,923,673,379.32		0	1,478.6
dc1c	27,348,312.19	4	19.3	33,458,468.26		0	15.3
dc1l	8,256,022.49	5	94.4	752,840,672.81		0	67.6
dolom1	298,684,615.17	7	32.1	584,923,856.01		0	29.2
siena1	104,004,996.99	5	91.8	591,385,634.57		0	66.4
trento1	356,179,003.01	2	17.8	621,044,078.07		0	18.1
rail507	178.00	2	41.1	205.00		0	32.9
rail2536c	715.00	4	26.7	771.00		0	27.1
rail2586c	1,007.00	5	81.6	1,072.00		0	68.6
rail4284c	1,124.00	3	1095.8	1,218.00		0	273.1
rail4872c	1,614.00	5	311.9	1,737.00		0	305.6
A2C1S1	19,879.93	5	3.7	20,865.33		0	0.0
B1C1S1	38,530.65	7	5.2	69,933.52		0	0.1
B2C1S1	48,279.95	6	4.5	70,625.52		0	0.1
sp97ic	1,280,793,707.52	3	2.7	515,786,416.96		0	1.7
sp98ar	988,402,511.36	4	4.4	599,527,422.56		0	2.4
sp98ic	959,924,716.00	3	2.1	550,157,878.72		0	1.5
blp-ar98	25,094.03	161	23.6	N/A	9,473.66	50	37.2
blp-ic97	7,874.87	4	0.7	6,408.43		0	0.4
blp-ic98	14,848.96	6	1.4	9,080.53		0	0.6
blp-ir98	5,388.84	3	0.3	2,927.29		0	1.2
CMS750_4	606.00	131	18.9	803.00		0	13.9
berlin_5_8_0	79.00	10	0.1	89.00		0	0.4
railway_8_1_0	440.00	13	0.3	478.00		0	0.4
usAbbrv.8.25_70	164.00	34	0.8	N/A	130.00	6036	46.8
bg512142	120,738,665.00	0	0.1	120,670,203.50		0	0.3
dg012142	153,406,945.50	0	0.8	153,392,273.00		0	1.7
manpower1	8.00	66	38.5	N/A	N/A	34	1,800.0
manpower2	7.00	148	157.9	N/A	N/A	10	1,800.0
manpower3	6.00	49	56.9	N/A	N/A	10	1,800.0
manpower3a	6.00	73	67.4	N/A	N/A	10	1,800.0
manpower4	7.00	192	107.7	N/A	N/A	17	1,800.0
manpower4a	7.00	53	85.1	N/A	N/A	16	1,800.0
ljb2	7.24	0	0.0	1.63		0	0.4
ljb7	8.61	0	0.5	0.81		0	3.9
ljb9	9.48	0	0.8	9.48		0	6.2
ljb10	7.31	0	1.0	7.31		0	6.9
ljb12	6.20	0	0.7	3.21		0	6.4

Table 1.4: Convergence to a first feasible solution (cont.d)

number of branch-and-bound nodes ("nodes") needed to initialize the incumbent solution.

The first order of business here was to evaluate the percentage of success in finding a feasible MIP solution without resorting to branching. In this respect, the FP performance is very satisfactory: whereas ILOG-CPLEX could not find any feasible solution at the root node in 19 cases (and in 10 cases even allowing for 1,800 seconds of branching), FP was unsuccessful only 3 times.

Also interesting is the comparison of the quality of the FP solution with that found by the root-node ILOG-CPLEX heuristics: the latter delivered a strictly-better solution in 33 cases, whereas the solution found by FP was strictly better in 46 cases. The computing times to get to the first feasible solution appear comparable: excluding the instances for which both methods required less than 1 second, ILOG-CPLEX was faster in 26 cases, and FP was faster in 31 cases. Finally, column nIT (FP iterations) shows that the number of LPs solved by FP for finding its first feasible solution is typically very small, which confirms the effectiveness of the distance function used at step 5 in driving x^* towards integrality.

Quite surprisingly, sometimes FP requires just a few iterations but takes much more time than expected. E.g., for problem rail4284c in Table 1.4 the root node of ILOG-CPLEX took only 273.1 seconds—including the application of the internal heuristics. FP found a feasible solution after just 3 iterations but the overall computing time was 1095.8 seconds—about 4 times larger. This can be partly explained by observing that FP requires the initial solution of *two* LPs with different objective functions: the initialization LP at step 1 (which uses the original objective function), and the LP at the first execution of step 5 (using the distance-related objective function). Hence we take for granted that no effective parametrization between these two LPs can be obtained. However, a better integration of FP with the LP solver is likely to produce improved results in several cases.

As already stated, in the experiments any problem-dependent fine tuning of the LP parameters were deliberately avoided, and for both FP and ILOG-CPLEX their default values were used. However, some knowledge of the type of instance to be solved can improve both the FP and ILOG-CPLEX performance considerably, especially for highly degenerate cases. For instance, the choice of the LP algorithm used for re-optimization at step 5 may have a strong impact on the overall FP computing times. E.g., if you force the use of the dual simplex, the overall computing time for rail4284c decreases from 1095.8 to just 311.1 sec-

onds. This is of course true also for ILOG-CPLEX. E.g., for manpower instances Bixby [10] suggested an ad-hoc tuning consisting of (a) avoiding the generation of cuts (set mip cut all -1), and (b) activating a specific dual-simplex pricing algorithm (set simp dg 2). This choice considerably reduces the time spent by the LP solver at each branching node, and allows ILOG-CPLEX to find a first feasible solution (of value 6.0) for instances manpower1, manpower2, manpower3, manpower3a, manpower4 and manpower4a after 111, 150, 107, 156, 202 and 197 branching nodes, and after 28.4, 115.4, 99.7, 70.7, 100.2, and 84.7 CPU seconds, respectively.

A pathological case for FP is instance p2756, which can instead be solved very easily by ILOG-CPLEX. This is due to the particular structure of this problem, which involves a large number of big-M coefficients. More specifically, several constraints in this model are of the type $\alpha_i^T y \leq \beta_i + M_i z_i$, where M_i is a very large positive value, y is a binary vector, and z_i is a binary variable whose value 1 is used to actually deactivate the constraint. Feasible solutions of this model can obtained quite easily by setting $z_i = 1$ so as to deactivate these constraints. However, this choice turns out to be very expensive in terms of the LP objective function, where variables z_i are associated with large costs. Therefore, the LP solutions (y^*, z^*) tend to associate very small values to all variables z_i^*, namely $z_i^* = \max\{0, (\alpha_i^T y^* - \beta_i)/M_i\}$, which are then systematically rounded down by our scheme. As a consequence, FP is actually looking for a feasible y that fulfills *all* the constraints $\alpha_i^T x \leq \beta_i$—an almost impossible task.

1.3 FP variants

The basic FP scheme will next be elaborated in the attempt of improving (a) the required computing time, and/or (b) the quality of the heuristic solution delivered by the method.

1.3.1 Reducing the computing time

In the article are evaluated the following two simple FP variants:

1. FP1: At step 1, the LP relaxation of the original MIP (i.e., the one with the original objective function $c^T x$) is solved approximately through a primal-dual method (e.g., the ILOG-CPLEX barrier algorithm), and as soon as a

prefixed primal-dual gap γ is reached the execution is stopped and no crossover is performed. The almost-optimal dual variables are then used as Lagrangian multipliers to compute a mathematically-correct lower bound on the optimal LP value. Moreover, at step 5 each LP relaxation is solved approximately via the primal simplex method with a limit of `SIL` simplex pivots (if this limit is reached within the simplex phase 1, the approximate LP solution x^* is not guaranteed to be primal feasible, hence we skip step 6).

2. `FP2`: The same as `FP1`, but at step 1 the first $\widetilde{x}$ is obtained by just rounding a random initial solution $x^* \in [0,1]^n$ (no LP solution is required).

1.3.2 Improving the solution quality

As stated, the `FP` method is designed to provide a feasible solution to hard MIPs—no particular attention is paid to the *quality* of this solution. In fact, the original MIP objective function is only used for the initialization of $\widetilde{x}$ in step 1—while it is completely ignored in variant `FP2` above. On the other hand, `FP` proved quite fast in practice, and one may think of simple modifications to provide a *sequence* of feasible solutions of better and better quality.[2] The authors have therefore investigated a natural extension of our method, based on the idea of adding the upper-bound constraint $c^T x \leq UB$ to the LPs solved at step 5, where UB is updated dynamically each time a new feasible solution is found. To be more specific, right after step 1 we initialize $z^*_{LP} = c^T x^*$ (= LP relaxation value) and $UB = +\infty$. Each time a new feasible solution x^* of value $z^H = c^T x^*$ is found at step 5, we update $UB = \alpha z^*_{LP} + (1-\alpha) z^H$ for $\alpha \in (0,1)$, and continue the while-do loop. Furthermore, in the test at step 4 the condition `nIT-nIT0 < IL` is added, where `nIT0` gives the value of `nIT` when the first feasible solution is found (`nIT0`=0 if none is available), and the input parameter `IL` gives the maximum number of additional `FP` iterations allowed after the initialization of the incumbent solution.

The above scheme can also be applied to variant `FP1`, where the LP at step 1 is solved approximately. As to `FP2`, where no bound is computed, z^*_{LP} is left

[2] A possible way to improve the quality of the first solution found by `FP` is of course to exploit local-search methods based on enumeration of a suitable solution neighborhood of the first feasible solution found, such as the recently-proposed local branching [2], RINS or guided dives [12] schemes.

undefined and the upper bound UB is heuristically reduced after each solution updating as $UB = z^H - \beta|z^H|$ (assuming $z^H \neq 0$).

A final comment is in order. Due to the additional constraint $c^T x \leq UB$, it is often the case that the integer components of $\widetilde{x}$ computed at step 8 define a feasible point for the *original* system $Ax \geq b$, but not for the current one. In order to improve the chances of updating the incumbent solution, right after step 8, a simple post-processing of $\widetilde{x}$ is applied, consisting in solving the LP $\min\{c^T x : Ax \geq b,\ x_j = \widetilde{x}_j\ \forall j \in \mathcal{I}\}$ and comparing the corresponding solution (if any exists) with the incumbent one.

1.3.3 Computational results

Table 1.5 reports the results of the feasibility pump variants FP1 and FP2. For this experiment, 26 instances out of the 83 in our testbed were selected, chosen as those for which (a) both FP and ILOG-CPLEX were able to find a solution within the time limit of 1,800 CPU seconds, and (b) the computing time required by either ILOG-CPLEX or FP was at least 10 CPU seconds. Also the manpower instances were included, and ran ILOG-CPLEX with the ad-hoc tuning described in the previous section.

For this reduced testbed, you find an evaluation of the capability of FP1 and FP2 to converge quickly to an initial solution (even if worse than that produced by FP) and to improve it in a given amount of additional iterations. The underlying idea is that, for problems in which the LP solution is very time consuming, it may be better to solve the LPs approximately, while trying to improve the first (possibly poor) solutions at a later time.

For the experiments reported in Table 1.5 the parameters were set as follows: $\alpha = 0.50$, $\beta = 0.25$, $\gamma = 0.20$, SIL $= 1,000$, and IL $= 250$.

In the table, the ILOG-CPLEX columns are taken from the previous experiments. For both FP1 and FP2 there are the time and value of the first solution found, and the time and value of the best solution found after IL=250 additional FP iterations. Moreover, for FP1 the extra computing time spent for computing the initial lower bound through the (approximate use of) ILOG-CPLEX barrier method ("LB time") is reported.

According to the table, FP2 is able to deliver its first feasible solution within an extremely short computing time—often 1-2 orders of magnitude shorter than ILOG-CPLEX and FP. E.g., FP2 took only 1.5 seconds for NSR8K, whereas ILOG-CPLEX

and FP required 1,478.6 and 1,751.4 seconds, respectively. In three cases however the method did not find any solution within the 1,800-second time limit. The quality of the first solution is of course poor (remember that the MIP objective function is completely disregarded until the first feasible solution is found), but it improves considerably during subsequent iterations. At the end of its execution, FP2 was faster than ILOG-CPLEX in 12 out of the 26 cases, and returned a better (or equal) solution in 11 cases.

FP1 performs somewhat better than this. Its first solution is much better than that of FP2 and strictly better than the ILOG-CPLEX solution in 4 cases; the corresponding computing time (increased by the LB time) is shorter than that of ILOG-CPLEX in 22 out of the 26 cases. After 250 more FP iterations, the quality of the FP1 solution is equal to that of ILOG-CPLEX in 6 cases, strictly better in 12 cases, and worse in 8 cases; the corresponding computing time compares favorably with that of ILOG-CPLEX in 12 cases.

	ILOG-CPLEX 8.1		FP2: no bound, random initial solution					FP1: approximate solution of LPs					
name	first value	time	first value	nIT	time	best value	time	LB time	first value	nIT	time	best value	time
air04	57,640.00	6.2	N/A	5210	1,800	N/A	1,800	1.3	62,398.00	599	441.8	59,807.00	973.8
dano3mip	768.37	161.2	N/A	5531	1,800	N/A	1,800	12.1	2,649,999.80	241	64.2	155,597.13	143.1
fast0507	209.00	33.1	60,770.00	1	0.6	198.00	63.7	4.2	205.00	2	1.2	188.00	25.7
net12	214.00	1593.7	337.00	1259	116.4	337.00	136.7	42.9	337.00	63	6.4	337.00	26.3
swath	826.66	38.6	46,277.73	39	2.1	1,512.21	17.1	0.3	45,023.47	382	15.8	45,023.47	17.1
van	6.59	100.3	N/A	517	1,800	N/A	1,800	72.5	22.75	159	649.7	22.75	1,730.2
NSR8K	4,923,673,379.32	1478.6	3,431,645,501.71	2	1.5	279,901,658.55	108.4	218.4	3,568,380,063.65	3	2.0	268,661,660.39	340.5
dc1c	33,458,468.26	15.3	195,977,054.50	4	0.8	10,732,532.92	87.3	12.6	8,644,498,480.28	2	0.4	5,074,719.02	97.8
dc1l	752,840,672.81	67.6	41,577,097,275.19	0	0.3	27,865,094.81	167.2	11.7	126,035,913.11	2	1.7	11,498,038.84	172.4
dolom1	584,923,856.01	29.2	431,992,801.00	28	13.3	149,854,956.11	116.3	12.4	475,952,465.07	24	10.8	155,077,538.15	130.3
siena1	591,385,634.57	66.4	8,883,564,918.89	1	0.6	139,122,554.83	360.9	44.6	953,570,679.98	3	1.3	430,116,204.90	340.5
trento1	621,044,078.07	18.1	1,296,470,184.01	15	3.0	65,746,910.00	137.4	8.4	1,296,470,184.01	15	3.0	86,011,231.01	38.9
rail507	205.00	32.9	20,251.00	1	0.8	220.00	41.9	5.2	247.00	2	1.8	187.00	89.9
rail2536c	771.00	27.1	2,430.00	1	0.3	717.00	553.3	14.5	919.00	1	0.3	718.00	450.1
rail2586c	1,072.00	68.6	2,900.00	1	0.2	1,122.00	134.8	5.1	1,376.00	1	0.3	1,028.00	735.4
rail4284c	1,218.00	273.1	4,531.00	1	0.5	2,067.00	113.3	50.7	1,554.00	2	0.8	1,174.00	121.2
rail4872c	1,737.00	305.6	4,513.00	2	0.8	3,385.00	108.1	17.7	2,132.00	2	1.1	1,611.00	197.7
blp-ar98	9,473.66	37.2	25,459.18	562	64.5	25,459.18	84.5	1.6	24,876.87	959	106.7	24,876.87	111.5
CMS750_4	803.00	13.9	1,000.00	4	2.9	748.00	34.9	1.0	1,000.00	3	1.9	742.00	33.1
usAbbrv.8.25_70	130.00	46.8	195.00	3	0.1	195.00	4.8	0.1	189.00	8	0.2	180.00	4.1
manpower1	6.00*	28.4	9.00	13	3.3	7.00	12.7	38.8	12.00	21	4.5	6.00	14.1
manpower2	6.00*	115.4	8.00	53	11.5	6.00	20.5	55.6	8.00	24	6.0	6.00	14.7
manpower3	6.00*	99.7	7.00	21	5.1	7.00	13.7	50.4	11.00	85	17.4	6.00	26.3
manpower3a	6.00*	70.7	10.00	120	25.9	6.00	35.0	52.6	9.00	64	14.7	6.00	23.6
manpower4	6.00*	100.2	6.00	169	36.6	6.00	45.1	49.6	10.00	43	9.4	6.00	18.0
manpower4a	6.00*	84.7	7.00	24	6.3	7.00	14.7	52.1	9.00	21	6.0	6.00	14.7

Table 1.5: Performance of two FP variants (* ILOG-CPLEX was run with an ad-hoc tuning)

Chapter 2

ILOG CPLEX

2.1 Introduction

ILOG CPLEX is a tool for solving linear optimization problems, commonly referred to as Linear Programming (LP) problems, of the form:

Maximize (or Minimize) $c_1x_1 + c_2x_2 + ... + c_nx_n$

subject to
$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n \sim b_1 \\ &a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n \sim b_2 \\ &... \\ &a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n \sim b_m \end{aligned}$$

with these bounds
$$\begin{aligned} &l_1 \leq x_1 \leq u_1 \\ &... \\ &l_n \leq x_n \leq u_n \end{aligned}$$

where $\sim$ can be $\leq$, $\geq$ or $=$, and the upper bounds u_i and lower bounds l_i may be positive infinity, negative infinity, or any real number.
The optimal solution that CPLEX computes and returns is:

Variables x_1, x_2, ... , x_n

CPLEX also can solve several extensions to LP:

- Network Flow problems, a special case of LP that CPLEX can solve much

faster by exploiting the problem structure.

- Quadratic Programming (QP) problems, where the LP objective function is expanded to include quadratic terms.

- Mixed Integer Programming (MIP) problems, where any or all of the LP or QP variables are further restricted to take integer values in the optimal solution (and where MIP itself is extended to include constructs like Special Ordered Sets (SOS) and semi-continuous variables).

2.2 ILOG CPLEX Technologies

CPLEX comes in three forms to meet a wide range of users' needs:

- The CPLEX Interactive Optimizer is an executable program that can read a problem interactively or from files in certain standard formats, solve the problem, and deliver the solution interactively or into text files. The program consists of the file cplex.exe on Windows platforms or CPLEX on UNIX platforms.

- Concert Technology is a set of C++ and Java class libraries offering an API that includes modeling facilities to allow the programmer to embed CPLEX optimizers in C++ or Java applications. The Concert Technology libraries make use of the Callable Library

- The CPLEX Callable Library is a C library that allows the programmer to embed CPLEX optimizers in applications written in C, Visual Basic, FORTRAN, or any other language that can call C functions.The library is provided in files cplex81.lib and cplex81.dll on Windows platforms, and in libcplex.a, libcplex81.so, and libcplex81.sl on UNIX platforms.

2.3 CPLEX Algorithms

ILOG CPLEX algorithms can be accessed from the CPLEX Component Libraries as well as the CPLEX Interactive Optimizer, an easy-to-use interactive program. CPLEX provides all the basic features and utilities for using these solvers: sophisticated problem preprocessing; file reading and writing utilities;

reporting; messaging control; interactive revision capability; efficient restart from an advanced basis; sensitivity analysis; and an infeasibility finder.

2.3.1 CPLEX Simplex Optimizers

CPLEX Simplex Optimizers provide the power to solve quadratic programs and linear programs with millions of constraints and continuous variables, at record-breaking speed.
ILOG CPLEX Simplex Optimizers are fast, robust implementations of the dual simplex and primal simplex methods for linear and quadratic programming. CPLEX Simplex Optimizers also provide lightning-fast implementation of the network simplex method. Specially suited for pure network problems, the network simplex method can even solve problems that have side constraints.

