UNIVERSITÀ DEGLI STUDI DI PADOVA
FACOLTÀ DI INGEGNERIA
DIPARTIMENTO DI ELETTRONICA ED INFORMATICA

TESI DI LAUREA

IMPLEMENTATION OF AN LP-BASED HEURISTIC FOR MIXED INTEGER PROGRAMMING THROUGH OPEN-SOURCE SOFTWARE

Relatore: Prof. Matteo Fischetti

Laureando: Luigi Bancaro

ANNO ACCADEMICO 2003-2004

Contents

Summary

The goal of this thesis is the implementation of the Feasibility Pump (FP), a MIP heuristic originally written for CPLEX, through open-source software like QSopt and Coin-Or. This aim has been reached implementing two libraries of functions (one for each solver) that capture all CPLEX routines called by FP and perform the same operations using the function of Coin-Or or QSopt. In this way all functionalities of FP are preserved and the interfaces obtained can be applied to any program written for CPLEX with the possibility to run it also with Coin-Or and QSopt, without the need to rewrite the code.

The FP and the correspondent interfaces have been tested on a good number of MIPs problems and so it has been possible to evaluate potentialities of FP through the three solvers and to establish the performance of each software.

The interface for QSopt has been written in C, while that for Coin-Or has been written in C++.

Sommario

Lo scopo di questa tesi è l'implementazione della Feasibility Pump (FP), un euristico MIP originariamente scritto per CPLEX, attraverso software open-source quali QSopt e Coin-Or. Questo obiettivo è stato raggiunto implementando due librerie di funzioni (una per ciascun solver) che catturano tutte le routine di CPLEX chiamate dalla FP ed eseguono le stesse operazioni utilizzando le funzioni di Coin-Or e QSopt. In questo modo tutte le funzionalità della FP sono state mantenute e le interfacce realizzate possono essere applicate anche ad altri programmi scritti per CPLEX, con la possibilità di eseguirli con Coin-Or e QSopt, senza la necessità di riscrivere il codice.

La FP e le due interfacce sono state testate su un certo numero di problemi di tipo MIP; è stato così possibile valutare le potenzialità della FP attraverso i tre solvers ed anche le prestazioni di ciascun software.

L'interfaccia per QSopt è stata scritta in C, mentre quella per Coin-Or in C++.

Introduction

This thesis describes the implementation of an LP-based heuristic for mixed integer programming (Feasibility Pump), through open-source software.

Feasibility Pump (FP) is a heuristic algorithm, proposed by M. Fischetti, A. Lodi and F. Glover [25], whose target is to provide a feasible solution to NP-hard 0-1 MIP 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. Therefore, a heuristic method like the FP to find a feasible solution for hard MIPs is highly important.

Original FP is written to be linked to the commercial software ILOG-CPLEX 8.1 [26]. The main target of this thesis is the implementation of this algorithm through open-source software like Coin-Or [27] and QSopt [28]: using these it is possible to evaluate the feasible solutions found, the time to find them by applying the FP, and then to compare the results.

Remarkably, this target has been reached without the need to rewrite the entire code of FP, but implementing a program (one for each solver) that captures all calls to CPLEX's routines (from FP) and performs the same operations by using the functions of each software. In this way we have obtained a general interface applicable to other programs written for CPLEX, to be linked with Coin-Or and QSopt.
For this thesis the following solvers have been used:

- ILOG-CPLEX 8.1, obtained through a departmental license
- QSopt, free software downloaded from the web
- Coin-Or, free software downloaded from the web

and the platform Cygwin [29], that is a free Linux-like environment for Windows.

Chapter-by-chapter:

- **Chapter 1 Feasibility Pump**: the article written by Fischetti, Lodi and Glover with the description of the Feasibility Pump [25].
- **Chapter 2 ILOG-CPLEX**: description of the commercial solver taken from [26].
- **Chapter 3 QSopt**: brief description of the free software taken from [28].
- **Chapter 4 Coin-Or**: complete description of the open-source software taken from [27].
- **Chapter 5 Feasibility Pump implemented with QSopt and Coin-Or**: development of the interfaces from CPLEX to QSopt and from CPLEX to Coin. Subsequently, a detailed description of the implementation for each routine called by FP.
- **Chapter 6 Computational results**: a set of tests to compare the performance of FP through the new interfaces to different solvers.
- **Chapter 7 Conclusions**: final considerations about the work.