All ILOG CPLEX algorithms are tightly integrated with cutting-edge presolve algorithms. These algorithms reduce problem size and provide significant reductions in solve times, without requiring any special user intervention. Each optimizer has numerous options that enable performance to be tuned for specific problems.
`Simplex algorithm features`

- Multiple crash basis options
- Primal and dual steepest-edge algorithms
- IIS finder for detecting problem infeasibilities
- Sophisticated degeneracy resolution
- Efficient restarts from existing bases or solutions
- Integrated and automatic problem-reduction algorithms with preprocessing and postprocessing

`Network simplex algorithm features`

- Natural node/arc network representation
- Automatic network extraction
- Multiple pricing algorithms
- Efficient restarts from advanced network bases

2.3.2 CPLEX Barrier Optimizers

CPLEX Barrier Optimizer provides an alternative to the simplex method for solving linear and quadratic programs. It also offers a fast, robust method for solving quadratically constrained programs. Based on a primal-dual, predictor-corrector method, CPLEX Barrier Optimizer provides unsurpassed performance for large-scale linear programs.
All ILOG CPLEX algorithms are tightly integrated with cutting-edge presolve algorithms. These algorithms reduce problem size and provide significant reductions in solve times, without requiring special user intervention. Numerous options enable each optimizer's performance to be tuned for specific problems.
CPLEX Barrier Optimizer includes the fast, robust ILOG CPLEX crossover algorithm. Nonbasic solutions created by the ILOG CPLEX barrier algorithm are converted into basic solutions. Typically provided by the simplex method, these basic solutions are used for fast restarts and sensitivity analysis.
`Features of ILOG CPLEX barrier algorithm`

- Fast crossover to basic solutions
- Integrated and automatic problem-reduction algorithms with preprocessing and postprocessing
- Facilities for handling dense columns
- Multiple ordering algorithms
- State-of-the-art Cholesky factorization algorithms, tuned for specific platforms
- Tight integration with other CPLEX optimizers
- Solutions available without use of crossover algorithm
- Available in Parallel CPLEX on specific platforms
- Available for solving MIP subproblems
- Primal and dual crossover algorithms

2.3.3 CPLEX Mixed Integer Optimizer

ILOG CPLEX Mixed Integer Optimizer employs a branch-and-bound technique that takes advantage of innovative, cutting-edge strategies. It provides fast, robust solutions to the most difficult mixed integer programs.
CPLEX incorporates and expands on the latest results of worldwide research in mixed integer programming. Default settings and parameter selections work well for many problems. Users may also customize the branching process, or select specialized techniques that take advantage of structures in their specific problems.

CPLEX Mixed Integer Optimizer solves mixed-integer linear programs (MILP); mixed-integer quadratic programs (MIQP); and mixed-integer quadratically constrained programs (MIQCP). Implementation includes the CPLEX presolve algorithm and sophisticated cutting-plane strategies such as Gomory, clique and cover, flow cover, GUB cover and implied bound.

Users have full control of ILOG CPLEX Mixed Integer Optimizer. Customize node and variable selection strategies. Control the frequency and type of CPLEX heuristics applied to find integer feasible solutions. Users can also tell CPLEX whether it is more important to find an optimal solution or quickly determine a good feasible solution – CPLEX Mixed Integer Optimizer will automatically adjust its strategy to user needs.
`Features of the Mixed Integer Algorithm`

- Multiple types of cutting planes
  - Gomory fractional
  - Flow covers
  - GUB covers
  - Implied bound
  - Mixed integer rounding
  - Flow paths
  - Disjunctive
  - Cliques
  - Covers
- User choices for emphasizing optimality or feasibility

- Special ordered sets (SOS)
- Heuristics
- Integrated and automatic mixed-integer problem reduction algorithms with preprocessing and postprocessing
- Breadth-first, best-first or depth-first search
- User-defined branching priorities and directions
- User-determined node selection algorithms
- User-determined variable selection options
- Multiple LP algorithm options for nodes and initial relaxation
- Cut-off and shortcut techniques
- Customized branching strategies
- User-defined memory controls, allowing disk storage to be efficiently used as secondary memory
- Probing
- Available in Parallel CPLEX

Chapter 3

Xpress-MP by Dash Optimization

3.1 Xpress-MP overview

Dash develops and commercializes Xpress-MP [5], one of the best software products for modeling and optimization. Xpress-MP has been applied in sectors as diverse as manufacturing, processing, distribution, retailing, transport, finance and investment. The guiding principles of the development of this software are:

- Capability to solve large, difficult problems
- Speed of solution
- Ease of use
- Reliability
- Ease of integration

Xpress-MP is a suite of optimization software, used to solve linear, integer, quadratic and non-linear optimization problems. The main components of this suite are next described.

The Xpress-Optimizer

The Xpress-Optimizer features three optimization algorithms which enable the

user to solve linear programming problems (LP), mixed integer programming problems (MIP), quadratic programming problems (QP), mixed integer quadratic programming problems (MIQP), non-linear programming problems (NLP), and mixed interger non-linear programming problems (MINLP). These three algorithms correspond to three different optimizers which are:

- the simplex optimizer, which includes primal and dual methods and solves LP problems; it is also used within a branch-and-bound framework to solve MIP and MIQP problems;
- The Newton barrier optimizer is an interior point method for solving LP and QP problems.
- The MIP/MIQP optimizer uses a sophisticated branch-and-bound algorithm to solve MIP and MIQP problems, and is particularly known for its ability to find high quality solutions fast. MIP problems can have an exponential number of possible solutions, and the essential property of the Xpress MIP optimizer is its ability to cut down the number of solutions to a manageable size, and then to navigate through them so it can find good ones quickly. Some of the more sophisticated techniques include various classes of cutting planes, which are generated automatically during the optimization to improve the quality of bounds and reduce the size of the search (so the MIP algorithm is really called "branch-cut"). The presolve is particularly effective on MIP problems, as it is able to tighten the formulation, which improves the quality of initial solutions and enables better cutting planes to be generated.

Xpress-MP uses ultra-efficient sparse matrix handling allowing it to solve the largest problems in record time. A presolve procedure reduces the size of the problem before it is solved, sometimes by an order of magnitude. Xpress-MP is also noted for its ability to solve numerically hard or unstable problems.

Xpress-Parallel

When it is important to solve MIP problems in the shortest possible time, or to obtain solutions for the hardest MIP problems, Xpress-Parallel is the ideal solution. Operating on multi-processor machines it enables the user to harness

parallel computing power to solve MIP problems in parallel.

Xpress-SLP

Xpress-SLP is able to solve non-linear (NLP) and mixed integer non-linear (MINLP) problems. Xpress-SLP is the world's first large-scale, globally supported MINLP component and is able to solve a much wider range of optimization problems than is possible with the LP, QP and MIP optimizers. Xpress-SLP has full modeling support within Xpress-Mosel and Xpress-IVE, and the full range of embedding and interfacing capabilities enjoyed by the LP/MIP/QP and MIQP optimizers available within the Xpress-MP family.

Xpress-MP is also useful allowing the user to define their problem in different ways through modeling interfaces;

Xpress-Mosel

Xpress-Mosel allows the users to formulate their problem, solve it using the Xpress-Optimizer, and analyze the solution using a programming language specifically designed for the purpose. Mosel programs are compiled, which makes them fast and hides the intellectual property within them from end-users. They can be run interactively or embedded within an application.Mosel includes extension libraries, one of which provides direct control of the Xpress-Optimizer, through optimization statements in the Mosel program. Moreover provides an ODBC data driver which enables the user to interface directly to all common databases and spreadsheets, and data can also be imported/exported directly from/to text files.

Xpress-IVE

It is a visual development environment which makes the process od modeling easier. It gives full support for arbitrary ranges, index sets, and sparse objects means even the largest and most sophisticated problem can be expressed clearly and concisely, and completely independently of a particular data instance.

Xpress-BCL

Xpress-BCL is an object-oriented library for building problems within an application. It uses a step-by-step approach, with functions to add a variable, and add a constraint, which the developer calls within their C/C++ or Java program, gradually building up the complete problem. Once the complete problem is defined, it is solved using the Xpress-Optimizer. Further BCL functions enable the developer to access the solution directly within their application.

Libraries and console

Xpress-Optimizer, Xpress-BCL and Xpress-Mosel are available as libraries, enabling the developer to embed the algorithms of Xpress-MP within their application. This functionality allows the developer to define a problem, solve it, and examine the solution, all within their application code. Moreover the developer can control and tune the optimization algorithms, manipulate the problem after it has been solved, building up optimization heuristics and techniques, and handle multiple problems in different threads. At the lowest level, the developer can declare callback functions to interact with the Xpress-Optimizer during the MIP optimization process, to generate cutting planes and implement their own branching strategies. The primary library interface for all products is C/C++. This is also the library I used to implement my interface for the FP code. Various other interfaces are also available, such as Java and VB.

The Xpress-Optimizer and Xpress-Mosel are also available in console form, that is, stand alone executables controlled using simple text driven interfaces. They have low overhead and development requirements, and offer a straightforward means to get simple batch-driven applications up and running with the minimum of effort.

Chapter 4

GLPK by Andrew Makhorin

4.1 GLPK overview

GLPK [4] stands for the GNU Linear Programming Kit. It is a set of routines written in ANSI C and organized in the form of a callable library. This package is intended for solving large-scale linear programming (LP), mixed integer linear programming (MIP), and other related problems. The GLPK package includes the following main components:

- implementation of the simplex method
- implementation of the primal-dual interior point method
- implementation of the branch-and-bound method
- application program interface (API)
- GNU MathProg modeling language (a subset of AMPL)
- GLPSOL, a stand-alone LP/MIP solver.

It is currently developed and maintained by Andrew Makhorin, Department for Applied Informatics, Moscow Aviation Institute, Moscow, Russia.

4.2 Problem (LP-MIP) formulation

The problem formulation of GLPK is slightly different from the one of other software. For a LP problem we have:

minimize (or maximize)

$$Z = c_1 x_{m+1} + c_2 x_{m+2} + ... + c_n x_{m+n} + c_0 \tag{4.1}$$

subject to linear constraints:

$$\begin{aligned} x_1 &= a_{11} x_{m+1} + a_{12} x_{m+2} + \ldots + a_{1n} x_{m+n} \\ x_2 &= a_{21} x_{m+1} + a_{22} x_{m+2} + \ldots + a_{2n} x_{m+n} \\ &\ldots\ldots\ldots\ldots \\ x_m &= a_{m1} x_{m+1} + a_{m2} x_{m+2} + \ldots + a_{mn} x_{m+n} \end{aligned} \tag{4.2}$$

and bounds of variables:

$$\begin{aligned} l_1 \le x_1 \le u_1 \\ l_2 \le x_2 \le u_3 \\ \ldots\ldots \\ l_{m+n} \le x_{m+n} \le u_{m+n} \end{aligned} \tag{4.3}$$

where $x_1, x_2, \ldots, x_m$ are the auxiliary variables and $x_{m+1}, x_{m+2}, \ldots, x_{m+n}$ are the structural variables; Z is the objective function and $c_1, c_2, \ldots, c_m$ are the objective coefficients; you can also specify c_0, the constant term of the objective function and this is a peculiarity of this software;$a_{11}, a_{12}, \ldots, u_{mn}$ are the constraints coefficients; $u_1, u_2, \ldots, u_{m+n}$ and $l_1, l_2, \ldots, l_{m+n}$ are respectively the upper and lower bounds of variables.

Auxiliary variables are also called *rows*, because they correspond to rows of the constraints matrix; analogously, structural variables are also called *columns*, because they correspond to the 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:

$-\infty <$	x_k	$< +\infty$	Free variable
$l_k \le$	x_k	$\le +\infty$	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.

As we can think the problem is solved if the value of aux. and structural variables we fond is such that:

a) satisfy to all the linear constraints

b) are within their bounds

c) provide a smallest (largest) value of the objective function.

As to the MIP problems (some variables are additionally required to be integer), GLPK assumes that they have the same formulation as ordinary LP with the addition of integer requirement.

4.3 API Routines

Some considerations are important to understand the implementations we have done. Indeed the GLPK functions differ from CPLEX and Xpress in error handling and in array indexing.

Error handling

If some GLPK API routine detects erroneous or incorrect data passed by the application program, it sends appropriate diagnostic messages to the standard output and then abnormally terminates the application program. In most practical cases this allows to simplify programming avoiding numerous checks of return codes. Thus, in order to prevent crashing, the application program should check all data, which are suspected to be incorrect, before calling GLPK API routines. Should note that this kind of error handling is used only in cases of incorrect data passed by the application program. If, for example, the application program call some GLPK API routine to read data from an input file and these data are incorrect, the GLPK API routine reports about error in the usual way by means of return code. Unfortunately CPLEX does not work in the same way, almost all the CPLEX functions return a code which equals to 0 if the routine is successful, otherwise it is equals to the error number generated. For this reason you cannot have a perfect simulation of the CPLEX behavior (as to errors).

Array indexing

Normally all GLPK routines start array indexing from 1, not from 0 as CPLEX or Xpress. This means, for example, if some vector x of the length n is passed

as an array to some GLPK routine, the latter expects vector components to be placed in locations $x[1],\ x[2], \ldots, x[n]$, and the location $x[0]$ normally is not used.

4.4 LPX: The Problem Object

It is interesting to understand how GLPK organizes the structure of the problem objects. This is important if one wants to modify the GLPK package which is a completely open-source software.
Each problem object consists of five logical segments, which are:

- problem segment
- basis segment
- interior point segment
- MIP segment
- control parameters and statistic segment.

Problem segment

The problem segment contains original LP/MIP data, which corresponds to the problem formulation (4.1), (4.2) and (4.3). The most important elements are:

- rows
  - ordinal number (integer from 1 to m)
  - symbolic name
  - type (free, lower bound, upper bound, double bound, fixed)
  - numerical values of lower and upper bounds
  - scale factor
- columns (the same as rows)
- objective function
- constraint matrix (it is stored in column-wise and row-wise sparse formats.

Once the problem object has been created, the application program can access and modify any components in arbitrary order.

Basis segment

This segment keeps information related to a current basic solution. This information includes:

- row and columns statuses: they define which rows and columns are basic and which are non basic (independently on whether the corresponding basis is valid or not;
- basic solution statuses (include primal and dual status);
- factorization of the current basis matrix: it is used by the simplex-based solver and kept when the solver terminates the search; this feature allows efficiently reoptimizing the problem after some modifications;
- basic solution components: they include primal and dual values of all auxiliary and structural variables for the most recently obtained basic solution.

Interior point segment

It contains interior point solution components, which include the solution status, and primal and dual values of all auxiliary and structural variables.

MIP segment

It is used only for MIP problems. This segment includes:

- column kinds
- MIP solution status
- MIP solution components

Control parameters and statistic segment

This segment contains a fixed set of parameters, where each parameter has the following three attributes:

- code
- type
- current value.

Chapter 5

The interfaces

5.1 Description of the interfaces

We combined a Feasibility Pump code, written for ILOG CPLEX 8.1, with the solvers and the functionalities of GLPK and Xpress. We have done this without modifying any line of the original code, by writing an interface which captures the calls to CPLEX and redirects them to GLPK or Xpress. In other words, we have implemented all the CPLEX functions used by the FP using the functions of the others two software. Obviously some times it was not hard work (especially for Xpress which has a style very similar to CPLEX), but in many cases this required a complex mixing of the basic functions.
In many cases we obtained a perfect simulation of CPLEX, but this was not the case for some functions because of the limits of the software used. However, in our opinion the result is satisfactory because the FP code works and the performances reflect the powerful of the solvers. We can say that the interfaces are "transparent" in the sense that almost all the functions do their conversion with a small overhead.
The interface is made by two files: *interface.h*, where the error codes, the functions of CPLEX and the types of object are redefined; *ifc_imp.c* where all the functions are implemented. The language used is C, the same as the FP code. So we have used the C library of functions of Xpress and the C library of GLPK. The only thing we changed in the FP code is in *LocBra.h* where we substituted the inclusion of CPLEX interface *cplex.h* for *interface.h*. One can also rename *interface.h* and put it in the installation directory of CPLEX. Another thing we modified is the makefile, to compile also our files, but nothing else is changed.

5.2 Step-by-step interfaces development

After the thesis commission, the first thing we have done was finding and installing the programs. We downloaded GLPK from the GNU site (http://www.gnu.org). The first version we used was 4.4, the final test and considerations refer to 4.7 version. This one solves some bugs we found, for example it can reed all the mps (also the free standard) of our testbed. We installed GLPK in CYGWIN (a Linux-like environment for Windows, see http://www.cygwin.com). This let us use WindowsXP and a Linux platform at the same time; it was very useful in all the comparative tests we have done to test each function. Indeed the CPLEX version we used is for Win9X operative systems. Prof. Fischetti provided CPLEX 8.1 with a departmental license and Xpress-MP 2004b. This last software is a Win9X release too. We used the gcc compiler of Cygwin for GLPK based interface and the MinGW compiler (see http://www.mingw.org) for Xpress. The experimental FP code we used for tests was available at the beginning of August. We had no problem during the installation of the software, in particular, in GLPK package we found all the instructions to compile and install the software.

Once we had installed all the software, we studied all the manuals. We have also studied the FP code to understand which functions it needs. Only when we were sure that all the fundamental CPLEX functions used by FP were reproducible with the other two software, we started the implementation.

At the beginning we thought of using a meta-language, but this way required too much time and the results could be not satisfactory. So we decided to implement a new interface substituting the CPLEX one. In our opinion this method allows the developer to create a more powerful software, which can be easily modified and improved.

We tested all functions by using simple instances and simple programs which performed only the function under consideration. We compared the results of the same problem using CPLEX first, and then the two interfaces.

Once we implemented all the functions, we applied the interface to the FP code, modifying Locbra.h and the makefile. At this point we began to test the modified FP.

5.3 The functions

For each CPLEX function we took into consideration, we will next describe what it does and which are the limits of the conversion. One can see that the number of functions which the FP used is smaller than the number of functions we implemented, but the interface is ready if one wants to activate some experimental functions in the FP code. But it isn't the only reason, indeed the interfaces are the basis for a more general work. We have grouped the functions in this way:

1. Creating problems
2. Optimizing problems
3. Accessing LP/MIP results
4. Problem modifications
5. Accessing problem data
6. File reading/writing
7. Parameters setting and querying routines
8. General utilities

We did not give too much importance at the error code that has to be returned by some functions if something goes wrong; we have done an interface for a specific algorithm which uses CPLEX functions in the correct way and so there is no reason why an error code could be generated in some functions. It is not so if the interface is applied to a general algorithm written for CPLEX.
For each function we have written a brief description and we have described only the parameters which have a different behavior in GLPK or Xpress with respect to CPLEX. We do not want to write a manual of CPLEX, but a guide to understand which are the limits (and the capabilities) of the interfaces. So, after the description, we stressed, for each software, the point where one must

pay attention. For each function, we also wrote which solver functions it uses. Especially for GLPK, each new version modifies some functions, so one can easily find if and where they are used. All the information regarding the functions used come from [3], [4] and [5].

5.3.1 Creating problems

CPXcreateprob

Creates a problem object.

Synopsis

*CPXLPptr CPXcreateprob (CPXENVptr env, int *status_p, char *probname)*

Description The routine creates a CPLEX problem object in the CPLEX environment. The problem created is an LP minimization problem with zero constraints, zero variables, and an empty constraint matrix.
**status_p* is pointer to an integer used to return any error code produced and **probname* is a string with the name problem; if the routine is successful, **status_p* is 0.

GLPK: The interface creates a new LPX problem object; there will be no problems in creating the object (you don't need any kind of license) and so **status_p* is always 0. The functions used are:
lpx_create_prob
lpx_set_prob_name

Xpress: The routine creates a new XPRSprob problem object and sets its name. **status_p* has the *XPRScreateprob* error returned code. The routine also creates a callback function to capture all the messages generated. If one doesn't do this, no messages will be printed in Windows. The functions used are:
XPRSsetcbmessage
XPRSsetprobname

CPXcloneprob

Clone a problem object.

Synopsis

*CPXLPptr CPXcloneprob (CPXENVptr env, CPXLPptr lpx, int *status_p);*

Description This routine is used to create a new CPLEX problem object and copy all the problem data from an existing problem object to it. Solution and starting information is not copied.
**status_p* is pointer to an integer used to return any error code produced; if the routine is successful, **status_p* is 0.

GLPK: GLPK doesn't have a routine which is able to clone a problem. To solve the problem the interface reeds, the number of time it need, the input file. Of course this method isn't correct because of the changes you have done in the meantime are discarded. The function is very similar to *CPXreadcopyprob*. **status_p* is always 0. The functions used are the same of *CPXreadcopyprob*.

Xpress: Xpress has a specific function to clone a problem. New problem's name will be "cloned". Because of a new problem object is created, even in this case a callback function is defined. **status_p* has the *XPRScopyprob* error returned code. The functions used are:
XPRScreateprob
XPRScopyprob
XPRSsetcbmessage

5.3.2 Optimizing problems

CPXprimopt

Optimizes a problem using primal simplex.

Synopsis

int CPXprimopt(CPXENVptr env, CPXLPptr lpx);

Description The routine is used to find a solution to *LPX* problem using the primal simplex method algorithm.
It returns zero unless an error occurred during the optimization.

GLPK: The interface uses *lpx_simplex* function, but disables the dual simplex. In GLPK many parameters are specific of the *LPX* object so the interface sets some of them before lunching the simplex algorithm; *LPX_K_ITLIM* which is the total amount of iterations the simplex can do before terminating (you can change it setting *CPX_PARAM_ITLIM*), *LPX_K_MSGLEV* which modifies the output message level (you can choose it by *CPX_PARAM_SCRIND*), *LPX_K_PRESOL* which decides if presolver has to be used (*CPX_PARAM_PREIND* in CPLEX). *LPX_K_ITCNT* is set to 0 otherwise each time you use *lpx_simplex* the counter is incremented. The routine returns always 0. The functions used are:
lpx_set_int_parm
lpx_simplex

Xpress: First of all the routine sets some control parameters of the solver. These are *XPRS_OUTPUTLOG* (output messages), *XPRS_MAXTIME* (max time allowed for each simplex call), *XPRS_PRESOLVE* (if you want the presolver), *XPRS_LPITERLIMIT* (max number of iterations the simplex algorithm can perform). They can be set respectively through the CPLEX parameters *CPX_PARAM_SCRIND, CPX_PARAM_TILIM, CPX_PARAM_PREIND, CPX_PARAM_ITLIM*. The interface checks if the object is a minimization problem or a maximization problem and at the end the primal simplex is performed. The routine returns always 0. The functions used are:
XPRSsetintcontrol
XPRSgetdblattrib
XPRSminim, XPRSmaxim

CPXlpopt

Optimizes an LP problem letting CPLEX select the method.

Synopsis

int CPXlpopt (CPXENVptr env, CPXLPptr lpx);

Description The routine is used to find a solution to *lpx* problem using the dual simplex method unless a primal feasible basis is loaded and the advanced indicator is on. In that case, it will use the primal simplex method.
The routine returns zero unless an error occurred during the optimization.

GLPK: We always consider the *CPX_PARAM_LPMETHOD* parameter set *AUTOMATIC*. The routine is the same as *CPXprimopt*, but in this case dual simplex algorithm is allowed. The routine returns always 0. The functions used are:
lpx_set_int_parm
lpx_simplex

Xpress: The routine is the same as *CPXprimopt* but this time Xpress decides which simplex algorithm to perform.

CPXbaropt

Optimizes a problem using barrier algorithm.

Synopsis

int CPXbaropt (CPXENVptr env, CPXLPptr lpx);

Description The routine may be used to find a solution to a linear program, using the barrier algorithm.

GLPK: The interface set some parameters: *LPX_K_ITLIM, LPX_K_ITCNT* and *LPX_K_MSGLEV* (see *CPXprimopt* for the descriptions). Then the barrier al-

gorithm is applied to the the problem object. Note that the version of GLPK we used has a very poor implementation of the barrier method. For example it has no features against numerical instability. The routine returns always 0. The functions used are:
lpx_set_int_parm
lpx_interior

Xpress: Not implemented.

CPXmipopt

Optimizes a MIP problem.

Synopsis

int CPXmipopt(CPXENVptr env, CPXLPptr lpx);

Description The routine may be used to find a solution to a mixed integer program. It returns 0 unless it encounters an error.

GLPK: The interface uses the GLPK branch-and-bound solver. This solver uses simple heuristics for branching and backtracking, and therefore it is not perfect. Most probably this solver can be used for solving MIP problems with one or two hundreds of integer variables. Hard or very large scale MIP problems cannot be solved by this routine. As in the previous solver the iteration number can be limit and the counter has to be set to 0. One can change the parameters *LPX_K_TOLINT* and *LPX_K_TOLOBJ* which correspond to *CPX_PARAM_EPAGAP* and *CPX_PARAM_EPGAP*. The routine returns always 0. The functions used are:
lpx_set_int_parm
lpx_set_real_parm
lpx_integer

Xpress: Even for this solver one can set *XPRS_OUTPUTLOG* and

XPRS_MAXTIME. Then the interface initializes the global search and starts it. The routine returns the *XPRSglobal* returned code. The functions used are:
XPRSsetintcontrol
XPRSinitglobal
XPRSglobal

CPXdualopt

Optimizes a problem using dual simplex.

Synopsis

int CPXdualopt(CPXENVptr env, CPXLPptr lpx);

Description The routine may be used to find a solution to a LP problem using dual simplex algorithm. It returns 0 unless it encounters an error.

GLPK: GLPK cannot force the use of dual simplex, so the routine is the same as *CPXlpopt*.

Xpress: The routine is the same as *CPXprimopt* but this time the dual simplex algorithm is forced.

5.3.3 Accessing LP/MIP results

CPXgetstat

Accesses optimization status information.

Synopsis

int CPXgetstat (CPXENVptr env, CPXLPptr lpx);

Description The routine is used to access the solution status of the problem

after an LP or mixed integer optimization. The routine returns the solution status of the most recent optimization performed on the CPLEX problem object.

GLPK: We have implemented the routine for MIP and LP problem. CPLEX distinguishes much more cases than GLPK, so we have unified some cases. The interface takes in global variable the last solver used because GLPK uses different functions for different solvers to retrieve the solution status. The functions used are:
lpx_get_status
lpx_mip_status

Xpress: Xpress has different functions for different solvers if you want to know the solution status. The interface checks an internal variable which takes the last solver used (mipsolver or lpsolver) and than it calls the corresponding function. Not all the CPLEX cases are present in Xpress, so we have unified some of them but we think there will be no problems in future implementations. The function used is:
XPRSgetintattrib

CPXgetobjval

Accesses LP solution objective value.

Synopsis

*int CPXgetobjval (CPXENVptr env, CPXLPptr lpx, double *objval_p);*

Description The routine is used to return the LP solution objective value. The routine returns a zero on success, and a nonzero if no solution exists.

GLPK: The routine returns always 0. The function used is:
lpx_get_obj_val

Xpress: The interface assigns the *XPRS_LPOBJVAL* parameter value to **obj-*

val_p and returns 0. The function used is:
XPRSgetdblattrib

CPXgetx

Accesses optimal variable values.

Synopsis

*int CPXgetx (CPXENVptr env, CPXLPptr lpx, double *x, int begin, int end);*

Description The routine is used to access the solution values for a range of problem variables of a linear problem. The routine returns a zero on success, and a nonzero if an error occurs.

GLPK: The routine returns always 0. The function used is:
lpx_get_col_prim

Xpress: The routine returns always 0.

CPXgetpi

Accesses constraint dual values.

Synopsis

*int CPXgetpi (CPXENVptr env, CPXLPptr lpx, double *pi, int begin, int end);*

Description The routine *CPXgetpi* is used to access the dual values for a range of the constraints of a linear program. The routine returns a zero on success, and a nonzero if an error occurs.

GLPK: The routine returns always 0. The function used is:

lpx_get_row_dual

Xpress: The interface uses one of the parameters of *XPRSSgetsol*. It returns always 0. The functions used are:
XPRSgetintattrib
XPRSgetsol

CPXgetdj

Accesses variable reduced-costs.

Synopsis

*int CPXgetdj (CPXENVptr env, CPXLPptr lpx, double *dj, int begin, int end);*

Description The routine is used to access the reduced costs for a range of the variables of a linear program. The routine returns a zero on success, and a nonzero if an error occurs.

GLPK: The routine returns always 0. The function used is:
lpx_get_col_dual

Xpress: The routine is the same as the previous one but it uses another parameter of *XPRSSgetsol*.

CPXgetbase

Accesses a basis.

Synopsis

*int CPXgetbase (CPXENVptr env, CPXLPptr lpx, int *cstat, int *rstat);*

Description The routine is used to get the basis resident in a CPLEX problem object. Either of the arguments cstat or rstat may be NULL, if only one set of statuses is needed. The routine returns a zero if a basis exists. It returns zero if no solution exists or any other type of error occurs.

GLPK: There is a row/column status of GLPK basis which there isn't in CPLEX: *LPX_NS* (non basic fixed variable). The routine returns always 0. The functions used are:
lpx_get_num_cols, lpx_get_num_rows
lpx_get_col_stat, lpx_get_row_stat

Xpress: The Xpress function *XPRSSgetbasis* has the same syntax as the CPLEX one. So we have simply redirected the call. Note that it returns the *XPRSgetbasis* returned code. Obviously, the only function used is:
XPRSSgetbasis

CPXgetphase1cnt

Accesses number of Phase I simplex iterations.

Synopsis

int CPXgetphase1cnt (CPXENVptr env, CPXLPptr lpx);

Description The routine is used to access the number of Phase I iterations to solve a problem using the primal or dual simplex method. If a solution exists, the routine returns the Phase I iteration count. If no solution exists, it returns the value 0.

GLPK: It returns the value of *LPX_K_ITCNT*. The function used is:
lpx_get_int_parm

Xpress: It returns the value of *XPRS_SIMPLEXITER*. The function used is:
XPRSgetintattrib

CPXgetmipobjval

Accesses the MIP solution objective value.

Synopsis

*int CPXgetmipobjval (CPXENVptr env, CPXLPptr lpx, double *objval_p);*

Description The routine is used to access the mixed integer solution objective value. The routine returns a zero on success, and a nonzero if an error occurs.

GLPK: The routine returns always 0. The function used is:
lpx_mip_obj_val

Xpress: **objval_p* takes the value of *XPRS_MIPOBJVAL*. The routine returns always 0. In this case the function used is:
XPRSgetdblattrib

CPXgetmipx

Accesses a range of MIP variable values.

Synopsis

*int CPXgetmipx (CPXENVptr env, CPXLPptr lpx, double *x, int begin, int end);*

Description The routine is used to access a range of mixed integer solution values. The routine returns a zero on success, and a nonzero if an error occurs.

GLPK: The routine returns always 0. The function used is:
lpx_mip_col_val

Xpress: The interface uses another parameter of the *XPRSSgetsol*. The routine returns always 0. The function used is:
XPRSgetsol

CPXgetbestobjval

Accesses objective value of best remaining node.

Synopsis

*int CPXgetbestobjval (CPXENVptr env, CPXLPptr lpx, double *objval_p);*

Description The routine is used to access the objective function value of the best remaining node in the branch-and-bound tree. The routine returns a zero on success and a nonzero if an error occurs.

GLPK: This functionality isn't implemented in GLPK and so the root value is returned. All the nodes will be explored.

Xpress: The same as GLPK, so the interface returns *XPRS_LPOBJVAL* value. The function used is:
XPRSgetdblattrib

5.3.4 Problem modification

CPXaddrows

Adds constraints.

Synopsis

*int CPXaddrows (CPXENVptr env, CPXLPptr lpx, int ccnt, int rcnt, int nzcnt, double *rhs, char *sense, int *rmatbeg, int *rmatind, double *rmatval, char **colname, char **rowname);*

Description The routine adds constraints to the CPLEX problem object *lpx*. The routine returns a zero on success, and a nonzero if an error occurs.

GLPK: As in CPLEX, also in this implementation are added new columns and are set to the CPLEX default value. It always returns 0. The functions used are:
lpx_add_cols, lpx_add_rows
lpx_set_col_bnds, lpx_set_row_bnds
lpx_set_col_name, lpx_set_row_name
lpx_set_mat_row

Xpress: *XPRSSaddrows* is very similar to *CPXaddrows*. The interface, at the beginning, adds the necessary columns and sets them as the CPLEX default. New rows have default names (*colname* and *rowname* are ignored).The functions used are:
XPRSaddcols
XPRSaddrows

CPXdelsetrows

Deletes set of constraints.

Synopsis

*int CPXdelsetrows (CPXENVptr env, CPXLPptr lpx, int *delstat);*

Description The routine deletes a set of rows. After the deletion occurs, the remaining rows are indexed consecutively starting at 0, and in the same order as before the deletion. The routine returns a zero on success, and a nonzero if an error occurs.

GLPK: It always returns 0. The functions used are:
lpx_get_num_rows
lpx_del_rows

Xpress: It always returns 0. The function used is:
XPRSgetintattrib
XPRSdelrows

CPXaddcols

Adds variables.

Synopsis

*int CPXaddcols (CPXENVptr env, CPXLPptr lpx, int ccnt, int nzcnt, double *obj, int *cmatbeg, int *cmatind, double *cmatval, double *lb, double *ub, char **colname);*

Description The routine adds columns to the specified CPLEX problem object *lpx*. It returns a zero on success, and a nonzero if an error occurs.

GLPK: In this function the most important thing is the difference in the value of infinite of GLPK an CPLEX. This because of the bounds of the new variables added. If one looks the code it is easy to understand how we solved this problem after many tests. The routine always returns 0. The functions used are:
lpx_add_cols
lpx_set_col_name
lpx_set_col_bnds
lpx_set_obj_coef
lpx_set_mat_col

Xpress: *XPRSSaddcols* is very similar to the CPLEX function, but new rows have default names. The routine returns *XPRSaddcols* return code. The function used is:
XPRSSaddcols

CPXchgbds

Changes bounds.

Synopsis

*int CPXchgbds(CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, char *lu, double *bd);*

Description The routine is used to change the upper or lower bounds on a set of variables of a problem. Several bounds can be changed at once, with each bound specified by the index of the variable with which it is associated. The value of a variable can be fixed at one value by setting the upper and lower bounds to the same value. The routine returns a zero on success, and a nonzero if an error occurs.

GLPK: We have done many tests to understand which is the mechanism of CPLEX. For example if you have a fixed variable and you change the upper bound, the variable becomes dual bounded and the lower bound is the old fixed value. The code of this function is purposely redundant, so one can better understand all the cases. The routine always returns 0. The function used are:
lpx_get_col_type
lpx_set_col_bnds
lpx_get_col_lb, lpx_get_col_ub

Xpress: The routine simply redirects the call to *XPRSchgbounds*, these functions have an identical syntax. Also the returned code is of the Xpress function. The function used is:
XPRSchgbounds

CPXchgsense

Changes constraint sense.

Synopsis

*int CPXchgsense (CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, char *sense);*

Description The routine is used to change the sense of a set of constraints of a CPLEX problem object. The routine returns a zero on success, and a nonzero if an error occurs.

GLPK: It always returns 0. The function used is:
lpx_set_row_bnds

Xpress: Also this function is a simply redirection, in this case to *XPRSSchgrowtype*. Also the returned code is of the Xpress function. The function used is:
XPRSchgrowtype.

CPXchgobj

Changes coefficients in objective function.

Synopsis

*int CPXchgobj (CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, double *values);*

Description The routine is used to change the objective coefficients of a set of variables in a CPLEX problem object. The routine returns a zero on success, and a nonzero if an error occurs.

GLPK: It always returns 0. The function used is:
lpx_set_obj_coef

Xpress: This is another redirection. The routine returns *XPRSchgobj* returned code. The function used is:
XPRSchgobj

CPXchgrhs

Changes coefficients in right-hand side.

Synopsis

*int CPXchgrhs (CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, double *values);*

Description The routine is used to change the right-hand side coefficients of a set of constraints in the CPLEX problem object. The routine returns a zero on success, and a nonzero if an error occurs.

GLPK: In this function one can see the difference between the model formulation of CPLEX and of GLPK. Indeed the GLPK constraints are formulated in this way:

$$x_p = a_{p1}x_{p+1} + a_{p2}x_{p+2} + \ldots + a_{pn}x_{p+n},$$

$$l_p \leq x_p \leq u_p$$

one can modify l_p and u_p with l*px_set_row_bnds*; CPLEX is different, the formulation is:

$$a_{p1}x_{p+1} + a_{p2}x_{p+2} + \ldots + a_{pn}x_{p+n} \lesseqgtr rhs_p$$

and you can modify rhs_p with the function we know. As a consequence, we have to see the kind of constraint (*LPX_UP, LPX_LO, LPX_DB, LPX_FX*) before changing one or two of its bounds. If the constraint is of dual-bounded type, we fix its value to the new rhs one. The routine always returns 0. The functions it uses are:
lpx_get_row_type
lpx_set_row_bnds

Xpress: The routine returns *XPRSchgrhs* return code. The function used is:

XPRSchgrhs

CPXprobtype

Changes problem type.

Synopsis

int CPXchgprobtype (CPXENVptr env, CPXLPptr lpx, int type);

Description The routine is used to change the current problem to a related problem type.

GLPK: Only LP (*CPXPROB_LP*) and MIP (*CPXPROB_MILP*) problems are supported. It always returns 0. The function used is:
lpx_set_class

Xpress: It isn't necessary for Xpress, you can use MIP and LP functions in a parallel way. So the interface doesn't do anything.

5.3.5 Accessing problem data

CPXgetprobname

Accesses name of problem.

Synopsis

*int CPXgetprobname (CPXENVptr env, CPXLPptr lpx, char *buf_str, int bufspace, int *surplus_p);*

Description The routine is used to access the name of the problem.

GLPK: The functions *bufspace* and *surplus_p* are well implemented. The routine

returns *CPXERR_NEGATIVE_SURPLUS* if name is larger than buffer. Otherwise 0 is returned. The function used is:
lpx_get_prob_name

Xpress: Also in this interface *bufspace* and *surplus_p* are well implemented. There is a Xpress limitation, the characters allowed are 200.
CPXERR_NEGATIVE_SURPLUS will be returned if the space isn't enough. The function used is:
XPRSgetprobname

CPXgetcolname

Accesses variable (column) names.

Synopsis

*int CPXgetcolname (CPXENVptr env, CPXLPptr lpx, char **name, char *namestore, int storespace, int *surplus_p, int begin, int end);*

Description The routine is used to access a range of column names or, equivalently, the variable names of a CPLEX problem object.

GLPK: As in the previous one, also in this case the functions *storespace* and *surplus_p* are well implemented and *CPXERR_NEGATIVE_SURPLUS* is returned if *storespace* is too small. The function used is:
lpx_get_col_name

Xpress: The interface doesn't do anything.

CPXgetnumcols

Accesses number of columns in problem.

Synopsis

int CPXgetnumcols (CPXENVptr env, CPXLPptr lpx);

Description The routine is used to access the number of columns in the constraint matrix, or equivalently, the number of variables in the CPLEX problem object.