In appendix:

- **A - Source code: "from ILOG-CPLEX to QSopt"**
- **B - Source code: "from ILOG-CPLEX to Coin-Or"**

Chapter 1

The Feasibility Pump

1.1 Introduction

This chapter is taken by the paper [25] and it introduces the Feasibility Pump (FP), an algorithm developed by M. Fischetti, F. Glover and A. Lodi. The focus is 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 [9] and RINS/guided dives [7] 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 [2, 3, 4, 7, 9, 10, 11, 12, 13, 14, 18, 17, 20, 21, 24].

In this chapter a new approach to compute heuristic MIP solutions is shown and it is called *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 its implementation for 0-1 MIPs. Computational results are presented in Section 1.2, where a comparison is reported, 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 addressed in Section 1.3, where the use of approximate LP solutions is investigated. In the same section it is also addressed the possibility of using the `FP` scheme to produce a sequence of feasible solutions of better and better quality.

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 [9]. 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 [6].

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. At

```
The Feasibility Pump (basic version):
  1. initialize nIT := 0 and $x^* := \mathrm{argmin}\{c^T x : Ax \geq b\}$;
  2. if $x^*$ is integer, return($x^*$);
  3. let $\widetilde{x} := [x^*]$ (= rounding of $x^*$);
  4. while (time < TL) do
  5.    let nIT := nIT $+1$ and compute $x^* := \mathrm{argmin}\{\Delta(x, \widetilde{x}) : Ax \geq b\}$;
  6.    if $x^*$ is integer, return($x^*$);
  7.    if $\exists\, j \in \mathcal{I} : [x_j^*] \neq \widetilde{x}_j$ then
  8.       $\widetilde{x} := [x^*]$
        else
  9.       flip the TT = rand(T/2,3T/2) entries $\widetilde{x}_j$ ($j \in \mathcal{I}$)
           with highest $|x_j^* - \widetilde{x}_j|$
 10.    endif
 11. enddo
```

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

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 [1] 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	$\vert\mathcal{I}\vert$	m	Name	n	$\vert\mathcal{I}\vert$	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 [1]

Name	n	$\vert\mathcal{I}\vert$	m	source	Name	n	$\vert\mathcal{I}\vert$	m	source
biella1	7328	6110	1203	[9]	blp-ar98	16021	15806	1128	[18]
NSR8K	38356	32040	6284	[9]	blp-ic97	9845	9753	923	[18]
dc1c	10039	8380	1649	[8]	blp-ic98	13640	13550	717	[18]
dc1l	37297	35638	1653	[8]	blp-ir98	6097	6031	486	[18]
dolom1	11612	9720	1803	[8]	CMS750_4	11697	7196	16381	[15]
siena1	13741	11775	2220	[8]	berlin_5_8_0	1083	794	1532	[15]
trento1	7687	6415	1265	[8]	railway_8_1_0	1796	1177	2527	[15]
rail507	63019	63009	509	[9]	usAbbrv.8.25_70	2312	1681	3291	[15]
rail2536c	15293	15284	2539	[9]	manpower1	10565	10564	25199	[22]
rail2586c	13226	13215	2589	[9]	manpower2	10009	10008	23881	[22]
rail4284c	21714	21705	4284	[9]	manpower3	10009	10008	23915	[22]
rail4872c	24656	24645	4875	[9]	manpower3a	10009	10008	23865	[22]
A2C1S1	3648	192	3312	[9]	manpower4	10009	10008	23914	[22]
B1C1S1	3872	288	3904	[9]	manpower4a	10009	10008	23866	[22]
B2C1S1	3872	288	3904	[9]	ljb2	771	681	1482	[7]
sp97ic	12497	12497	1033	[9]	ljb7	4163	3920	8133	[7]
sp98ar	15085	15085	1435	[9]	ljb9	4721	4460	9231	[7]
sp98ic	10894	10894	825	[9]	ljb10	5496	5196	10742	[7]
bg512142	792	240	1307	[19]	ljb12	4913	4633	9596	[7]
dg012142	2080	640	6310	[19]					

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

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 [23] 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 optimality 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 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,

	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)

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 seconds. This is of course true also for `ILOG-Cplex`. E.g., for `manpower` instances Bixby [5] 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 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.

[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 [9], RINS or guided dives [7] schemes.

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: $\alpha = 0.50$, $\beta = 0.25$, $\gamma = 0.20$, $\texttt{SIL} = 1,000$, and $\texttt{IL} = 250$.

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 ILOG-Cplex

ILOG-CPLEX [26] 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

QSopt

3.1 QSopt

The QSopt software [28] was developed by David Applegate, William Cook, Sanjeeb Dash, and Monika Mevenkamp.