GLPK: No problems. The function used is:
lpx_get_num_cols

Xpress: The number of columns in Xpress is an attrib (*XPRS_COLS*) an so the function used is:
XPRSgetintattrib

CPXgetnumrows

Accesses number of rows in problem.

Synopsis

int CPXgetnumrows (CPXENVptr env, CPXLPptr lpx);

Description The routine is used to access the number of rows in the constraint matrix, or equivalently, the number of constraints in the CPLEX problem object.

GLPK: No problems. The function used is:
lpx_get_num_cols

Xpress: The number of columns in Xpress is an attrib (*XPRS_ROWS*) an so the function used is:
XPRSgetintattrib

CPXgetobjsen

Accesses objective sense.

Synopsis

int CPXgetobjsen (CPXENVptr env, CPXLPptr lpx);

Description The routine is used to access whether the objective function sense of a CPLEX problem object is maximization or minimization.

GLPK: No problems. The function used is:
lpx_get_obj_dir

Xpress: Also the objective function sense is an attrib (*XPRS_OBJSENSE*) and so the function used is:
XPRSgetdblattrib

CPXgetobj

Accesses objective coefficient values sense.

Synopsis

*int CPXgetobj (CPXENVptr env, CPXLPptr lpx, double *obj, int begin, int end);*

Description The routine is used to access a range of objective function coefficients of a CPLEX problem object. The routine returns a zero on success, and a nonzero if an error occurs.

GLPK: It always returns 0. The functions used is:
lpx_get_obj_coef

Xpress: The routine returns *XPRSgetobj* return code. The function used is:
XPRSgetobj

CPXgetrhs

Accesses right-hand side values sense.

Synopsis

*int CPXgetrhs (CPXENVptr env, CPXLPptr lpx, double *rhs, int begin, int end);*

Description The routine is used to access the right-hand side coefficients for a range of constraints in a CPLEX problem object. The routine returns a zero on success, and a nonzero if an error occurs.

GLPK: In this function we found the same problem of *CPXchgrhs*; also in this case the interface controls the kind of constraint and then it returns the bound required. Another problem is the definition of infinite, but it isn't so important because if rhs_i is infinite the constraint is negligible. The functions used are:
lpx_get_row_type
lpx_get_row_ub, lpx_get_row_lb

Xpress: *XPRSSgetrhs* has the same syntax, the interface uses that function, also for the returned code. So the function used is:
XPRSSgetrhs

CPXgetsense

Accesses constraint senses.

Synopsis

*int CPXgetsense (CPXENVptr env, CPXLPptr lpx, char *sense, int begin, int end);*

Description The routine is used to access the sense for a range of constraints in a CPLEX problem object. The routine returns a zero on success, and a nonzero if an error occurs.

GLPK: Also dual-bounded constraints, with upper bound equals to lower bound, are defined fixed constraints. The routine always returns 0. The functions used are:
lpx_get_row_type
lpx_get_row_ub, lpx_get_row_lb

Xpress: The routine uses *XPRSgetrowtype* and returns its code. The function used is:
XPRSgetrowtype

CPXgetub

Accesses a range of upper bounds sense.

Synopsis

*int CPXgetub (CPXENVptr env, CPXLPptr lpx, double *ub, int begin, int end);*

Description The routine is used to access a range of upper bounds on the variables of a CPLEX problem object. The routine returns a zero on success, and a nonzero if an error occurs.

GLPK: Note that if the upper bound is larger than the *CPX_INFBOUND* parameter, than the interface returns *CPX_INFBOUND*: we think that it is better if one uses the interface with a generic program written for CPLEX. The routine always returns 0. The functions used are:
lpx_get_col_type
lpx_get_col_ub

Xpress: The routine returns *XPRSgetub* return code. The function used is:

XPRSgetub

CPXgetrows

Accesses a range of rows sense.

Synopsis

*int CPXgetrows (CPXENVptr env, CPXLPptr lpx, int *nzcnt, int *rmatbeg, int *rmatind, double *rmatval, int rmatspace, int *surplus, int begin, int end);*

Description The routine is used to access a range of rows of the constraint matrix, not including the objective function or the bounds constraints on the variables of a CPLEX problem object.

GLPK: *surplus* is well implemented. The routine always returns 0. The functions used are:
lpx_get_num_cols
lpx_get_mat_row

Xpress: The CPLEX function is very similar to *XPRSgetrows* but we have done some changes to implement *surplus*.
The function used is:
XPRSgetrows

CPXgetctype

Accesses a priority order.

Synopsis

*int CPXgetctype (CPXENVptr env, CPXLPptr lpx, char *xctype, int begin, int end);*

Description The routine is used to access the types for a range of variables in a problem object. The routine returns a zero on success, and a nonzero if an error occurs.

GLPK: The interface defines binary also the integer variables with lower bound equals to 0 and upper bounds equals to 1. The routine always returns 0. The functions used are:
lpx_get_col_kind
lpx_get_col_lb, lpx_get_col_ub

Xpress: The routine creates the *xctype* vector and, as for GLPK, defines binary also the integer variables with lower bound equals to 0 and upper bounds equals to 1. The routine always returns 0. The functions used are:
XPRSgetcoltype
XPRSgetub, XPRSgetlb

5.3.6 File reading/writing

CPXreadcopyprob

Reads and copies a problem in LP or MPS format.

Synopsis

*int CPXreadcopyprob (CPXENVptr env, CPXLPptr prob, char *filename_str, char *filetype_str);*

Description The routine reads an MPS, LP or SAV file into an existing CPLEX problem object. The type of the file may be specified with the filetype argument. When the filetype argument is NULL, the end of the file name is checked for one of the strings .lp, .lp.gz, .lp.z, .mps, mps.gz, .mps.z, .sav, .sav.gz, or .sav.z. If one of these strings is present, filetype is set accordingly. If filetype is NULL and none of these strings is at the end of the file name, the routine automatically detects the type of the file by examining the first few bytes of the file. The routine

returns a zero on success, and a nonzero if an error occurs.

GLPK: GLPK recognizes only LP and MPS formats. If filetype argument is NULL, the interface checks the filename for one of the strings .lp, .LP, .mps, .MPS. If none of these strings is present, the interface cannot detect the kind of file. The interface accepts free mps format. The functions used are:
lpx_read_cpxlp
lpx_read_freemps

Xpress: Xpress recognizes only LP and MPS format; the routine returns *XPRSreadprob* return code and recognizes the kind of problem. The function used is:
XPRSreadprob

CPXwriteprob

Writes a problem file in any format.

Synopsis

*int CPXwriteprob (CPXENVptr env, CPXLPptr lpx, char *filename, char *filetype);*

Description The routine is used to write the CPLEX problem object to a file in a specified format. The routine returns a zero on success, and a nonzero if an error occurs.

GLPK: The interface recognizes only LP and MPS format (filetype). The functions used are:
lpx_write_cpxlp
lpx_write_freemps

Xpress: Xpress writes only in LP or MPS format. The function used is:
XPRSwriteprob

5.3.7 Parameters setting and querying routines

CPXsetdblparam (CPXsetintparam)

Changes a parameter of type double (integer).

Synopsis

int CPXsetdblparam (CPXENVptr env, int whichparam, double newvalue);
int CPXsetintparam (CPXENVptr env, int whichparam, int newvalue);

DescriptionThe routine sets the value of a CPLEX parameter (*whichparam*) of type double (integer). Only a few parameters are compatible with the CPLEX one.

GLPK: The recognized parameters are:
CPX_PARAM_SCRIND,
CPX_PARAM_ITLIM,
CPX_PARAM_PREIND,
CPX_PARAM_EPAGAP,
CPX_PARAM_EPGAP.

Xpress: The recognized parameters are:
CPX_PARAM_PREIND,
CPX_PARAM_SCRIND,
CPX_PARAM_ITLIM,
CPX_PARAM_TILIM.

5.3.8 General utilities

CPXopenCPLEX

Initializes CPLEX environment.

Synopsis

*CPXENVptr CPXopenCPLEX (int *status_p);*

Description The routine initializes a CPLEX environment when accessing a license for CPLEX and works only if the computer is licensed for Callable Library use. The routine CPXopenCPLEX() must be the first CPLEX routine called. The routine returns a pointer to a CPLEX environment.

GLPK: GLPK is free so the programs based on its library run even if one doesn't have the license. This is the always the first routine called so we used it to initialize some parameters to CPLEX default values. This routine doesn't need any GLPK function.

Xpress: Xpress finds the license in a different way, so this function returns a pointer even if the solver isn't available. We used this routine to initialize some parameters to CPLEX default values. status_p has the *XPRSinit* returned code. The function used is:
XPRSinit

CPXcloseCPLEX

Closes CPLEX environment.

Synopsis

*int CPXcloseCPLEX (CPXENVptr *env_p);*

DescriptionThe routine closes the CPLEX environment. The routine returns a zero on success, and a nonzero if an error occurs.

GLPK: It doesn't do anything.

Xpress: The routine releases the Xpress license. The function used is:
XPRSfree

CPXflushstdchannels

Causes CPLEX default channels to be flushed.

Synopsis

int CPXflushstdchannels (CPXENVptr env);

DescriptionThis routine flushes the output buffers of the four standard channels *cpxresults, cpxwarning, cpxerror, and cpxlog.* Use this routine where it is important to see all of the output created by CPLEX either on the screen or in a disk file without calling *CPXflushchannel* for each of the four channels.

GLPK: It doesn't do anything.

Xpress: It doesn't do anything.

Chapter 6

Test

6.1 Testbed

Our testbed is made by 44 0-1 MIP instances collected in MIPLIB 2003 [6] and described in Table 6.1, plus an additional set of 38 hard 0-1 MIP, described in Table 6.2, provided by Fischetti and Lodi. The two tables report the instance names and the corresponding number of variables (n), of 0-1 variables ($|\mathcal{I}|$) and of constraints (m).

Name	n	$\lvert\mathcal{I}\rvert$	m	Name	n	$\lvert\mathcal{I}\rvert$	m
10teams	2025	1800	230	mod011	10958	96	4480
A1C1S1	3648	192	3312	modglob	422	98	291
aflow30a	842	421	479	momentum1	5174	2349	42680
aflow40b	2728	1364	1442	net12	14115	1603	14021
air04	8904	8904	823	nsrand_ipx	6621	6620	735
air05	7195	7195	426	nw04	87482	87482	36
cap6000	6000	6000	2176	opt1217	769	768	64
dano3mip	13873	552	3202	p2756	2756	2756	755
danoint	521	56	664	pk1	86	55	45
ds	67732	67732	656	pp08a	240	64	136
fast0507	63009	63009	507	pp08aCUTS	240	64	246
fiber	1298	1254	363	protfold	1835	1835	2112
fixnet6	878	378	478	qiu	840	48	1192
glass4	322	302	396	rd-rplusc-21	622	457	125899
harp2	2993	2993	112	set1ch	712	240	492
liu	1156	1089	2178	seymour	1372	1372	4944
markshare1	62	50	6	sp97ar	14101	14101	1761
markshare2	74	60	7	swath	6805	6724	884
mas74	151	150	13	t1717	73885	73885	551
mas76	151	150	12	tr12-30	1080	360	750
misc07	260	259	212	van	12481	192	27331
mkc	5325	5323	3411	vpm2	378	168	234

Table 6.1: The 44 0-1 MIP instances collected in MIPLIB 2003 .

Name	n	$\|\mathcal{I}\|$	m	Name	n	$\|\mathcal{I}\|$	m
biella1	7328	6110	1203	blp-ar98	16021	15806	1128
dg012142	2080	640	6310	blp-ic97	9845	9753	923
dc1c	10039	8380	1649	blp-ic98	12640	13550	717
dc1l	37297	35638	1653	blp-ir98	6097	6031	486
dolom1	11612	9720	1803	CMS750_4	11697	7196	16381
siena1	13741	1775	2220	berlin_5_8_0	1083	794	1532
trento1	7687	6415	1265	railway_8_1_0	1796	1177	2527
rail507	63019	63009	509	usAbbrv.8.25_70	2312	1681	3291
rail2536c	15293	15284	2539	manpower1	10565	10564	25199
rail2586c	13226	13215	2589	manpower2	10009	10008	23881
rail4284c	21714	21705	4284	manpower3	10009	10008	23915
rail4872	24656	24645	4875	manpower3a	10009	10008	23865
A2C1S1	3648	192	3312	manpower4	10009	10008	23914
B1C1S1	3872	288	3904	manpower4a	10009	10008	23866
B2C1S1	3872	288	3904	ljb2	771	681	1482
sp97ic	12497	12497	1033	ljb7	4163	3920	8133
sp98ar	15085	15085	1435	ljb9	4721	4460	9231
sp98ic	10894	10894	825	ljb10	5496	5196	10742
bg51242	792	240	1307	ljb12	4913	4633	9596

Table 6.2: The additional set of 38 0-1 MIP instances.

6.2 Test description

The results of the original FP implementation based on CPLEX are reported in Table 6.3 and 6.4, with a comparison with the two FP interfaced with GLPK and Xpress.
The software releases used are ILOG CPLEX 8.1, GLPK 4.7 and Dash Xpress-MP 2004b.
The focus of this experiment was to measure the capability of the three FP implementations to converge to an initial feasible solution. The three algorithms were stopped as soon as the first feasible solution was found. We have compared the quality of the first solution found, the number of iterations and the time the algorithms needed. Computing times are expressed in CPU seconds and refer to an AMD AthlonXP 2200+ (1800MHz) with 768MByte of RAM.
We said that the FP we used is an experimental code, it has many functions and capabilities that one can enable changing a file named *param.txt*, uncommenting some lines in *LocBra.h* and using different command strings. Now we explain in detail the configuration we used. In *Locbra.h* we have enabled the line $\#define\ OPT$. This function allows the solver to perform all the simplex iterations it needs, without imposing a time limit or an iteration limit. This method attempts to obtain, at each FP iteration, the optimal solution of the LP relaxation. This setting is very sensible to the capabilities of the solver used. Indeed, if in a simplex session a stall situation occurs and the solver doesn't notice it, the FP algorithm will enter an infinite loop. This situation happened with some

difficult instance (only with GLPK solver). The *param.txt* used is:

```
1 0 0 50 10 0 0 25 100 5000.5 .002 1.5 -1 250 5000 .2 0 0 250 0

     pre  -> presolve: 1 yes, 0 no");
heurFreq  -> heuristic frequency");
    prec  -> precision: 0 = default; 1 = 1e-12");
  mipInt  -> print MIP interval");
 timeInt  -> number of time intervals");
   video  -> video: YES = 1, NO = 0");
     emp  -> emphasis: default = 0);
      IT  -> RINS/DIST every ... iterations
   EVERY  -> restart every ... iterations
   times  -> initial worsening of the bound allowed
  soglia  -> threshold for flipping
    divv  -> to devide the gap at each improvement
 addTime  -> additional time after the first solution (< 0 is disabled)
   nodes  -> number of B&B nodes for the RINS/DIST approach
   itLim  -> simplex iteration limit (used #ifndef OPT)
   pdgap  -> primal-dual gap for baropt (used #ifdef BAR)
     rep  -> number of loops of itLim iterations for the first LP
                                                  (used #ifndef OPT)
      SM  -> simplex method: 0 = CPLEX decides; P = primal; D = dual
                                                   (used #ifdef OPT)
 addIter  -> number of additional iteration after the first sol has been found
             (SAT must be set to 0 to be active)
   WRITE  -> 1 any feasible solution encountered is written in file
                                           'solution.mst', 0 otherwise
```

As one can see, we activated the presolver and disabled the heuristics. We also disabled the RINS/DIST method. We also let the solver decide the simplex method to use. All these choices have the target of taking the algorithms "problem independent". However, some knowledge of the type of instance to be solved can improve the FP performance considerably, especially for highly degenerate cases.
As to the string command, we used:

FP 2500 20 100000 0 0 2 *problem.mps*

$FPglpk$ 2500 20 100000 0 0 2 *problem.mps*

$FPxpress$ 2500 20 100000 0 0 2 *problem.mps*

Note that for all the instances we chosed an *mps* version. This because GLPK supports this format in a better way. Indeed, if one uses the LP format, GLPK takes some time for conversion. The first parameter (2500) is the max time (in CPU seconds) allowed to find the first integer feasible solution. When this time is reached the algorithm terminates in any case. The second one is the T parameter discussed in Chapter 1. Value 100000 is the maximum number of FP iterations allowed. This is a very large number as, it isn't important if the algorithm requires several iterations to terminate. The subsequent parameter is important if one wants to use enumeration, but we did not enable this function. The next one decides if RINS (1) or DISTANCE (2) heuristic method can be used; as we said, these functions are disabled but the two interfaces give the possibility of using them. The last one parameter (before the problem file name) stops the algorithm at the first feasible solution found. All the test results are in Table 6.3 and 6.4. If the algorithm terminated because of the time limit, we report the FP iterations performed. If stalling is detected during a simplex session, we wrote "stall". We wrote "error" if an error was generated and the program terminated. We wrote "iterations" if the program terminated due to iteration limit.

6.3 Computational Results

As one can see, the time needed and sometime the first feasible solution value obtained with the original FP are different with respect to the results reported in [1]: this because we have used a different PC and an updated version of the FP algorithm. Moreover, we don't know which is the FP and ILOG-CPLEX tuning used for tests in the article.
Our first order of business here was to evaluate the percentage of success in finding a feasible MIP solution (with a time limit of 2500s). In this respect, the original FP was unsuccessful only in 3 cases. As to the FP with GLPK (FPglpk), in 9 cases there was a stall, and in 3 cases FPglpk was unsuccessful. There are also 11 critical instances in which the time limit stopped the search (in particular in all the `manpower` and in many `rail` problems); in these last cases, the GLPK simplex solver needed much time for each iteration but without stalling or instability. So FPglpk was successful in 59 cases. As to the FP with Xpress (FPxpress), the percentage of success is very good, indeed only in 5 cases the program failed and only in 2 cases due to time limit. None of the tests caused a stall situation.

Talking about speed, tables 6.5 and 6.6 report the ratio of the average times for iteration between FPglpk (FPxpress) and the original FP (the higher is the ratio, the slower is the solver). Even if an FP implementation did not find a feasible value, we reported the ratio considering the iterations performed before the time limit exceeding (not in case of stalling). If the total time is lower than 1 second for all the problems we wrote "too low" because the ratio is not meaningful. The average ratio of FPglpk is 71.2, a very large value which is negatively influenced by two instances: `CMS750_4` and `swath`. Indeed, with this instances, FPglpk was respectively 712.45 and 840.34 times slower than the CPLEX version. If we not consider these problems which penalizes too much the solver, the average ratio becomes 45.25. The results in the tables show us that FPglpk was very slow also in the `manpower` instances: from 181.08 to 411.84 times slower (if we not consider also these problems, the average ratio is 14.31). There are instances in which FPglpk obtained a very good result, for example in `air05` or `cap6000`; with `net12` FPglpk obtained more or less the same time per iteration as the original FP.
As to FPxpress, the average ratio is 2.52. As one can see from the tables, there are no critical cases in which the ratio is very high as in the previous implementation. We obtained the worst results with the `manpower` instances (from 2.44 to 12.14 times slower), but with many problems FPxpress was faster than the CPLEX version: first of all `net12` and then all the `ljb` instances.

As to the solution quality and the number of iterations performed, there is not a clear winner. Indeed, the results are similar in all cases, as the framework is the same. There are some noticeable cases, for example the quality of the first solution found with FPxpress of `dano3mip` and `mas76` is very good. One must take into consideration the random choices of the framework used. With `danoint` instance the original FP found a way which needed only 3 iterations to terminate (191 FPglpk and 81 FPxpress!). Another example is `harp2`: FPglpk found the best solution with 344 iterations while the original FP required 654 iterations and FPxpress 977. None of the FP implementations has found a feasible solution for `ds`, `rd-rplusc-21` and `p2756`. This last problem is a pathological case for FP (see [1] for an explanation).

	ILOG CPLEX			GLPK			Dash Xpress-MP		
Name	value	nIT	time	value	nIT	time	value	nIT	time
10teams	992.00	53	10.13	970.00	177	121.53	1,076.00	173	44.64
A1C1S1	18,377.24	5	8.53	19,153.53	5	46.53	16,257.50	5	28.13
aflow30a	4,398.00	16	0.16	3,325.00	3	0.22	5,123.00	17	0.42
aflow40b	6,859.00	7	0.73	4,562.00	5	2.22	5,210.00	7	1.34
air04	58,950.00	6	112.33	59,987.00	7	647.81	58,278	4	21.47
air05	29,937.00	2	15.30	31,899.00	8	110.00	32,120.00	4	11.48
cap6000	-2,354,320.00	2	0.80	-2,354,320.00	2	1.58	-2,354,320.00	2	0.34
dano3mip	2,882,022.45	2	118.83	stall	1	2500.00	756.62	2	179.55
danoint	77.00	3	0.23	73.33	191	52.62	74.50	81	8.95
ds	-	112	2500.00	error	-	-	-	18	2500.00
fast0507	181.00	4	51.91	190.00	2	586.56	186.00	4	84.2
fiber	1,911,617.79	2	0.03	1,324,277.64	2	0.11	1,686,325.48	2	0.05
fixnet6	9,131.00	4	0.03	8,542.00	4	0.14	11,261.00	4	0.09
glass4	5,600,050,250.00	124	0.34	4,650,041,100.00	3	0.05	3,691,696,333.33	0	0.01
harp2	-43,856,974.00	654	5.36	-38,231,320.00	344	16.33	-49,695,263.00	977	19.31
liu	6,262.00	0	0.06	6,450.00	0	0.58	5,384.00	0	0.06
markshare1	1,114.00	9	0.02	165.00	10	0.02	851.00	1	0.00
markshare2	1,738.00	0	8.00	182.00	3	0.02	2,546.00	1	0.00
mas74	52,429,700.59	1	0.00	14,372.87	1	0.02	19,524.00	1	0.02
mas76	194,527,859.06	1	0.00	43,744.26	1	0.02	48,566.00	1	0.00
misck07	3,700.00	29	0.17	4,385.00	149	8.00	4,200.00	38	1.27
mkc	-164.56	2	0.36	-45.85	3	1.22	-126.40	3	0.44
mod011	0.00	0	0.23	0.00	0	2.51	0.00	0	1.22
modglob	35,147,088.88	0	0.02	35,147,088.88	0	0.02	35,147,088.88	0	0.02
momentum1	462,127.33	502	1870.02	-	20	2500.00	436,506.00	228	1799.00
net12	337.00	3	82.19	337.00	12	326.80	337.00	84	205.77
nsrand_ipx	340,800.00	3	0.92	339,200.00	3	4.81	384,800.00	3	1.38
nw04	19,882.00	1	3.36	19,792.00	1	16.25	19,792.00	1	7.45
opt1217	-12.00	0	0.02	-16.00	0	0.03	0.00	1	0.03
p2756	iterations	-	-	iterations	-	-	iterations	-	-
pk1	57.00	1	0.02	68.00	0	0.00	69.00	0	0.00
pp08a	11,150.00	2	0.00	12,430.00	3	0.05	11,870.00	3	0.02
pp08aCUTS	10,940.00	2	0.02	11,740.00	3	0.09	12,690.00	3	0.05
protfold	-10.00	367	1104.67	num. instability	-	-	-13.00	216	926.94
qiu	389.36	3	0.45	918.72	2	1.34	688.98	3	0.69
rd-rplusc-21	-	115	2500.00	-	6	2500.00	-	14	2500.00
set1ch	76,951.50	2	0.03	98,147.25	2	0.14	87,866.25	2	0.06
seymur	452.00	9	6.27	457.00	5	215.88	443.00	6	9.83
sp97ar	1,398,705,728.00	6	6.03	stall	1	2500.00	1,657,416,281.60	6	7.98
swath	19,221.42	49	3.56	39,212.87	3	183.16	42,665.45	296	139.78
t1717	826,848.00	42	1459.30	-	7	2500.00	225,200.00	22	2163.81
tr12-30	277,218.00	9	0.13	248,684.00	8	1.88	279,714.00	7	0.45
van	8.21	4	326.03	7.37	4	931.53	6.41	7	1038.00
vpm2	19.25	3	0.02	17.75	2	0.05	21.00	2	0.02

Table 6.3: FP performance comparison with different solvers

	ILOG CPLEX			GLPK			Dash Xpress-MP		
Name	value	nIT	time	value	nIT	time	value	nIT	time
biella1	3,537,959.54	5	14.41	13,678,241.92	6	135.78	3,269,058.80	5	23.83
dc1c	27,348,312.19	4	30.32	21,986,395.78	4	350.83	17,044,961.00	6	41.95
dc1l	8,256,022.49	5	153.94	stall	1	2500.00	76,686,594.47	5	164.53
dolom1	298,684,615.17	7	48.27	514,489,757.13	4	743.30	186,950,753.06	8	70.00
siena1	104,004,996.99	5	138.30	stall	2	2500.00	-	6	208.58
trento1	356,179,003.01	2	29.22	2,771,711,194.07	2	280.38	347,208,438.00	2	45.66
rail507	178.00	2	65.23	197.00	3	959.61	860,986.05	3	91.67
rail2536c	715.00	4	44.22	stall	2	2500.00	714.00	4	98.41
rail2586c	1,007.00	5	129.22	stall	2	2500.00	1,004.00	5	181.66
rail4284c	1,124.00	3	1623.05	stall	1	2500.00	1,119.00	5	515.19
rail4872	1,614.00	5	467.56	stall	1	2500.00	1,622.00	5	564.56
A2C1S1	19,879.93	5	8.69	19,883.00	5	40.84	19,528.00	5	21.67
B1C1S1	38,530.65	7	11.58	stall	2	2500.00	42,489.19	7	32.84
B2C1S1	48,279.95	6	9.88	59,217.00	6	78.53	46,198.04	6	33.08
sp97ic	1,280,793,707.52	3	3.84	314,809,721.12	4	21.98	1,073,817,106.08	4	5.61
sp98ar	988,402,511.00	4	6.17	1,050,813,686.88	4	55.25	1,136,153,068.32	3	6.84
sp98ic	959,924,716.00	3	3.08	107,765,747.52	4	18.48	75,902,608.00	4	4.59
blp-ar98	25,094.03	161	30.63	-	498	2500.00	22,372.94	1279	670.03
blp-ic97	7,874.87	4	0.97	7,618.51	7	11.58	7,423.29	11	3.83
blp-ic98	14,848.96	6	2.09	14,822.98	8	17.81	13,707.62	5	3.56
blp-ir98	5,388.84	3	0.56	6,573.28	4	3.38	5,277.00	4	1.05
CMS750_4	606.00	131	28.73	-	16	2500.00	6.23	128	113.69
berlin_5_8_0	79.00	10	0.19	79.00	11	5.95	79.00	14	0.45
railway_8_1_0	440.00	13	0.38	436.00	13	22.89	441.00	16	1.06
usAbbrv.8.25_70	164.00	34	0.95	162.00	28	67.17	error		
bg512142	120,738,665.00	0	0.20	120,690,438.67	0	0.47	120,738,655.00	0	0.48
dg012142	153,406,945.50	0	1.75	140,064,121.25	0	5.03	153,406,921.50	0	2.95
manpower1	8.00	66	58.86	-	11	2500.00	10.00	55	120.09
manpower2	7.00	148	240.17	-	7	2500.00	8.00	30	453.67
manpower3	6.00	49	84.56	-	8	2500.00	6.00	49	622.39
manpower3a	6.00	73	99.86	-	5	2500.00	7.00	65	848.77
manpower4	7.00	192	166.50	-	7	2500.00	7.00	77	810.47
manpower4a	7.00	53	121.83	-	7	2500.00	9.00	119	1349.39
ljb2	7.31	0	1.72	7.31	0	0.40	7.34	0	0.03
ljb7	8.61	0	0.06	8.61	0	27.83	8.61	0	0.50
ljb9	9.48	0	1.23	9.48	0	37.44	9.48	0	0.59
ljb10	7.31	0	1.72	7.31	0	52.44	7.31	0	0.81
ljb12	6.20	0	1.20	6.20	0	42.33	6.20	0	0.63

Table 6.4: FP performance comparison with different solvers (cont.d)

	ILOG CPLEX		GLPK		Dash Xpress-MP	
Name	value	ratio	value	ratio	value	ratio
10teams	992.00	1.00	970.00	3.59	1,076.00	1.35
A1C1S1	18,377.24	1.00	19,153.53	5.45	16,257.50	3.29
aflow30a	4,398.00	1.00	3,325.00	7.33	5,123.00	2.47
aflow40b	6,859.00	1.00	4,562.00	4.25	5,210.00	1.83
air04	58,950.00	1.00	59,987.00	4.94	58,278	0.29
air05	29,937.00	1.00	31,899.00	1.79	32,120.00	0.37
cap6000	-2,354,320.00	1.00	-2,354,320.00	1.97	-2,354,320.00	0.42
dano3mip	2,882,022.45	1.00	stall	-	756.62	1.51
danoint	77.00	1.00	73.33	3.59	74.50	1.44
ds	-	1.00	error	-	-	6.22
fast0507	181.00	1.00	190.00	22.60	186.00	1.62
fiber	1,911,617.79	1.00	1,324,277.64	3.67	1,686,325.48	1.67
fixnet6	9,131.00	1.00	8,542.00	4.66	11,261.00	3.00
glass4	5,600,050,250.00	1.00	4,650,041,100.00	too low	3,691,696,333.33	too low
harp2	-43,856,974.00	1.00	-38,231,320.00	5.79	-49,695,263.00	2.41
liu	6,262.00	1.00	6,450.00	too low	5,384.00	too low
markshare1	1,114.00	1.00	165.00	too low	851.00	too low
markshare2	1,738.00	1.00	182.00	too low	2,546.00	too low
mas74	52,429,700.59	1.00	14,372.87	too low	19,524.00	too low
mas76	194,527,859.06	1.00	43,744.26	too low	48,566.00	too low
misck07	3,700.00	1.00	4,385.00	9.15	4,200.00	5.70
mkc	-164.56	1.00	-45.85	2.25	-126.40	0.81
mod011	0.00	1.00	0.00	10.91	0.00	5.30
modglob	35,147,088.88	1.00	35,147,088.88	too low	35,147,088.88	too low
momentum1	462,127.33	1.00	-	33.56	436,506.00	2.11
net12	337.00	1.00	337.00	0.99	337.00	0.09
nsrand_ipx	340,800.00	1.00	339,200.00	5.22	384,800.00	1.50
nw04	19,882.00	1.00	19,792.00	4.84	19,792.00	2.22
opt1217	-12.00	1.00	-16.00	too low	0.00	too low
p2756	iterations	1.00	iterations	-	iterations	-
pk1	57.00	1.00	68.00	too low	69.00	too low
pp08a	11,150.00	1.00	12,430.00	too low	11,870.00	too low
pp08aCUTS	10,940.00	1.00	11,740.00	too low	12,690.00	too low
protfold	-10.00	1.00	num. instability	-	-13.00	1.43
qiu	389.36	1.00	918.72	4.47	688.98	1.53
rd-rplusc-21	-	1.00	-	19.17	-	8.21
set1ch	76,951.50	1.00	98,147.25	too low	87,866.25	too low
seymur	452.00	1.00	457.00	61.97	443.00	2.35
sp97ar	1,398,705,728.00	1.00	stall	-	1,657,416,281.60	1.32
swath	19,221.42	1.00	39,212.87	840.34	42,665.45	6.50
t1717	826,848.00	1.00	-	10.28	225,200.00	2.83
tr12-30	277,218.00	1.00	248,684.00	too low	279,714.00	too low
van	8.21	1.00	7.37	2.85	6.41	1.82
vpm2	19.25	1.00	17.75	too low	21.00	too low

Table 6.5: Average time per iteration (ratio with respect to FPcplex)

	ILOG CPLEX		GLPK		Dash Xpress-MP	
Name	value	value	ratio	value	ratio	
biella1	3,537,959.54	1.00	13,678,241.92	7.85	3,269,058.80	1.65
dc1c	27,348,312.19	1.00	21,986,395.78	11.57	17,044,961.00	0.92
dc1l	8,256,022.49	1.00	stall	-	76,686,594.47	1.07
dolom1	298,684,615.17	1.00	514,489,757.13	26.95	186,950,753.06	1.26
siena1	104,004,996.99	1.00	stall	-	-	1.25
trento1	356,179,003.01	1.00	2,771,711,194.07	9.60	347,208,438.00	1.56
rail507	178.00	1.00	197.00	9.80	860,986.05	0.94
rail2536c	715.00	1.00	stall	-	714.00	2.22
rail2586c	1,007.00	1.00	stall	-	1,004.00	1.41
rail4284c	1,124.00	1.00	stall	-	1,119.00	0.19
rail4872	1,614.00	1.00	stall	-	1,622.00	1.21
A2C1S1	19,879.93	1.00	19,883.00	4.69	19,528.00	2.49
B1C1S1	38,530.65	1.00	stall	-	42,489.19	2.83
B2C1S1	48,279.95	1.00	59,217.00	7.94	46,198.04	3.34
sp97ic	1,280,793,707.52	1.00	314,809,721.12	4.30	1,073,817,106.08	1.10
sp98ar	988,402,511.00	1.00	1,050,813,686.88	8.95	1,136,153,068.32	1.48
sp98ic	959,924,716.00	1.00	107,765,747.52	4.50	75,902,608.00	1.12
blp-ar98	25,094.03	1.00	-	26.38	22,372.94	2.75
blp-ic97	7,874.87	1.00	7,618.51	6.82	7,423.29	1.44
blp-ic98	14,848.96	1.00	14,822.98	6.39	13,707.62	2.04
blp-ir98	5,388.84	1.00	6,573.28	4.53	5,277.00	1.41
CMS750_4	606.00	1.00	-	712.45	6.23	4.04
berlin_5_8_0	79.00	1.00	79.00	28.46	79.00	1.70
railway_8_1_0	440.00	1.00	436.00	60.23	441.00	2.26
usAbbrv.8.25_70	164.00	1.00	162.00	85.86	error	-
bg512142	120,738,665.00	1.00	120,690,438.67	too low	120,738,655.00	too low
dg012142	153,406,945.50	1.00	140,064,121.25	2.87	153,406,921.50	1.65
manpower1	8.00	1.00	-	254.84	10.00	2.44
manpower2	7.00	1.00	-	220.08	8.00	9.31
manpower3	6.00	1.00	-	181.08	6.00	7.36
manpower3a	6.00	1.00	-	365.51	7.00	9.54
manpower4	7.00	1.00	-	411.84	7.00	12.14
manpower4a	7.00	1.00	-	155.37	9.00	4.93
ljb2	7.31	1.00	7.31	too low	7.34	too low
ljb7	8.61	1.00	8.61	too low	8.61	too low
ljb9	9.48	1.00	9.48	30.44	9.48	0.48
ljb10	7.31	1.00	7.31	30.48	7.31	0.47
ljb12	6.20	1.00	6.20	35.28	6.20	0.53

Table 6.6: Average time per iteration (ratio with respect to FPcplex)(cont.d)

Chapter 7

Conclusions

This thesis required the study of three important optimization codes: ILOG CPLEX, GLPK, and Dash Xpress. Moreover, the Feasibility Pump code was analyzed to find all the CPLEX functions it uses and to understand which tuning it requires. After this studies, we have implemented the interfaces previously discussed.
The functioning of these interfaces is very good, indeed they preserve all the functionalities of the FP software, exploiting in the best possible way the solvers abilities.
Even if GLPK is a free LP solver, it can solve many difficult MIP problems if used within the FP framework. As to Xpress, this is a commercial software and better performance are expected. Indeed this software is faster than GLPK, but, on average, it is slightly slower than CPLEX. However, for the FP algorithm, Xpress can be considered a valid choice.
The interfaces limit, as explained in Chapter 5, is mainly the management of the error codes returned, which is too different in the three solvers. It is also very difficult to find a correspondence in many control attributes.
If further developed, the software proposed in this thesis could be very useful because it may be applied to all the pieces of software based on CPLEX which do not need a powerful solver as CPLEX.

Appendix A

Interface for GLPK (code)

A.1 interface.h

```
#include "c:/cygwin/usr/local/include/glpk.h"

typedef int * CPXENVptr;

typedef LPX * CPXLPptr;

#define CPXoptimize     CPXlpopt

*********************************

I don't report all the CPLEX parameters redefinition

*********************************
char *nomefile;
char *tipofile;

int lastsolver; //1: lp  2: mip 3:bar
int limiter; //Iteration limit  for simplex
int msglev;
int presolver;
double epagap;
double epgap;

//Creating problems
CPXLPptr CPXcreateprob (CPXENVptr env, int *status_p, char *probname);
CPXLPptr CPXcloneprob (CPXENVptr env, CPXLPptr lpx, int *status_p);

//Optimizing problems
int CPXprimopt ( CPXENVptr env, CPXLPptr lpx);
int CPXlpopt ( CPXENVptr env, CPXLPptr lpx);
int CPXbaropt ( CPXENVptr env, CPXLPptr lpx);
int CPXmipopt( CPXENVptr env, CPXLPptr lpx);
int CPXdualopt( CPXENVptr env, CPXLPptr lpx);
```

```
//Accessing LP/MIP results
int CPXgetstat (CPXENVptr env, CPXLPptr lpx);
int CPXgetobjval ( CPXENVptr env, CPXLPptr lpx, double *objval_p);
int CPXgetx (CPXENVptr env, CPXLPptr lpx, double *x, int begin, int end);
int CPXgetpi (CPXENVptr env, CPXLPptr lpx, double *pi, int begin, int end);
int CPXgetdj (CPXENVptr env, CPXLPptr lpx, double *dj, int begin, int end);
int CPXgetbase (CPXENVptr env, CPXLPptr lpx, int *cstat, int *rstat);
int CPXgetphase1cnt ( CPXENVptr env, CPXLPptr lpx);
int CPXgetmipobjval (CPXENVptr env, CPXLPptr lpx, double *objval_p);
int CPXgetmipx (CPXENVptr env, CPXLPptr lpx, double *x, int begin, int end );
int CPXgetbestobjval (CPXENVptr env, CPXLPptr lpx, double *objval_p);

//Problem modification
int CPXaddrows (CPXENVptr env, CPXLPptr lpx, int ccnt, int rcnt, int nzcnt, double *rhs,
    char *sense, int *rmatbeg, int *rmatind, double *rmatval, char **colname, char **rowname);
int CPXdelsetrows (CPXENVptr env, CPXLPptr lpx, int *delstat);
int CPXaddcols (CPXENVptr env, CPXLPptr lpx, int ccnt, int nzcnt, double *obj, int *cmatbeg,
    int *cmatind, double *cmatval, double *lb, double *ub, char **colname);
int CPXchgbds(CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, char *lu, double *bd);
int CPXchgsense (CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, char *sense);
int CPXchgobj (CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, double *values);
int CPXchgrhs (CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, double *values);
int CPXchgprobtype ( CPXENVptr env, CPXLPptr lpx, int type);

//Accessing problem data
int CPXgetprobname (CPXENVptr env, CPXLPptr lpx, char *buf_str, int bufspace, int *surplus_p);
int CPXgetcolname (CPXENVptr env, CPXLPptr lpx, char **name, char *namestore, int storespace,
    int *surplus_p, int begin, int end);
int CPXgetnumcols (CPXENVptr env, CPXLPptr lpx);
int CPXgetnumrows (CPXENVptr env, CPXLPptr lpx);
int CPXgetobjsen (CPXENVptr env, CPXLPptr lpx);
int CPXgetobj (CPXENVptr env, CPXLPptr lpx, double *obj, int begin, int end);
int CPXgetrhs (CPXENVptr env, CPXLPptr lpx, double *rhs, int begin, int end);
int CPXgetsense (CPXENVptr env, CPXLPptr lpx, char *sense, int begin, int end);
int CPXgetub (CPXENVptr env, CPXLPptr lpx, double *ub, int begin, int end);
int CPXgetrows (CPXENVptr env, CPXLPptr lpx, int *nzcnt, int *rmatbeg, int *rmatind,
     double *rmatval, int rmatspace, int *surplus, int begin, int end);
int CPXgetctype (CPXENVptr env, CPXLPptr lpx, char *xctype, int begin, int end);

//File reading/writing
int CPXreadcopyprob (CPXENVptr env, CPXLPptr lpx, char *filename_str, char *filetype_str);
int CPXwriteprob (CPXENVptr env, CPXLPptr lpx, char *filename, char *filetype);

//Parameters setting and querying routines
int CPXsetintparam (CPXENVptr env, int whichparam, int newvalue);
int CPXsetdblparam (CPXENVptr env, int whichparam, double newvalue);

//General utilities
CPXENVptr CPXopenCPLEX (int *status_p);
int CPXcloseCPLEX (CPXENVptr *env_p);
int CPXflushstdchannels (CPXENVptr env);
int CPXgetcallbackinfo ( CPXENVptr env, void *cbdata, int wherefrom, int whichinfo,
    void *result_p);
```