The main purpose of QSopt is to provide a callable function library for use within applications such as the traveling salesman problem or mixed-integer-programming. It can also be used as a stand-alone code to solve large-scale linear programming problems. The QSopt function library has been implemented in the C programming language and is available on Windows and various LINUX/UNIX platforms. An alpha version of the QSopt library has been converted to Java as well. QSopt has been extensively used by a team of researchers in their work on the traveling salesman problem; their Concorde code has successfully solved many TSP instances, the largest having 15,112 cities. The QSopt library, can be used at no cost for research or education purposes.

3.2 Problems format

QSopt supports two ASCII formats for specifying LP and MIP problems. The formats are based on two industry standards, the MPS format and the LP format. Both formats provide convenient mechanisms to define an LP or MIP problem, that is:

- its objective and whether it should be minimized or maximized,
- the constraints with their expressions, senses, and right-hand-sides,

- the problem's variables, and their bounds,
- which variables are integer variables, and
- the problem's name.

The QSopt parser expects that all problems that it reads have at least one constraint and at least one variable. The objective function that it reads may be empty, that is, it may contain no terms; in such a case, the optimization routines attempt only to find a feasible solution when solving the problem.

3.3 Default variable bounds

The LP and MPS input formats do not require that all variable upper and lower bounds be given explicitly. The QSopt parser assumes by default that variables are greater than or equal to zero and have no upper bound. Integer variables are assumed to be binary, that is, they have a lower bound of 0 and an upper bound of 1. The bounds sections in LP and MPS files are used to define different variable bounds. When defining variable bounds keep the following rules in mind.

1. A variable's upper bound must be greater than or equal to its lower bound.
2. Negative infinity may not be used as an upper bound and positive infinity may not be used as lower bound.
3. If a variable's upper bound is defined either as a nonnegative number or as positive infinity, then its lower bound defaults to zero. Thus defining a zero upper bound without defining a lower bound fixes a variable to zero.
4. If a variable's upper bound is defined as negative number its lower bound defaults to negative infinity.
5. If a variable's lower bound is defined either as a negative number or as negative infinity, then its upper bound defaults to zero.
6. If a variable's lower bound is defined either as zero or as a positive number, then its upper bound is assumed to be positive infinity.

3.4 Callable Library

QSopt's callable function library provides a set of functions for creating, manipulating, and solving LP problems. The library is written in the (ANSI) C programming language, and it is distributed as a header file qsopt.h and as an archive file qsopt.a on UNIX/LINUX systems, and as a dynamic link library QSlib.dll on Windows systems. Thread-safe versions of the library are available for use in re-entrant programming applications.

Chapter 4

Coin-Or

4.1 Description

The Computational Infrastructure for Operations Research (COIN-OR, or simply COIN) [27] project is an initiative to spur the development of open-source software for the operations research community. Coin is open source because when people can read, redistribute, and modify the source code, software evolves. People improve it, people adapt it, people fix bugs. The results of open-source development have been remarkable. Community-owned and -developed software written under open-source licenses have produced high-quality, high-performance, secure code – code on which much of the Internet is run.

The goal of project is to create for mathematical software what the open literature is for mathematical theory. To build an open-source community for operations research software in order to speed deployment of models, algorithms, and cutting-edge research, as well as provide a forum for peer review of software similar to that provided by archival journals for theoretical research.

This is a lofty goal, but it's a worthwhile one. Only the community of users and contributors can define what is needed to make it a reality.

The following is a list of projects currently being hosted by COIN-OR:

- ALPS: the abstract library for parallel search
- BCP: a parallel branch-cut-price framework
- CBC: Coin Branch and Cut, a branch and cut code
- CGL: a cut generation library

- CLP: COIN L P, a native simplex solver
- DFO: a package for solving general nonlinear optimization problems when derivatives are unavailable
- IPOPT: an interior point algorithm for general large-scale nonlinear optimization
- Multifario: a continuation method for computing implicitly defined manifolds
- NLPAPI: a subroutine interface for defining and solving nonlinear programming problems
- OSI: an open solver interface layer
- OTS: an open framework for tabu search
- SBB: Simple Branch and Bound, a branch and cut code
- SMI: Stochastic Modeling Interface, for optimization under uncertainty
- SYMPHONY: a callable library for solving mixed-integer linear programs
- VOL: the Volume Algorithm

4.2 Projects hosted by Coin-Or

4.2.1 ALPS

ALPS is a framework for implementing parallel graph search algorithms. It generalizes many of the notions present in BCP, allowing the implementation of a wider range of algorithms with a simplified interface.

ALPS implements the search handling methods required for implementing large-scale, data-intensive parallel search algorithms, such as those used for solving discrete optimization problems.

It is the base layer of a planned hierarchy that will include a library for solving mixed-integer linear programs. However, ALPS is still considered an experimental code.

4.2.2 BCP

BCP is a parallel framework for implementing branch, cut, and price algorithms for solving mixed integer programs (MIPs). BCP provides the user with an object-oriented framework that can be used to develop an efficient problem class specific MIP solver without all the implementational effort involved with implementing a branch and bound framework from scratch.

Because BCP is an open-source framework, users have the flexibility to customize any aspect of their BCP algorithm. BCP is appropriate for researchers who would like to experiment with different MIP formulations, new cut and/or variable generation techniques, branching strategies, etc., as well as power users who would like to solve intractable problems in a parallel environment.

BCP processes the Branch-and-Bound search tree nodes in parallel by employing a master/slave model.

BCP uses the OSI, which enables the use of any LP solver that OSI is interfaced with. The main features of BCP are generalized branching objects, strong branching, reduced cost fixing, use of cut and variable pools, handling locally valid cuts, etc.

BCP has been used for problems of type multiple knapsack with color constraints(mkc), max-cut, and some proprietary projects. An alternative to BCP is SYMPHONY that is a very similar open-source, parallel branch, cut, and price framework implemented in C. It has a more simplified interface and may be a better option for some users.

4.2.3 CBC

It is a branch and cut code designed to work with any OSI capable solver and in particular Clp. It used to be called Sbb (Simple Branch and Bound) but due to people confusing it with GAMS Sbb solver it has been renamed to Cbc.

CBC is not very fast out of the box, but is very flexible so it can be very effective with correct use of cut generators and heuristics. For instance, cut generators can be switched off, on every so often or only at root node (which is probably best default). It is written using the Osi interface so it can use any code which is supported.

CBC is designed to be less heavyweight than BCP or Symphony. It is very easy to add new cut generators and heuristics and branching methods such as lot-sizing variables.

4.2.4 CGL

CGL is a Cut Generator Library which will include the standard cuts from the literature and provide a starting point for more advanced and problem specific cuts. The cut generator currently in Cgl are:

- simple rounding cut generator
- knapsack cover cut generator
- generalized odd hole cut generator Gomory cut generator
- lift-and-project cuts using "norm 1"
- probing cuts
- flow cover cut generator

4.2.5 CLP

The COIN-OR LP code is designed to be a high quality Simplex code provided under the terms of the Common Public License. CLP is written in C++, and is primarily intended to be used as a callable library (though a rudimentary stand-alone executable exists). The first release was version .90. The current release is version .99.9.

CLP includes primal and dual Simplex solvers. Both dual and primal algorithms can use matrix storage methods provided by the user (0-1 and network matrices are already supported in addition to the default sparse matrix). The dual algorithm has Dantzig and Steepest edge row pivot choices; new ones may be provided by the user. The same is true for the column pivot choice of the primal algorithm. The primal can also use a non linear cost which should work for piecewise linear convex functions. CLP also includes a barrier method for solving LPs.

CLP has been tested on many problems of up to 1.5 million constraints and has shown itself as reliable as OSL. It is also being tested in the context of SBB ("Simple Branch and Bound", which is used to solve integer programs), but more testing is needed before it can get to version 1.0. CLP uses sparse matrix techniques designed for very large problems. The design criteria were for it not to be too slow. Some speed has been sacrificed to make the code less opaque OSL (not difficult!).

The CLP barrier method solves convex QPs as well as LPs. In general, a barrier method requires implementation of the algorithm, as well as a fast Cholesky factorization. CLP provides the algorithm, and is expected to have a reasonable factorization implementation by the release of CLP version 1.0. However, the sparse factorization requires a good ordering algorithm, which the user is expected to provide (perhaps a better factorization code as well).
Then Cholesky factorizations codes are supported by CLP's barrier method becauset he Cholesky interface is flexible enough so that a variety of Cholesky ordering and factorization codes can be used. Interfaces are provided to each of the following:

- Anshul Gupta's WSSMP parallel enabled ordering and factorization code
- Sivan Toledo's TAUCS parallel enabled factorization code (the package includes third party ordering codes)
- University of Florida's Approximate Minimum Degree (AMD) ordering code (the CLP native factorization code is used with this ordering code)
- CLP native code: very weak ordering but competitive nonparallel factorization
- Fast dense factorization

For quadratic programming can be used the interior point algorithm that is much more elegant and normally much faster than the quadratic simplex code. Caution is suggested with the presolve as not all bugs have been found and squashed when a quadratic objective is used. One may wish to switch off the crossover to a basic feasible solution as the simplex code can be slow. The sequential linear code is useful as a “crash” to the simplex code; its convergence is poor but, say, 100 iterations could set up the problem well for the simplex code.