A.2 ifc_imp.c

```
#include "LocBra.h"

CPXENVptr CPXopenCPLEX (int *status_p){
    limiter = -1;
    presolver = 1;
    int valore = 1;
    msglev = 3;
    int *falso;
        falso = &valore;
    return falso;
}

CPXLPptr CPXcreateprob (CPXENVptr env, int *status_p, char *probname){
        LPX *temp;
        temp = lpx_create_prob();
        *status_p = 0;
        lpx_set_prob_name(temp, probname);
    return temp;
}

int CPXgetprobname (CPXENVptr env, CPXLPptr lpx, char *buf_str, int bufspace, int *surplus_p){
    int tmp = bufspace;
    char *probname = lpx_get_prob_name(lpx);
    int size = ( strlen( probname ) ) + 1;
    tmp = tmp - size;
    if(tmp >= 0)
    {
        int t;
        for( t = 0 ; t < size ; t++ )
        {
            *buf_str = probname[t];
            buf_str++;
        }
        *surplus_p = tmp;
        return 0;
    }
    *surplus_p = tmp;
    return CPXERR_NEGATIVE_SURPLUS;
}

int CPXgetnumrows (CPXENVptr env, CPXLPptr lpx){
    return lpx_get_num_rows(lpx);
}

int CPXgetnumcols (CPXENVptr env, CPXLPptr lpx){
    return lpx_get_num_cols(lpx);
}

int CPXreadcopyprob (CPXENVptr env, CPXLPptr lpx,  char *filename_str,  char *filetype_str)
{
    nomefile = filename_str;
    tipofile = filetype_str;
```

```
char *ext = filetype_str;
char *nome = filename_str;
int s;
int size = strlen(filename_str);
char *tmp = filename_str;

if (filetype_str == NULL)
{
    tmp = (tmp + size) - 3;
    s = Controllo(tmp,".lp",".LP");
    if (s == 1)
    {
        lp = lpx_read_cpxlp(filename_str);
        return 0;
    }

    if (s == 0)
    {
        tmp = (filename_str + size) - 4;
        s = Controllo(tmp,".mps",".MPS");
        if (s == 1)
        {
            lp = lpx_read_freemps(filename_str);
            return 0;
        }
    }
    return 1;
}

else
{
    int l = strlen(ext);
    if (l == 2)
    {
        if( Controllo(ext,"lp","LP") )
        {
            lp = lpx_read_cpxlp(filename_str);
            return 0;
        }
        else
        {
            return 1; //Ext unknown
        }
    }
    if (l == 3)
    {
        if( Controllo(ext,"mps","MPS") )
        {
            lp = lpx_read_freemps(filename_str);
            return 0;
        }
        else
        {
            return 1;  //Ext unknown
```

```
            }
        }
    }

    return 1;
}

int Controllo ( char *tmp, char *t, char *T)
{   int s = 0;
    if ( (strcmp(tmp,t) == 0)  ||  (strcmp(tmp,T)== 0) ) s=1;
    return s;
}

int CPXgetobjsen (CPXENVptr env, CPXLPptr lpx){
    if (lpx_get_obj_dir(lpx) == LPX_MAX) return CPX_MAX;
    if (lpx_get_obj_dir(lpx) == LPX_MIN) return CPX_MIN;
    return 0;
}

CPXLPptr CPXcloneprob ( CPXENVptr env, CPXLPptr lploc, int *status_p )
{
    CPXLPptr copia;
    char *ext = tipofile;
    char *nome = nomefile;
    int s;
    int size = strlen(nomefile);
    char *tmp = nomefile;
    *status_p = 0;
    if (tipofile == NULL)
    {
        tmp = (tmp + size) - 3;
        s = Controllo(tmp,".lp",".LP");
        if (s == 1)
        {
            copia = lpx_read_cpxlp(nomefile);
            return copia;
        }

        if (s == 0)
        {
            tmp = (nomefile + size) - 4;
            s = Controllo(tmp,".mps",".MPS");
            if (s == 1)
            {
                copia = lpx_read_freemps(nomefile);
                return copia;
            }
        }
        return NULL;
    }

    else
    {
        int l = strlen(ext);
```

```
        if (l == 2)
        {
            if( Controllo(ext,"lp","LP") )
            {
                copia = lpx_read_cpxlp(nomefile);
                return copia;
            }
            else
            {
                return NULL;
            }
        }
        if (l == 3)
        {
            if( Controllo(ext,"mps","MPS") )
            {
                copia = lpx_read_freemps(nomefile);
                return NULL;
            }
            else
            {
                return NULL;
            }
        }
    }

    return NULL;
}

int CPXgetctype( CPXENVptr env, CPXLPptr lpx, char *xctype, int begin, int end)
{
    lpx_set_class(lpx, LPX_MIP);
    begin++;
    end++;
    int i;
    for( i = begin ; i < (end + 1); i++ )
    {
        int t = lpx_get_col_kind(lpx, i);
        if ( t == LPX_CV )
        {
            *xctype = 'C';
            xctype++;
        }
        else
        {
            if ( t == LPX_IV )
            {
                if ( (lpx_get_col_lb(lpx, i) == 0) && (lpx_get_col_ub(lpx, i) == 1) )
                {
                    *xctype = 'B';
                    xctype++;
                }
                else
                {
```

```
                    *xctype = 'I';
                    xctype++;
                }
            }
        }
    }
    return 0;    //To see for errors
}

int CPXgetobj (CPXENVptr env, CPXLPptr lpx, double *obj, int begin, int end)
{
    begin++;
    end++;
    int i;
    for( i = begin ; i < (end+ 1) ; i++ )
    {
        obj[i-begin] = lpx_get_obj_coef(lpx, i);

    }
    return 0; //To see for errors
}


int CPXlpopt ( CPXENVptr env, CPXLPptr lpx) {
    lastsolver = 1;
    lpx_set_int_parm(lpx, LPX_K_MSGLEV, msglev);
    lpx_set_int_parm (lpx, LPX_K_PRESOL, presolver);
    lpx_set_int_parm (lpx, LPX_K_DUAL , 1);
    lpx_set_int_parm(lpx, LPX_K_ITCNT, 0);
    lpx_set_int_parm (lpx, LPX_K_ITLIM , limiter);
    lpx_simplex(lpx);
    return 0;
}

int CPXbaropt ( CPXENVptr env, CPXLPptr lpx) {
    lastsolver = 3;
    lpx_set_int_parm (lpx, LPX_K_ITLIM , limiter);
    lpx_set_int_parm(lpx, LPX_K_ITCNT, 0);
    lpx_set_int_parm(lpx, LPX_K_MSGLEV, msglev);
    lpx_interior(lpx);
    return 0;                    //To see for errors
}
int CPXgetstat (CPXENVptr env, CPXLPptr lpx) {  //OK for lpx_simplex and lpx_integer
    int stato;
    int res;
    switch (lastsolver)
    {
        case 1:
            stato = lpx_get_status (lpx);
            switch(stato)
            {
                case LPX_OPT:
                res = CPX_STAT_OPTIMAL;
                break;
```

```
                case LPX_FEAS:
                res = CPX_STAT_NUM_BEST;
                break;
                case LPX_INFEAS:
                res = CPX_STAT_INFEASIBLE;
                break;
                case LPX_NOFEAS:
                res = CPX_STAT_INFEASIBLE;
                break;
                case LPX_UNBND:
                res = CPX_STAT_INForUNBD;
                break;
                default:
                res = CPX_STAT_ABORT_USER;
                break;
            }
            break;
        case 2:
            stato = (lpx);
            switch(stato)
            {
                case LPX_I_OPT:
                res = CPXMIP_OPTIMAL;
                break;
                case LPX_I_FEAS:
                res = CPXMIP_NODE_LIM_FEAS;
                break;
                case LPX_I_UNDEF:
                res = CPXMIP_INForUNBD;
                break;
                case LPX_I_NOFEAS:
                res = CPXMIP_INFEASIBLE;
                break;
                default:
                res = CPX_STAT_ABORT_USER;
                break;
            }
            break;
    default:
    break;
    }
    return res;
}

int CPXprimopt ( CPXENVptr env, CPXLPptr lpx) {
    lastsolver = 1;
    lpx_set_int_parm(lpx, LPX_K_MSGLEV, msglev);
    lpx_set_int_parm (lpx, LPX_K_PRESOL, presolver);
    lpx_set_int_parm (lpx, LPX_K_DUAL , 0);
    lpx_set_int_parm(lpx, LPX_K_ITCNT,0);
    lpx_set_int_parm (lpx, LPX_K_ITLIM , limiter);
    lpx_simplex(lpx);
  return 0;                 //To see for errors
}
```

```
int CPXgetphase1cnt ( CPXENVptr env, CPXLPptr lpx) {
    return lpx_get_int_parm(lpx,LPX_K_ITCNT);
}

int CPXgetobjval ( CPXENVptr env, CPXLPptr lpx, double *objval_p) {
        *objval_p = lpx_get_obj_val(lpx);
    return 0;
}

int CPXdualopt (  CPXENVptr env, CPXLPptr lpx) {
    lastsolver = 1;
    lpx_set_int_parm(lpx, LPX_K_MSGLEV, msglev);
    lpx_set_int_parm (lpx, LPX_K_PRESOL, presolver);
    lpx_set_int_parm (lpx, LPX_K_DUAL , 1);
    lpx_set_int_parm(lpx, LPX_K_ITCNT,0);
    lpx_set_int_parm (lpx, LPX_K_ITLIM , limiter);
    lpx_simplex(lpx);
  return 0;         //To see for errors
}

int CPXchgobj (CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, double *values)
{
    int t;
    for( t = 0 ; t < cnt ; t++ )
    {
        lpx_set_obj_coef(lpx, (indices[t]+1), values[t]);
    }
    return 0; //To see for errors
}

int CPXgetub (CPXENVptr env, CPXLPptr lpx, double *ub, int begin, int end)
{
    begin++;
    end++;
    int i;
    for( i = begin ; i < (end + 1) ; i++ )
    {
        int bound = lpx_get_col_type(lpx, i);
        if( (bound == LPX_UP) || (bound == LPX_DB) || (bound == LPX_FX) )
        {
            if ( lpx_get_col_ub(lpx, i) > CPX_INFBOUND) *ub = CPX_INFBOUND;
            else *ub = lpx_get_col_ub(lpx, i);
            ub++;
        }
        else
        {
            *ub = CPX_INFBOUND;
            ub++;
        }
    }
    return 0; //To see for errors
}
```

```
int CPXchgprobtype ( CPXENVptr env, CPXLPptr lpx, int type) //Only CPXPROB_LP and CPXPROB_MILP
{
    if (type == CPXPROB_LP) lpx_set_class(lpx, LPX_LP);
    if (type == CPXPROB_MILP)   lpx_set_class(lpx, LPX_MIP);
    return 0;
}

int CPXaddrows (CPXENVptr env, CPXLPptr lpx, int ccnt, int rcnt, int nzcnt,
    double *rhs, char *sense, int *rmatbeg, int *rmatind, double *rmatval, char **colname,
        char **rowname)
{
    int t;
    if ( ccnt > 0)
    {
        int ncol = lpx_add_cols(lpx, ccnt);
        for ( t = 0 ; t < ccnt ; t++ )
        {
            lpx_set_col_bnds(lpx, (ncol+t), LPX_LO, 0, 0 ); // it is set as the CPLEX default
            if (colname != NULL) lpx_set_col_name(lpx, (ncol+t), colname[t]);
        }
    }

    int nrow = lpx_add_rows(lpx, rcnt);
    int inizio;
    int fine;
    int len;

    for ( t = 0 ; t < rcnt ; t++)
    {
        if (rowname != NULL) lpx_set_row_name(lpx, (nrow+t), rowname[t]);

        inizio = rmatbeg[t];
        if (t == (rcnt-1)) fine = (nzcnt-1);
        else fine = (rmatbeg[t+1]-1);
        len = fine - inizio + 1;
        int ind[len+1];
        double val[len+1];
        int q;
        int lentmp = 1;
        for( q = 1 ; q < (len+1) ; q++ )
        {
            if (rmatval[inizio] != 0)
            {
                ind[lentmp] = (rmatind[inizio] + 1);
                val[lentmp] = rmatval[inizio];
                lentmp++;
            }
            inizio++;
        }
        lentmp--;

        lpx_set_mat_row(lpx, (nrow+t), lentmp, ind, val);
        if (sense != NULL)
        {
```

```
            char tipo = sense[t];
            switch(tipo)
            {
            case 'L':
                if (rhs != NULL) lpx_set_row_bnds(lpx, (nrow+t), LPX_UP, rhs[t], rhs[t]);
                else lpx_set_row_bnds(lpx, (nrow+t), LPX_UP, 0, 0);
                break;
            case 'G':
                if (rhs != NULL) lpx_set_row_bnds(lpx, (nrow+t), LPX_LO, rhs[t], rhs[t]);
                else lpx_set_row_bnds(lpx, (nrow+t), LPX_LO, 0, 0);
                break;
            case 'R':
                if (rhs != NULL) lpx_set_row_bnds(lpx, (nrow+t), LPX_DB, rhs[t], rhs[t]);
                else lpx_set_row_bnds(lpx, (nrow+t), LPX_DB, 0, 0);
                break;
            case 'E':
                if (rhs != NULL) lpx_set_row_bnds(lpx, (nrow+t), LPX_FX, rhs[t], rhs[t]);
                else lpx_set_row_bnds(lpx, (nrow+t), LPX_FX, 0, 0);
                break;
            default:
                lpx_set_row_bnds(lpx, (nrow+t), LPX_FX, 0, 0);
                break;
            }
        }
    }
    return 0; //To see for errors
}

int CPXdelsetrows (CPXENVptr env, CPXLPptr lpx, int *delstat)
{
    int nrows = lpx_get_num_rows(lpx);
    int delrows[nrows+1];
    int i;
    int nrs = 0;
    for( i = 0 ; i < nrows ; i++ )
    {
        if (delstat[i] == 1)
        {
            nrs++;
            delrows[nrs] = (i+1);
        }
    }
    if (nrs > 0) lpx_del_rows(lpx, nrs, delrows);
    return 0; //To see for errors
}

int CPXgetx ( CPXENVptr env, CPXLPptr lpx, double *x, int begin, int end) {
    begin++;
    end++;
    int i;
    for (i = begin; i < (end+1) ; i++)
    {
        x[i-begin] = lpx_get_col_prim(lpx, i);
    }
```

```
    return 0;    //To see for errors
}

int CPXgetdj ( CPXENVptr env, CPXLPptr lpx, double *dj, int begin, int end) {
    begin++;
    end++;
    int i;
    for (i = begin ; i < (end + 1) ; i++)
    {
        *dj = lpx_get_col_dual(lpx, i);
        dj++;
    }
    return 0;    //To see for errors
}

int CPXgetrhs ( CPXENVptr env, CPXLPptr lpx, double *rhs, int begin, int end)
{
    begin++;
    end++;
    int i;
    for (i = begin ; i < (end + 1) ; i++)
    {
        int ub = lpx_get_row_type(lpx, i);
        if ( (ub == LPX_UP) || (ub == LPX_DB) || (ub == LPX_FX) )
        {
            *rhs = lpx_get_row_ub(lpx, i);
            rhs++;
        }

        else
        {
            if ( (ub == LPX_LO) )
            {
                *rhs = lpx_get_row_lb(lpx, i);
                rhs++;
            }
            else
            {
                *rhs = -CPX_INFBOUND;
                rhs++;
            }
        }
    }
    return 0; //To see for errors
}


int CPXgetpi ( CPXENVptr env, CPXLPptr lpx, double *pi, int begin, int end)
{
    begin++;
    end++;
    int i;
    for (i = begin ; i < (end + 1) ; i++)
    {
```

```
        *pi = lpx_get_row_dual(lpx, i);
        pi++;
    }
    return 0; //To see for errors
}

int CPXchgrhs (CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, double *values)
{
    int i;
    for ( i = 0 ; i < cnt ; i++ )
    {
        int tipo = lpx_get_row_type(lpx, (*indices+1));
        switch(tipo)
        {
            case LPX_UP:
            lpx_set_row_bnds(lpx, (indices[i]+1), LPX_UP, *values, *values);
            break;
            case LPX_LO:
            lpx_set_row_bnds(lpx, (indices[i]+1), LPX_LO, *values, *values);
            break;
            case LPX_DB:
            lpx_set_row_bnds(lpx, (indices[i]+1), LPX_DB, *values, *values);
            break;
            case LPX_FX:
            lpx_set_row_bnds(lpx, (indices[i]+1), LPX_FX, *values, *values);
            break;
            default:
            break;
        }
        values++;
        indices++;
    }
    return 0;  //To see for errors
}

int CPXchgbds(CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, char *lu, double *bd)
{
    int i;
    int indice;
    for ( i = 0 ; i < cnt ; i++ )
    {
        indice = indices[i] + 1;
        int tipo = lpx_get_col_type(lpx, indice);
        double oldlb;
        double oldub;

        switch(tipo)
        {
            case LPX_FR:
                switch(lu[i])
                {
                    case 'U':
                        lpx_set_col_bnds(lpx, indice, LPX_UP, bd[i], bd[i]);
                        break;
```

```
        case 'L':
            lpx_set_col_bnds(lpx, indice, LPX_LO, bd[i], bd[i]);
            break;
        case 'B':
            lpx_set_col_bnds(lpx, indice, LPX_FX, bd[i], bd[i]);
            break;
        default:
            break;
    }
break;
case LPX_LO:
    switch(lu[i])
    {
        case 'U':
            oldlb = lpx_get_col_lb(lpx, indice);
            if ( bd[i] == oldlb )   lpx_set_col_bnds(lpx, indice, LPX_FX,
                oldlb, bd[i]);
            else
            {
                if ( bd[i] > oldlb )    lpx_set_col_bnds(lpx, indice, LPX_DB,
                    oldlb, bd[i]);
                else lpx_set_col_bnds(lpx, indice, LPX_UP, oldlb, bd[i]);
            }
            break;
        case 'L':
            lpx_set_col_bnds(lpx, indice, LPX_LO, bd[i], bd[i]);
            break;
        case 'B':
            lpx_set_col_bnds(lpx, indice, LPX_FX, bd[i], bd[i]);
            break;
        default:
            break;
    }
break;
case LPX_UP:
    switch(lu[i])
    {
        case 'U':
            lpx_set_col_bnds(lpx, indice, LPX_UP, oldlb, bd[i]);
            break;
        case 'L':
            oldub = lpx_get_col_ub(lpx, indice);
            if (oldub == bd[i]) lpx_set_col_bnds(lpx, indice, LPX_FX, bd[i], oldub);
            else
            {
                if (oldub > bd[i]) lpx_set_col_bnds(lpx, indice, LPX_DB, bd[i], oldub);
                else lpx_set_col_bnds(lpx, indice, LPX_LO, bd[i], oldub);
            }
            break;
        case 'B':
            lpx_set_col_bnds(lpx, indice, LPX_FX, bd[i], bd[i]);
            break;
        default:
            break;
```

```
    }
break;
case LPX_DB:
    switch(lu[i])
    {
        case 'U':
            oldlb = lpx_get_col_lb(lpx, indice);
            if (oldlb == bd[i]) lpx_set_col_bnds(lpx, indice, LPX_FX, oldlb, bd[i]);
            else
            {
                if (oldlb < bd[i]) lpx_set_col_bnds(lpx, indice, LPX_DB, oldlb, bd[i]);
                else lpx_set_col_bnds(lpx, indice, LPX_UP, oldlb, bd[i]);
            }
            break;
        case 'L':
            oldub = lpx_get_col_ub(lpx, indice);
            if (oldub == bd[i]) lpx_set_col_bnds(lpx, indice, LPX_FX, bd[i], oldub);
            else
            {
                if (oldub > bd[i]) lpx_set_col_bnds(lpx, indice, LPX_DB, bd[i], oldub);
                else lpx_set_col_bnds(lpx, indice, LPX_LO, bd[i], bd[i]);
            }
            break;
        case 'B':
            lpx_set_col_bnds(lpx, indice, LPX_FX, bd[i], bd[i]);
            break;
        default:
            break;
    }
break;
case LPX_FX:
    switch(lu[i])
    {
        case 'U':
        oldlb = lpx_get_col_lb(lpx, indice);
        if (oldlb == bd[i]) lpx_set_col_bnds(lpx, indice, LPX_FX, oldlb, bd[i]);
        else
        {
            if (oldlb < bd[i])  lpx_set_col_bnds(lpx, indice, LPX_DB, oldlb, bd[i]);
            else lpx_set_col_bnds(lpx, indice, LPX_UP, oldlb, bd[i]);
        }
        break;
        case 'L':
            oldub = lpx_get_col_ub(lpx, indice);
            if (oldub == bd[i]) lpx_set_col_bnds(lpx, indice, LPX_FX, bd[i], oldub);
            else
            {
                if (oldub > bd[i]) lpx_set_col_bnds(lpx, indice, LPX_DB, bd[i], oldub);
                else lpx_set_col_bnds(lpx, indice, LPX_LO, bd[i], bd[i]);
            }
            break;
        case 'B':
            lpx_set_col_bnds(lpx, indice, LPX_FX, bd[i], bd[i]);
            break;
```

```
                    default:
                        break;
                }
            break;
            default:
            break;
        }
    }
    return 0; //To see for errors
}

int CPXgetcolname ( CPXENVptr env, CPXLPptr lpx, char **name, char *namestore,
    int storespace, int *surplus_p, int begin, int end)
{
    begin++;
    end++;
    int i;
    int tmp = storespace;
    for ( i = begin ; i < (end+1) ; i++)
    {
        char *nametmp = lpx_get_col_name(lpx, i);
        int size = ( strlen( nametmp  ) ) + 1;
        tmp = tmp - size;
        if (tmp >= 0)
        {
            *name = namestore;
            name++;
            int t;
            for( t = 0 ; t < size ; t++ ){
                *namestore = *nametmp;
                namestore++;
                nametmp++;
            }
        }
    }
    *surplus_p = tmp;
    if (tmp < 0) return CPXERR_NEGATIVE_SURPLUS;
    return 0;
}

int CPXgetmipobjval (CPXENVptr env, CPXLPptr lpx, double *objval_p)
{
    *objval_p = lpx_mip_obj_val(lpx);
    return 0; //To see for errors
}

int CPXgetmipx (CPXENVptr env, CPXLPptr lpx, double *x, int begin, int end )
{
    begin++;
    end++;
    int i;
    for ( i = begin ; i < (end+1) ; i++)
    {
        *x = lpx_mip_col_val(lpx, i);
```

```
        x++;
    }
    return 0; //To see for errors
}

int CPXmipopt( CPXENVptr env, CPXLPptr lpx)
{
    lastsolver = 2;
    lpx_set_int_parm(lpx, LPX_K_ITCNT,0);
    lpx_set_int_parm (lpx, LPX_K_ITLIM , limiter);
    lpx_set_real_parm(lpx, LPX_K_TOLINT, epagap);
    lpx_set_real_parm(lpx, LPX_K_TOLOBJ, epgap);
    lpx_integer(lpx);
    return 0; //To see for errors
}

int CPXsetintparam (CPXENVptr env, int whichparam, int newvalue)
{
    switch(whichparam)
    {
        case CPX_PARAM_MIPINTERVAL:
        break;
        case CPX_PARAM_SCRIND:
          if (newvalue == 0) msglev = 0;
          if (newvalue == 1) msglev = 3;
        break;
        case CPX_PARAM_ITLIM:
            limiter = newvalue;
        break;
        case CPX_PARAM_INTSOLLIM:
        break;
        case CPX_PARAM_MIPSTART:
        break;
        case CPX_PARAM_PREIND:
            presolver = newvalue;
        default:
        break;
    }
    return 0;
}

int CPXsetdblparam (CPXENVptr env, int whichparam, double newvalue)
{
    switch(whichparam)
    {
        case CPX_PARAM_EPAGAP:
            epagap = newvalue;
        break;
        case CPX_PARAM_EPGAP:
            epgap = newvalue;
        break;
        case CPX_PARAM_CUTUP:
        break;
        case    CPX_PARAM_TILIM:
```

```
        break;
        default:
        break;
    }
    return 0;
}

int CPXwriteprob (CPXENVptr env, CPXLPptr lpx, char *filename, char *filetype)
    // Only mps and lp allowed
{
    if (filetype == NULL)
    {
        char *punto = strrchr( filename, '.' );
        punto++;
        if ( (strcmp( punto, "lp") == 0) || (strcmp( punto, "LP") == 0) )
            return lpx_write_cpxlp(lpx, filename);
        if ( (strcmp( punto, "mps") == 0) || (strcmp( punto, "MPS") == 0) )
            return lpx_write_freemps(lpx, filename);
        return 1;
    }
    if ( (strcmp( filetype, "lp") == 0) || (strcmp( filetype, "LP") == 0)  )
        return lpx_write_cpxlp(lpx, filename);
    else
    {
        if ( (strcmp( filetype, "mps") == 0) || (strcmp( filetype, "MPS") == 0) )
            return lpx_write_freemps(lpx, filename);
        else return 1;
    }
}

int CPXaddcols (CPXENVptr env, CPXLPptr lpx, int ccnt, int nzcnt, double *obj, int *cmatbeg,
    int *cmatind, double *cmatval, double *lb, double *ub, char **colname)
{
    int ncol = lpx_add_cols(lpx, ccnt);
    int t;
    double inf;
    double sup;
    for ( t = 0 ; t < ccnt ; t++ )
    {
        if (colname != NULL) lpx_set_col_name(lpx, (ncol+t), colname[t]);
        if (lb == NULL)
        {
            inf = 0;
        }
        else if (lb[t] <= -CPX_INFBOUND )
             {
                    inf = -CPX_INFBOUND;
             }
             else
             {
                    inf = lb[t];
             }
        if ( (ub[t] >= CPX_INFBOUND) || (ub == NULL) )
        {
```

```
            sup = CPX_INFBOUND;
        }
        else
        {
            sup = ub[t];
        }
        if (inf == -CPX_INFBOUND)
        {
            if (sup == CPX_INFBOUND) lpx_set_col_bnds(lpx, (ncol+t), LPX_FR, inf, sup);
            else lpx_set_col_bnds(lpx, (ncol+t), LPX_UP, inf, sup);
        }
        else
        {
            if (sup == CPX_INFBOUND) lpx_set_col_bnds(lpx, (ncol+t), LPX_LO, inf, sup);
            else lpx_set_col_bnds(lpx, (ncol+t), LPX_DB, inf, sup);
        }

    }

    int len;

    for ( t = 0 ; t < ccnt ; t++)
    {
        if (obj == NULL) lpx_set_obj_coef(lpx, (ncol+t), 0);
        else lpx_set_obj_coef(lpx, (ncol+t), obj[t]);
        if (t != ccnt-1) len = cmatbeg[t+1] - cmatbeg[t];
        else len = ( nzcnt-cmatbeg[t] );
        int ind[len+1];
        double val[len+1];
        int q;

        for( q = 1 ; q < (len + 1); q++ )
        {
            ind[q] = (1 + cmatind[cmatbeg[t] + q - 1]);
            val[q] = cmatval[cmatbeg[t] + q - 1];
        }

        lpx_set_mat_col(lpx, (ncol+t), len, ind, val);
    }
    return 0; //To see for errors
}

int CPXcloseCPLEX (CPXENVptr *env_p)
{
    return 0;
}

int CPXgetbase (CPXENVptr env, CPXLPptr lpx, int *cstat, int *rstat)
{
    int i;
    if (cstat != NULL)
    {
    for ( i = 0 ; i < lpx_get_num_cols(lpx) ; i++ )
    {
```

```
        int tipo = lpx_get_col_stat( lpx, (i+1) );
        switch(tipo)
        {
            case LPX_BS:
            cstat[i] = CPX_BASIC;
            break;
            case LPX_NL:
            cstat[i] = CPX_AT_LOWER;
            break;
            case LPX_NU:
            cstat[i] = CPX_AT_UPPER;
            break;
            case LPX_NF:
            cstat[i] = CPX_FREE_SUPER;
            break;
            case LPX_NS:    // Different in Cplex, set CPX_AT_LOWER
            cstat[i] = CPX_AT_UPPER;
            break;
            default:
            break;
        }
    }
    }
    if (rstat != NULL)
    {
    for ( i = 0 ; i < (lpx) ; i++ )
    {
        int tipo;
        tipo = lpx_get_row_stat( lpx, (i+1) );
        switch(tipo)
        {
            case LPX_BS:
            rstat[i] = CPX_BASIC;
            break;
            case LPX_NL:
            rstat[i] = CPX_AT_LOWER;
            break;
            case LPX_NU:
            rstat[i] = CPX_AT_UPPER;
            break;
            case LPX_NF:
            rstat[i] = CPX_FREE_SUPER;
            break;
            case LPX_NS:   //Different in CPLEX, set CPX_AT_LOWER
            rstat[i] = CPX_AT_LOWER;
            break;
            default:
            break;
        }
    }
    }
    return 0;
}
```

```
int CPXchgsense (CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, char *sense)
{
    int i;
    for (i = 0 ; i < cnt ; i++ )
    {
        char tipo  = sense[i];
        switch(tipo)
        {
            case 'L':
                lpx_set_row_bnds(lpx, (indices[i]+1), LPX_UP, sense[i], sense[i]);
                break;
            case 'G':
                lpx_set_row_bnds(lpx, (indices[i]+1), LPX_LO, sense[i], sense[i]);
                break;
            case 'E':
                lpx_set_row_bnds(lpx, (indices[i]+1), LPX_FX, sense[i], sense[i]);
                break;
            case 'R':
                lpx_set_row_bnds(lpx, (indices[i]+1), LPX_DB, 0, 0);
                break;
            default:
            break;
        }
    }
    return 0; //To see for errors
}

int CPXflushstdchannels (CPXENVptr env)
{
    return 0;
}

int CPXgetbestobjval (CPXENVptr env, CPXLPptr lpx, double *objval_p)
    //There isn't this function in GLPK
{
    *objval_p = lpx_get_obj_val(lpx); //Return the root value
    return 0;
}

int CPXgetcallbackinfo ( CPXENVptr env, void *cbdata, int wherefrom, int whichinfo, void *result_p)
{
    cbdata = lp;
    double r;
    if (wherefrom == CPX_CALLBACK_BARRIER)
    {   r = lpx_ipt_obj_val(lp); }
    else
    {   if (wherefrom == CPX_CALLBACK_MIP)
         {  r = lpx_mip_obj_val(lp); }
         else { r = lpx_get_obj_val(lp); }
    }
    result_p = &r;
    return 0;
}
```

```
int CPXgetrows (CPXENVptr env, CPXLPptr lpx, int *nzcnt, int *rmatbeg, int *rmatind,
    double *rmatval, int rmatspace, int *surplus, int begin, int end)
{
    begin++;
    end++;
    int nz = 0;
    int i;
    int tmp = rmatspace;
    int ncols  = lpx_get_num_cols(lpx);
    int len;
    for( i = begin ; i < (end + 1 ) ; i++ )
    {
        int ind[ncols+1];
        double val [ncols+1];
        len = lpx_get_mat_row(lpx, i, ind, val);
        int t;
        for ( t = 0 ; t < (len) ; t++ )
        {
            if ( tmp > 0 )
            {
                rmatbeg[i-begin] = nz;
                rmatval[nz] = val[t];
                rmatind[nz] = (ind[t] -1);
                nz++;
                tmp--;
            }
            *surplus--;
        }
    }
    *nzcnt = nz;
    return 0; //To see for errors
}

int CPXgetsense (CPXENVptr env, CPXLPptr lpx, char *sense, int begin, int end)
{
    begin++;
    end++;
    int i;
    int type;
    for (i = begin ; i< (end + 1); i++)
    {   type = lpx_get_row_type(lpx,i);
        if (type == LPX_LO)
        {
            *sense = 'G';
            sense++;
        }
        if (type == LPX_UP)
        {
            *sense = 'L';
            sense++;
        }
        if ( (type == LPX_DB) && ( lpx_get_row_ub(lpx, i) != lpx_get_row_lb(lpx, i) ) )
        {
            *sense = 'R';
```

```
            sense++;
        }
        if ( (type == LPX_FX) || ( (type == LPX_DB) && ( lpx_get_row_ub(lpx, i)
                                                == lpx_get_row_lb(lpx, i) ) ) )
        {
            *sense = 'E';
            sense++;
        }
    }
    return 0; //To see for errors
}
```

Appendix B

Interface for Xpress (code)

B.1 interface.h

```
#include "C:/XpressMP/include/xprs.h"

typedef int * CPXENVptr;

typedef XPRSprob CPXLPptr;

#define CPXoptimize     CPXlpopt

*********************************

I don't report all the CPLEX parameters redefinition

*********************************

int lastsolver;//1: lp  2: mip 3: bar
int probclass;
int presolver;
int lpiter;
int output;
int timemax;


//Creating problems
CPXLPptr CPXcreateprob (CPXENVptr env, int *status_p, char *probname);
CPXLPptr CPXcloneprob (CPXENVptr env, CPXLPptr lpx, int *status_p);

//Optimizing problems
int CPXprimopt ( CPXENVptr env, CPXLPptr lpx);
int CPXlpopt ( CPXENVptr env, CPXLPptr lpx);
int CPXdualopt( CPXENVptr env, CPXLPptr lpx);
int CPXmipopt( CPXENVptr env, CPXLPptr lpx);

//Accessing LP/MIP results
int CPXgetstat (CPXENVptr env, CPXLPptr lpx);
```

```
int CPXgetobjval ( CPXENVptr env, CPXLPptr lpx, double *objval_p);
int CPXgetx (CPXENVptr env, CPXLPptr lpx, double *x, int begin, int end);
int CPXgetdj (CPXENVptr env, CPXLPptr lpx, double *dj, int begin, int end);
int CPXgetpi (CPXENVptr env, CPXLPptr lpx, double *pi, int begin, int end);
int CPXgetbase (CPXENVptr env, CPXLPptr lpx, int *cstat, int *rstat)
int CPXgetphase1cnt ( CPXENVptr env, CPXLPptr lpx);
int CPXgetmipobjval (CPXENVptr env, CPXLPptr lpx, double *objval_p);
int CPXgetbestobjval (CPXENVptr env, CPXLPptr lpx, double *objval_p);
int CPXgetmipx (CPXENVptr env, CPXLPptr lpx, double *x, int begin, int end );

//Problem modification
int CPXaddrows (CPXENVptr env, CPXLPptr lpx, int ccnt, int rcnt, int nzcnt, double *rhs,
    char *sense, int *rmatbeg, int *rmatind, double *rmatval, char **colname, char **rowname);
int CPXdelsetrows (CPXENVptr env, CPXLPptr lpx, int *delstat);
int CPXaddcols (CPXENVptr env, CPXLPptr lpx, int ccnt, int nzcnt, double *obj, int *cmatbeg,
    int *cmatind, double *cmatval, double *lb, double *ub, char **colname);
int CPXchgsense (CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, char *sense);
int CPXchgobj (CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, double *values);
int CPXchgrhs (CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, double *values);
int CPXchgprobtype ( CPXENVptr env, CPXLPptr lpx, int type);
int CPXchgbds(CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, char *lu, double *bd);

//Accessing problem data
int CPXgetnumrows (CPXENVptr env, CPXLPptr lpx);
int CPXgetnumcols (CPXENVptr env, CPXLPptr lpx);
int CPXgetobjsen (CPXENVptr env, CPXLPptr lpx);
int CPXgetctype (CPXENVptr env, CPXLPptr lpx, char *xctype, int begin, int end);
int CPXgetprobname (CPXENVptr env, CPXLPptr lpx, char *buf_str, int bufspace, int *surplus_p);
int CPXgetobj (CPXENVptr env, CPXLPptr lpx, double *obj, int begin, int end);
int CPXgetub (CPXENVptr env, CPXLPptr lpx, double *ub, int begin, int end);
int CPXgetsense (CPXENVptr env, CPXLPptr lpx, char *sense, int begin, int end);
int CPXgetrhs (CPXENVptr env, CPXLPptr lpx, double *rhs, int begin, int end);
int CPXgetcolname (CPXENVptr env, CPXLPptr lpx, char **name, char *namestore, int storespace,
    int *surplus_p, int begin, int end);
int CPXgetrows (CPXENVptr env, CPXLPptr lpx, int *nzcnt, int *rmatbeg, int *rmatind,
    double *rmatval, int rmatspace, int *surplus, int begin, int end);

//File reading/writing
int CPXreadcopyprob (CPXENVptr env, CPXLPptr lpx, char *filename_str, char *filetype_str);
int CPXwriteprob (CPXENVptr env, CPXLPptr lpx, char *filename, char *filetype);

//Parameters setting and querying routines
int CPXsetintparam (CPXENVptr env, int whichparam, int newvalue);
int CPXsetdblparam (CPXENVptr env, int whichparam, double newvalue);

//General utilities
CPXENVptr CPXopenCPLEX (int  *status_p);
int CPXcloseCPLEX (CPXENVptr *env_p);
int CPXflushstdchannels (CPXENVptr env);
int CPXgetcallbackinfo ( CPXENVptr env, void *cbdata, int wherefrom, int whichinfo, void *result_p);
```