4.2.6 DFO

DFO is a Fortran package for solving general nonlinear optimization problems that have the following characteristics:

- they are relatively small scale (less than 100 variables),
- their objective function is relatively expensive to compute and derivatives of such functions are not available and
- cannot be estimated efficiently

There also may be some noise in the function evaluation procedures. Such optimization problems arise, for example, in engineering design, where the objective function evaluation is a simulation package treated as a black box.

4.2.7 IPOPT

IPOPT (Interior Point OPTimizer) implements an interior point algorithm for nonlinear, nonconvex, constrained optimization problems. The IPOPT distribution also contains components that allow the optimization of systems described by differential-algebraic equations (optimal control). The solver itself is implemented in Fortran 77, while some of the additional components are written in C.

IPOPT can be used as a general purpose nonlinear programming (NLP) solver for AMPL and CUTE using the interfaces provided with the IPOPT package, or it can be invoked from Fortran application. While the source code for IPOPT itself is released as open source under the Common Public License (CPL), compilation requires third party components (such as BLAS, LAPACK, some routines from the Harwell Subroutine Library, and possibly others) which you have to obtain separately. The included INSTALL file gives detailed instructions on how to obtain and compile these components. (These components are covered by license agreements different from CPL and may not be free for commercial use.)

IPOPT implements a interior point method for nonlinear programming. Search directions (coming from a linearization of the optimality conditions) can be computed in a full-space version by solving a large symmetric linear system. Alternatively, search directions can be obtained using a coordinate decomposition approach, which allows one to tailor the linear algebra to specific problem characteristics. The implementation of the dynamic optimization component is an example for the latter where the discretized equations, obtained from orthogonal collocation, are solved by an elemental decomposition. IPOPT can employ second derivative information, if available, or otherwise approximate it by means of a quasi-Newton approach (BFGS and SR1).
Global convergence of the method is ensured by a line search procedure, where one can choose from among several merit functions and a novel filter method.

4.2.8 Multifario

Multifario is a subroutine library which implements a continuation method for computing implicitly defined manifolds, and other manifolds that arise in dynamical systems. An implicitly defined manifold is a set of points which satisfy $F(u) = 0$, for some smooth $F : IR^n \to IR^{n-k}$. This set of points consists of pieces of k-dimensional manifolds which meet at (k-1)-dimensional manifolds of singular points. Multifario takes as input a set of initial points on the manifold, and computes the manifold of points connected to the initial points.

To use Multifario two “solvers” are provided, one for algebraic systems, and one for two point boundary value problems. The user creates a data structure called a “ImplicitMF”, by calling a routine and passing either subroutines or strings containing expressions. Then the user calls a routine to compute the manifold. The result is either a data structure in memory (an Atlas), which the user can interrogate, or files on disk containing a set of polyhedra which cover the manifold, and optionally the centers of the polyhedra. These files can be used to render the manifold as a tiff or a Postscript file, using utilities that are included. They can also be converted to a couple of other file formats (DX, VBM, POV-Ray).
While the source code for Multifario itself is released as open source under the

Common Public License (CPL), compilation requires third party components (such as LAPACK).

Multifario uses an algorithm that computes well spaced points on the manifold. At each point the tangent space (k-vectors), a radius and a polyhedron are stored. As points are added the polyhedron is trimmed so that the polyhedron form an approximate polyhedral tiling of the manifold. At each step a point on the boundary of this tiling is chosen, projected onto the manifold, the tangent space and rsius are calculated, and the polyhedra are updated.

4.2.9 NLPAPI

NLPAPI is a subroutine library with routines for building nonlinear programming problems. The general form is an objective with a set of simple bounds, equality and inequality constraints. It is built around the "Group Partially Separable" structure that LANCELOT defines, but constraints and objective may also be defined as functions of the problem variables. To use NLPAPI, the user creates a "NLProblem", which is a pointer to a data structure. Then the objective and constraints are added with various calls. Once the problem is defined our interface to LANCELOT may be used to create a NLLancelot data structure, and pass it and the problem to a routine which either minimizes or maximizes the objective subject to the constraints.

4.2.10 OSI

The Open Solver Interface (OSI) is a uniform API (Application Program Interface) for calling math programming solvers. Programs written to the OSI standard may be linked to any solver with an OSI interface and should produce correct results. The OSI has been significantly extended compared to its first incarnation. Currently, the OSI supports linear programming solvers (soon to be redesigned to support non-LP solvers) and has rudimentary support for integer programming. Among others the following operations are supported:

- creating the LP formulation;

- directly modifying the formulation by adding rows/columns;
- modifying the formulation by adding cutting planes provided by CGL;
- solving the formulation (and resolving after modifications);
- extracting solution information;
- invoking the underlying solver's branch-and-bound component.

Programs in the CGL module are written to call the solver through the OSI. The following solvers currently work with OSI:

- CLP
- CPLEX
- dylp
- GLPK
- OSL
- SOPLEX
- VOL
- XPRESS-MP

4.2.11 OTS

Tabu Search is a meta-strategy for guiding known heuristics past the traps of local optimality. Popularized by Glover in the early 90s, Tabu Search has been applied to integer programming problems involving scheduling, routing, traveling salesman and related problems.
OTS is a framework for Tabu Search written in Java , which enables researchers to quickly develop Tabu Search methodologies.

4.2.12 SBB

SBB stands for Simple Branch and Bound. When COIN LP was being written, the Osi interface demanded an integer solver. With Strong Branching it was 460 lines of code, without 300 lines. All future development will be on Cbc (Coin Branch and Cut) which is just a renamed version of Sbb.

4.2.13 SMI

SMI stands for Stochastic Modeling Interface. It is an interface for problems in which uncertainty and optimization appear together. There are many modeling and algorithmic approaches that could belong here, like: recourse programming, chance constrained programming, stochastic control and dynamic programming, robust optimization, etc, etc. SMI is intended to be like OSI in the sense that an SmiXX object is an implementation derived from a base class that takes care of a number of commonly encountered programming issues, like handling probability distributions, managing problem generation, interacting with solvers to obtain solution information, etc.

The current release implements a multiperiod scenario stochastic programming object called SmiScnModel. It supports an SMPS file reader method, a direct “genScenario” method, a method to generate a deterministic equivalent, and several methods to get solution data by scenario. This is a fully native COIN implementation.

Today there is no actively developed, generally available commercial platform to support optimization under uncertainty. It is important to have a reliable framework available for applications development. Furthermore, there is a multitude of academic and local efforts that might benefit by being based on or compared to a reliable implementation.
For future releases are planned two main categories:

- Linkage to modeling languages. Plan to release SISP (Simple Interface for Stochastic Programming) which allows a modeling language to use a stoch file.

- Linkage to solvers. Provide a native implementation of the model construction that allows a Benders decomposition solver node-by-node access problem data.

4.2.14 SYMPHONY

SYMPHONY is a callable library for solving mixed-integer linear programs (MILPs) that can be customized through a wide variety of user callback functions and control parameters.
About BCP and SYMPHONY, BCP is a C++ framework based roughly on an ancestor of SYMPHONY, so they take substantially the same approach and have similar functionality. Currently, SYMPHONY is easier to use "out of the box" and is a better choice for beginners. It is also more efficient for performing pure branch and cut (without column generation). BCP has improved on SYMPHONY is some areas, such as support for branch and price. SYMPHONY's support for branch and price is limited. BCP is missing some features that SYMPHONY has, such as an implementation of global cut pools. SYMPHONY and BCP may be combined into a third-generation framework currently under development.

4.2.15 VOL

The Volume Algorithm(VOL) is a subgradient method that produces primal as well as dual solutions. The primal solution comes from estimating the volumes below the faces of the dual problem. This is an approximate method so the primal vector might have small infeasiblities that are negligible in many practical settings. The original subgradient algorithm produces only dual solutions.

VOL it has been tested with a variety of combinatorial problems like: Crew scheduling, Fleet assignment, Facility location, Steiner trees, Max-Cut, Quadratic knapsack.
Then, with VOL can be solved linear program: one can get good and fast approximate solutions for combinatorial linear programs. These are LP's where all variables are between 0 and 1. And for integer programs VOL gives bounds based on a relaxation of the IP. It also gives primal (fractional) solutions. With

this information one can run a heuristic to produce an integer solution, or one can imbed VOL in a branch and bound code. About “branch and cut” there is an implementation of a cutting plane algorithm for the Max-cut problem, that combines BCP and Vol. No use of the simplex method is required.
At the end VOL has been very useful for accelerating column generation procedures, in particular for Crew Scheduling. The dual solutions given by VOL seem to provide faster convergence than if one uses dual solutions given by the Simplex method.