B.2 ifc_imp.c

```
#include "LocBra.h"


CPXENVptr CPXopenCPLEX (int *status_p){
    presolver = 1;
    lpiter = 2147483645;
    output = 1;
    *status_p = XPRSinit(NULL);
    int x;
    return &x;
}

void XPRS_CC Message(XPRSprob my_prob, void* my_object, const char *msg, int len, int msgtype)
{
    switch(msgtype)
    {
        case 4: /* error */

        case 3: /* warning */

        case 2: /* dialogue */

        case 1: /* information */

        printf("%s\n", msg);

        break;

        default: /* exiting - buffers need flushing */

        fflush(stdout);

    break;
    }
}

CPXLPptr CPXcreateprob (CPXENVptr env, int *status_p, char *probname){
        XPRSprob temp;
        *status_p = XPRScreateprob(&temp);
        XPRSsetcbmessage(temp , Message, NULL);
        XPRSsetprobname( temp, probname );
        return temp;
}

int CPXgetprobname (CPXENVptr env, CPXLPptr lpx, char *buf_str, int bufspace, int *surplus_p){
    int tmp = bufspace;
    char probname[200];
    XPRSgetprobname(lpx, probname);
    int size = ( strlen( probname ) ) + 1;
    tmp = tmp - size;
    if(tmp >= 0)
    {
```

```
        int t;
        for( t = 0 ; t < size ; t++ )
        {
            *buf_str = probname[t];
            buf_str++;
        }
        *surplus_p = tmp;
        return 0;
    }
    *surplus_p = tmp;
    return CPXERR_NEGATIVE_SURPLUS;
}

int CPXgetnumrows (CPXENVptr env, CPXLPptr lpx){
    int rows;
    XPRSgetintattrib(lpx,XPRS_ROWS,&rows);
    return rows;
}

int CPXgetnumcols (CPXENVptr env, CPXLPptr lpx){
    int cols;
    XPRSgetintattrib(lpx,XPRS_COLS,&cols);
    return cols;
}

int CPXreadcopyprob (CPXENVptr env, CPXLPptr lpx,  char *filename_str,  char *filetype_str)
{
    int size = strlen(filename_str);
    char * point = strchr( filename_str, '.' );
    if (point == NULL) return XPRSreadprob(lpx, filename_str,"");
    else
    {
        if ( Controllo(point,".lp",".LP") == 1 )
        {
            filename_str[size-3] = '\0';
            return XPRSreadprob(lpx, filename_str,"");
        }
        if ( Controllo(point,".mps",".MPS") == 1 )
        {
            filename_str[size-4] = '\0';
            return XPRSreadprob(lpx, filename_str,"");
        }
    }
return -1;
}

int Controllo ( char *tmp, char *t, char *T)
{
    int s = 0;
    if ( (strcmp(tmp,t) == 0)  ||  (strcmp(tmp,T)== 0) ) s=1;
    return s;
}

int CPXgetobjsen (CPXENVptr env, CPXLPptr lpx)
```

```
{
    double sense;
    XPRSgetdblattrib(lpx ,XPRS_OBJSENSE,&sense);
    if (sense == -1) return CPX_MAX;
    else return CPX_MIN;
}

CPXLPptr CPXcloneprob ( CPXENVptr env, CPXLPptr lpx, int *status_p )
{
    CPXLPptr tmp;
    XPRScreateprob(&tmp);
    XPRSsetcbmessage(tmp ,Message,NULL);
    *status_p =  XPRScopyprob(tmp, lpx, "cloned");
    return tmp;
}

int CPXgetctype( CPXENVptr env, CPXLPptr lpx, char *xctype, int begin, int end)
{
    XPRSgetcoltype(lpx, xctype, begin, end);
    int i;
    double ub;
    double lb;
    int mindex[1];
    char qctype[1];
    qctype[0] = 'B';
    int nels;
    for( i = begin; i < (end+1); i++)
    {
        if (xctype[i-begin] == 'I')
        {
            XPRSgetub(lpx, &ub, i, i);
            XPRSgetlb(lpx, &lb, i, i);
            if ( ( ub == 1 ) && (lb == 0 ) ) xctype[i-begin] = 'B';
        }
    }
    return 0; //To see for errors
}

int CPXgetobj (CPXENVptr env, CPXLPptr lpx, double *obj, int begin, int end)
{
    return XPRSgetobj(lpx, obj, begin, end);
}

int CPXgetstat (CPXENVptr env, CPXLPptr lpx)
{
    int res;
    int lpstat;
    int mipstat;
    switch(lastsolver)
    {
        case 1:
        XPRSgetintattrib(lpx ,XPRS_LPSTATUS,&lpstat);
        switch (lpstat)
        {
```

```
            case XPRS_LP_OPTIMAL:
            res = CPX_STAT_OPTIMAL;
            break;
            case XPRS_LP_INFEAS:
            res = CPX_STAT_INFEASIBLE;
            break;
            case XPRS_LP_UNBOUNDED:
            res = CPX_STAT_UNBOUNDED;
            break;
            case XPRS_LP_UNFINISHED:
            res = CPX_STAT_ABORT_IT_LIM;
            break;
            default:
            res = CPX_STAT_ABORT_USER;
            break;
        }
        break;

        case 2:
        XPRSgetintattrib(lpx ,XPRS_MIPSTATUS,&mipstat);
        switch(mipstat)
        {
            case XPRS_MIP_NO_SOL_FOUND:
            res =  CPXMIP_ABORT_INFEAS;
            break;
            case XPRS_MIP_SOLUTION:
            res = CPXMIP_ABORT_FEAS;
            break;
            case XPRS_MIP_INFEAS:
            res = CPXMIP_INFEASIBLE ;
            break;
            case XPRS_MIP_OPTIMAL:
            res = CPXMIP_OPTIMAL;
            break;
            default:
            res = CPXMIP_TIME_LIM_INFEAS;
            break;
        }
        break;
        default:
        res = -1;
        printf("Attention: a problem in CPXgetstat");
        break;
    }
    return res;
}

int CPXprimopt ( CPXENVptr env, CPXLPptr lpx)
{
    lastsolver = 1;
    XPRSsetintcontrol(lpx, XPRS_OUTPUTLOG, output);
    XPRSsetintcontrol(lpx, XPRS_MAXTIME, -timemax);
    XPRSsetintcontrol(lpx, XPRS_PRESOLVE, presolver);
    XPRSsetintcontrol(lpx, XPRS_LPITERLIMIT,lpiter);
```

```
    double sense;
    XPRSgetdblattrib(lpx, XPRS_OBJSENSE, &sense);
    if (sense == 1)  XPRSminim(lpx, "pl");
    else if (sense == -1) XPRSminim(lpx, "pl");
    else return -1;
  return 0;                     //To see for errors
}

int CPXgetphase1cnt ( CPXENVptr env, CPXLPptr lpx) {
    int iter;
    XPRSgetintattrib(lpx, XPRS_SIMPLEXITER, &iter);
    return iter;
}

int CPXgetobjval ( CPXENVptr env, CPXLPptr lpx, double *objval_p) {
    XPRSgetdblattrib(lpx, XPRS_LPOBJVAL, objval_p);
    return 0;
}

int CPXchgobj (CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, double *values)
{
    return XPRSchgobj(lpx, cnt, indices, values);
}

int CPXgetub (CPXENVptr env, CPXLPptr lpx, double *ub, int begin, int end)
{
    return XPRSgetub(lpx, ub, begin, end);
}

int CPXchgprobtype ( CPXENVptr env, CPXLPptr lpx, int type)
{
    if (type == CPXPROB_LP) probclass = 0;
    if (type == CPXPROB_MILP)   probclass = 1;
    return 0;
}

int CPXaddrows (CPXENVptr env, CPXLPptr lpx, int ccnt, int rcnt, int nzcnt, double *rhs,
    char *sense, int *rmatbeg, int *rmatind, double *rmatval, char **colname, char **rowname)
{
    if ( ccnt > 0)
    {
        double zerosc[ccnt];
        double bdl[ccnt];
        double bdu[ccnt];
        int i;
        for (i = 0; i < ccnt; i++)
        {
            zerosc[i] = 0;
            bdl[i] = 0;
            bdl[i] = XPRS_PLUSINFINITY;
        }
        XPRSaddcols(lpx, ccnt, 0, zerosc, 0, 0, 0, bdl, bdu);
    }
    double zerosr[rcnt];
```

```
    XPRSaddrows(lpx, rcnt, nzcnt, sense, rhs, zerosr, rmatbeg, rmatind, rmatval);
    return 0;
}

int CPXdelsetrows (CPXENVptr env, CPXLPptr lpx, int *delstat)
{
    int nrows;
    XPRSgetintattrib(lpx,XPRS_ROWS,&nrows);
    int delrows[nrows];
    int i;
    int nrs = 0;
    for( i = 0 ; i < nrows ; i++ )
    {
        if (delstat[i] == 1)
        {
            nrs++;
            delrows[nrs] = i;
        }
    }
    if (nrs > 0) XPRSdelrows(lpx, nrs, delrows);
    return 0; //To see for errors
}

int CPXgetx ( CPXENVptr env, CPXLPptr lpx, double *x, int begin, int end)
{
    int ncols;
    XPRSgetintattrib(lpx,XPRS_COLS,&ncols);
    double x2[ncols];
    XPRSgetsol(lpx, x2, NULL, NULL, NULL);
    int i;
    for (i = begin; i < (end+1) ; i++)
    {
        *x = x2[i];
        x++;
    }
    return 0; //To see for errors
}

int CPXgetdj ( CPXENVptr env, CPXLPptr lpx, double *dj, int begin, int end) {
    int ncols;
    XPRSgetintattrib(lpx,XPRS_COLS,&ncols);
    double dj2[ncols];
    XPRSgetsol(lpx, NULL, NULL, NULL, dj2);
    int i;
    for (i = begin; i < (end+1) ; i++)
    {
        *dj = dj2[i];
        dj++;
    }
    return 0; //To see for errors
}

int CPXgetrhs ( CPXENVptr env, CPXLPptr lpx, double *rhs, int begin, int end)
{
```

```
    XPRSgetrhs(lpx, rhs, begin, end);
    return 0; //To see for errors
}

int CPXgetpi ( CPXENVptr env, CPXLPptr lpx, double *pi, int begin, int end)
{
    int nrows;
    XPRSgetintattrib(lpx,XPRS_ROWS,&nrows);
    double pi2[nrows];
    XPRSgetsol(lpx, NULL, NULL, pi2, NULL);
    int i;
    for (i = begin; i < (end+1) ; i++)
    {
        *pi = pi2[i];
        pi++;
    }
    return 0;
}

int CPXchgrhs (CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, double *values)
{
    return XPRSchgrhs(lpx, cnt, indices, values);
}

int CPXchgbds(CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, char *lu, double *bd)
{
    return XPRSchgbounds(lpx, cnt, indices, lu, bd);
}

int CPXgetcolname ( CPXENVptr env, CPXLPptr lpx, char **name, char *namestore, int storespace,
    int *surplus_p, int begin, int end)
{
    return 0;
}

int CPXgetmipobjval (CPXENVptr env, CPXLPptr lpx, double *objval_p)
{
    XPRSgetdblattrib(lpx, XPRS_MIPOBJVAL, objval_p);
    return 0; //To see for errors
}

int CPXgetmipx (CPXENVptr env, CPXLPptr lpx, double *x, int begin, int end )
{
    int ncols;
    XPRSgetintattrib(lpx,XPRS_COLS,&ncols);
    double x2[ncols];
    XPRSgetsol(lpx, x2, NULL, NULL, NULL);
    int i;
    for (i = begin; i < (end+1) ; i++)
    {
        *x = x2[i];
        x++;
    }
    return 0; //To see for errors
```

```
}

int CPXmipopt( CPXENVptr env, CPXLPptr lpx)
{
    lastsolver = 2;
    XPRSsetintcontrol(lpx,XPRS_OUTPUTLOG, output);
    XPRSsetintcontrol(lpx,XPRS_MAXTIME, -timemax);
    XPRSinitglobal(lpx);
    return XPRSglobal(lpx);
}

int CPXsetdblparam (CPXENVptr env, int whichparam, double newvalue)
{
    switch(whichparam)
    {
        case CPX_PARAM_EPAGAP:
        break;
        case CPX_PARAM_EPGAP:
        break;
        case CPX_PARAM_CUTUP:
        break;
        case    CPX_PARAM_TILIM:
            timemax = newvalue;
        break;
        default:
        break;
    }
    return 0;
}

int CPXsetintparam (CPXENVptr env, int whichparam, int newvalue)
{

    switch(whichparam)
    {
        case CPX_PARAM_MIPINTERVAL:
        break;
        case CPX_PARAM_PREIND:
            presolver = newvalue;
        break;
        case CPX_PARAM_SCRIND:
                output = newvalue;
        break;
        case CPX_PARAM_ITLIM:
            lpiter = newvalue;
        break;
        case CPX_PARAM_INTSOLLIM:
        break;
        case CPX_PARAM_MIPSTART:
        break;
        default:
        break;
    }
    return 0;
```

```
}

int CPXwriteprob (CPXENVptr env, CPXLPptr lpx, char *filename_str, char *filetype)
{
    char * point = strchr( filename_str, '.' );
    int size = strlen(filename_str);
    if ((filetype == NULL) && (point == NULL)) return XPRSwriteprob(lpx, filename_str,"l");
    if ((filetype == NULL) && (point != NULL))
    {
        if ( Controllo(point,".lp",".LP") == 1 )
        {
            filename_str[size-3] = '\0';
            return XPRSwriteprob(lpx, filename_str,"l");
        }
        if ( Controllo(point,".mps",".MPS") == 1 )  //File .mat
        {
            filename_str[size-4] = '\0';
            return XPRSwriteprob(lpx, filename_str,"");
        }
    }
    if ( Controllo(filetype,".mps",".MPS") == 1 )  //File .mat
    {
        filename_str[size-4] = '\0';
        return XPRSwriteprob(lpx, filename_str,"");
    }
    if ( Controllo(filetype,".lp",".LP") == 1 )
    {
        filename_str[size-3] = '\0';
        return XPRSwriteprob(lpx, filename_str,"l");
    }
    return -1;
}

int CPXaddcols (CPXENVptr env, CPXLPptr lpx, int ccnt, int nzcnt, double *obj, int *cmatbeg,
    int *cmatind, double *cmatval, double *lb, double *ub, char **colname)
{
  return XPRSaddcols(lpx, ccnt, nzcnt, obj, cmatbeg, cmatind, cmatval, lb, ub);
}

int CPXcloseCPLEX (CPXENVptr *env_p)
{
    XPRSfree();
    return 0;
}

int CPXgetbase (CPXENVptr env, CPXLPptr lpx, int *cstat, int *rstat)
{
    return XPRSgetbasis(lpx, rstat, cstat);
}

int CPXchgsense (CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, char *sense)
{
    return XPRSchgrowtype(lpx, cnt, indices, sense);
}
```

```
int CPXflushstdchannels (CPXENVptr env)
{
    return 0;
}

int CPXgetbestobjval (CPXENVptr env, CPXLPptr lpx, double *objval_p)
{
    XPRSgetdblattrib(lpx, XPRS_LPOBJVAL, objval_p);  //Return the root value
    return 0;
}

int CPXgetrows (CPXENVptr env, CPXLPptr lpx, int *nzcnt, int *rmatbeg, int *rmatind,
    double *rmatval, int rmatspace, int *surplus, int begin, int end)
{
    XPRSgetrows(lpx, NULL, NULL, NULL, 0, nzcnt, begin, end);
    *surplus = *nzcnt - rmatspace;
    XPRSgetrows(lpx, rmatbeg, rmatind, rmatval, rmatspace, nzcnt, begin, end);
    return 0; //To see for errors
}

int CPXgetsense (CPXENVptr env, CPXLPptr lpx, char *sense, int begin, int end)
{
    return XPRSgetrowtype(lpx, sense, begin, end);
}

int CPXgetcallbackinfo ( CPXENVptr env, void *cbdata, int wherefrom, int whichinfo, void *result_p)
{
    cbdata = lp;
    double r;
    if (wherefrom == CPX_CALLBACK_BARRIER) XPRSgetdblattrib(lp, XPRS_BARPRIMALOBJ, &r);
    else
    {
        if (wherefrom == CPX_CALLBACK_MIP)  XPRSgetdblattrib(lp, XPRS_MIPOBJVAL , &r);
        else    XPRSgetdblattrib(lp, XPRS_LPOBJVAL, &r);
    }
    result_p = &r;
    return 0;
}

int CPXdualopt( CPXENVptr env, CPXLPptr lpx)
{
    lastsolver = 1;
    XPRSsetintcontrol(lpx,XPRS_MAXTIME, -timemax);
    XPRSsetintcontrol(lpx,XPRS_OUTPUTLOG, output);
    XPRSsetintcontrol(lpx, XPRS_PRESOLVE, presolver);
    XPRSsetintcontrol(lpx,XPRS_LPITERLIMIT,lpiter);
    double sense;
    XPRSgetdblattrib(lpx, XPRS_OBJSENSE, &sense);
    if (sense == 1) XPRSminim(lpx, "dl");
    else if (sense == -1) XPRSminim(lpx, "dl");
    else return -1;
  return 0;                     //To see for errors
}
```

```
int CPXlpopt( CPXENVptr env, CPXLPptr lpx)
{
    lastsolver = 1;
    XPRSsetintcontrol(lpx,XPRS_OUTPUTLOG, output);
    XPRSsetintcontrol(lpx,XPRS_MAXTIME, -timemax);
    XPRSsetintcontrol(lpx, XPRS_PRESOLVE, presolver);
    XPRSsetintcontrol(lpx,XPRS_LPITERLIMIT,lpiter);
    double sense;
    XPRSgetdblattrib(lpx, XPRS_OBJSENSE, &sense);
    if (sense == 1) XPRSminim(lpx, "l");
    else if (sense == -1) XPRSminim(lpx, "l");
    else return -1;
  return 0;                     //To see for errors
}
```

Bibliography

[1] M. Fischetti, F. Glover, A. Lodi: The Feasibility pump. May 8, 2004.

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

[3] CPLEX: ILOG CPLEX 8.1 User's Manual and Reference Manual. ILOG, S.A., 2003 (http://www.ilog.com)

[4] GNU Linear Programming Kit, Reference manual version 4.7. August 2004

[5] Xpress-MP: Dash optimization Xpress-MP release 2004b. Xpress-Optimizer Reference Manual, 2004 (http://www.dashoptimization.com)

[6] T. Achterberg, T. Koch, A. Martin. The mixed integer programming library: MIPLIB 2003. `http://miplib.zib.de`.

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

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

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

[10] R.E. Bixby. Personal communication, 2003.

[11] J.W. Chinneck. The constraint consesus method for finding approximately feasible points in nonlinear programs. *Technical Report Carleton University*, Ottawa, Ontario, Canada, October 2002.

[12] E. Danna, E. Rothberg, C. Le Paper. Exploring relaxation induced neighborhoods to improve MIP solutions. *Mathematical Programming* DOI 10.1007/s10107-004-0518-7, 2004.

[13] Double-Click sas. Personal communication, 2001.

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

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

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

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

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

[19] G.W. Klau. Personal communication, 2002.

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

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

[22] M. Lübbecke. Personal communication, 2002.

[23] A.J. Miller. Personal communication, 2003.

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

[25] J. Patel and J.W. Chinneck. Active-Constraint Variable Ordering for Faster Feasibility of Mixed Integer Linear Programs. *Technical Report Carleton University*, Ottawa, Ontario, Canada, November 2003.

[26] E. Rothberg. Personal communication, 2002.

[27] E. Rothberg. Personal communication, 2003.

[28] K. Spielberg, M. Guignard. Sequential (Quasi) Hot Start Method for BB (0,1) Mixed Integer Programming. *Wharton School Research Report*, 2002.