Chapter 5

Feasibility Pump implemented with QSopt and Coin-Or

5.1 Introduction

In this chapter we will describe how the Feasibility Pump has been implemented with the solvers QSopt and Coin-Or. Instead of re-write the program developed by Fischetti, Lodi and Glover, we have chosen to leave that code intact and to create a library of functions, one for each solver, written in C for QSopt and in C++ for Coin, that captures all the calls to CPLEX routines (from the Feasibility Pump) and performs the same operations using the functions of each software. In this way this partial (because only routines called by FP are implemented) interfaces between CPLEX and Qsopt and between CPLEX and Coin, can be used by other programs written for CPLEX, and if someone wants to run them also with these solvers, it is not necessary to re-write the entire code but only to link the interfaces with the other solver libraries.

Each interface is composed of two files:

- an interface file, with the definition of parameters, the definition of new data types and the declaration of the functions implemented

- a file "ifc_imp" with the implementation of all the functions.

5.1.1 From CPLEX to QSopt

Since all the solver routines deal with so called "problem objects", it has been necessary to define a correspondence between *CPXLPptr* objects and *QSprob* objects. To do this, in "interface.h", after *#include "c:/cygwin/qsopt/qsopt.h"* there are the type definitions:

- *typedef int * CPXENVptr;* — to define CPXENVptr as a pointer to integer
- *typedef QSprob CPXLPptr;* — to define CPXLPptr as QSprob

The reason for which CPXENVptr has been defined as a pointer to integer is very simple. The first routine that must be called for any application is *CPXopenCPLEX()* and this initializes a CPLEX environment when accessing a license for CPLEX and works only if the computer is licensed for Callable Library use. Since if an error occurs (including licensing problems), the value NULL is returned, otherwise the pointer to the CPLEX environment is returned. Now, if *CPXENVptr* has been defined as a pointer to integer, it is sufficient to return that pointer to any integer to create a virtual environment.

At the end, after the declaration of parameters used by CPLEX (and taken from the file "cplex.h" of CPLEX), there is the list of all the functions implemented (using QSopt's routines) into "ifc_imp.c".

Into the original code the only changing to bring is the substitution of the path to include the CPLEX interface ("cplex.h") with that to include "interface.h".

5.1.2 From CPLEX to Coin-Or

To capture and to perform all routines called by Feasibility Pump with Coin-Or solver we have chosen to use the package OSI that is a uniform API (Application Program Interface) for calling math model programming solvers, and programs written to the OSI standard may be linked to any solver with an OSI interface and should produce correct results.

As to QSopt, also with Coin it has been necessary to define a correspondence between problem objects. So, in the file "interface.h", after the declaration of references requested (to OsiClpSolverInterface.hpp, CoinPackedVector.hpp, CoinDistance.hpp and CoinPackedMatrix.hpp), as for QSopt a *CPXENVptr* object has been defined as a pointer to integer while a pointer to *CPXLPptr* has been defined as a pointer to an *OsiSolverInterface* object. Then, after the definition of the structure *move* and *ogg* (used by FP), there is the list of some parameters of CPLEX and the CPLEX functions called by FP and implemented with Coin-Or.

The Feasibility Pump code has been modified to :

- change the path to include the CPLEX interface ("cplex.h") with that to include "interface.h";
- add the instructions *#ifndef LOCBRA_H* and *#define LOCBRA_H*;
- change the name of variable *new* to *new1* to compile correctly the code;
- change the definition of the internal function *compare* (in "LocBra_sub1.c") from *int compare(ogg *b1, ogg *b2)* to *int compare (const void *a, const void *b)*; C++ indeed does not allow implicit conversion from a pointer to void to a pointer of another type, so to compile the code we have redefined *compare* as written above, and we have inserted the explicit conversions:
  - *ogg *b1 = (ogg*)a;*
  - *ogg *b2 = (ogg*)b.*

5.1.3 The files "ifc_imp.c" and "ifc_impc.cpp"

In the following sections there is the description of the libraries of functions to interface the FP written for CPLEX with QSopt and Coin-Or. The routines called by the FP have been grouped in eight classes, depending on their role and utility:

- optimization and result routines
- problem modification routines
- problem querying routines
- file reading and writing routines
- parameter settings and query routines
- utility routines

For each function of each group there is a brief comment to explain its role and its features and then, for QSopt and Coin-Or, there is the description of its implementation indicating, possible unsolved issues.

For some routines of ILOG-CPLEX, indeed, the implementation has been very simple because we have found analogous functions in Coin-Or and QSopt, but for other routines we had to write a lot of lines of code to perform the same operations.

5.2 Optimization and result routines

CPXcreateprob — create a problem object

Synopsis

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

Description *CPXcreateprob()* defines an LP problem name. The argument *env* is the pointer to the CPLEX environment as returned by *CPXopenCPLEX()*, *status_p* a pointer to integer used to return an error code, and *probname* a character string that specifies the name of the problem being created.

QSopt A new pointer to a QSprob problem object is declared and after calling function *QScreate_prob()* the new problem *probname* is created. The routine returns the pointer to the QSopt problem object.

Coin A new object *OsiSolverInterface *temp* is created and then the routine returns the pointer *temp*.

CPXcloneprob — clone a problem

Synopsis

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

Description The routine is used to create a new problem object and to copy all the problem data from an existing problem objective to it. Solution and starting information are not copied.

QSopt After having obtained *nomefile* and *tipofile* (global variables declared in "interface.h" indicating, respectively, name and extension of input file), the function *QSread_prob()* is called to read the problem from input file distinguishing these cases (*copia* is a *CPXLPptr* or *QSprob()* object):

- if *tipofile == NULL*:
  - if *nomefile* ending with ".LP" or ".lp" *copia = QSread_prob(nomefile, "LP");*
  - else if with ".mps" or ".MPS" *copia = QSread_prob(nomefile, "MPS");*
- else if *tipofile* is defined and equal to "LP" or "MPS" it is simply used *QSread_prob()* with correspondent extension.

The routine returns the pointer to *copia* or NULL if the extension is not recognized.

Coin The function *clone()* is called and the routine returns the pointer to cloned problem.

CPXprimopt — primal simplex method

Synopsis

int CPXprimopt (CPXENVptr env, CPXLPptr lpx);

Description The routine may be used to find a solution of a problem object using the primal simplex method. The routine returns zero unless an error occurred during the optimization; note that a zero return value does not necessarily mean that a solution exists.

QSopt The primal simplex method is invoked with the function *QSopt_primal()* and the routine returns zero.

Coin At first the variable *presolve* (global and initialized in "LocBra_fred.c") is analyzed: if *presolve* is equal to 1 *OsiDoPresolveInResolve* parameter is activated using the function *setHintParam()*. Then the function *resolve()* is invoked to perform the primal simplex method. The routine returns zero.

CPXgetobjval – return the solution objective value

Synopsis

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

Description The routine is used to obtain the solution objective value (*objval_p*) and it returns a zero on success, a nonzero if no solution exists.

QSopt The function *QSget_objval()* is called to obtain the solution objective value and then the routine returns zero.

Coin The function *getObjValue()* is called to obtain the solution objective value and then the routine returns zero.

CPXgetx — access the solution values for a range of variables

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 or quadratic program. The beginning and end of the range must be specified and the routine returns a zero on success, and a nonzero if an error occurs.

QSopt After to have obtained the number of columns is invoked the function *QSget_x_array()* that copy the solution vector into an array. Then the array *x* is initialized into a for cycle between *begin* and *end* with the values of the primal variables. The routine returns zero.

Coin Into a for cycle between *begin* and *end* the array *x* is initialized calling the function *getColSolution()* to obtain the solution value of each variable. Then the routine returns zero.

CPXgetdj — access the reduced costs for a range of the variables

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 beginning and end of the range must be specified and the routine returns a zero on success, and a nonzero if an error occurs.

QSopt After to have obtained the number of columns is invoked the function *QSget_rc_array()* that copy the reduced costs into an array. Then the array *dj* is initialized into a for cycle between *begin* and *end* with the reduced costs.

The routine returns zero.

Coin Into a for cycle between *begin* and *end* the array *dj* is initialized calling the function *getReducedCost()* to obtain the reduced cost of each variable. Then the routine returns zero.

CPXgetpi — access the dual values for a range of the constraints

Synopsis

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

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

QSopt After to have obtained the number of columns is invoked the function *QSget_pi_array()* that copy the values of the dual variables into an array. Then the array *pi* is initialized into a for cycle between *begin* and *end* with the dual values. The routine returns zero.

Coin Into a for cycle between *begin* and *end* the array *dj* is initialized calling the function *getRowPrice()* to obtain the dual value of each variable. Then the routine returns zero.

CPXgetphase1cnt — access the number of 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, *CPXgetphase1cnt()* returns the Phase I iteration count. If no solution exists the routine returns the value 0.

QSopt The function *QSget_param()* is called to return the parameter *QS_PRICE_PDANTZIG*. It is not the exact solution but with QSopt is impossible to determine the number of iterations.

Coin The function *getIterationCount()* is called to return the number of iterations to solve the problem.

CPXgetstat — access the solution status

Synopsis

int CPXgetstat (CPXENVptr env, CPXLPptr lpx);

Description The routine is used to access the solution status of the problem optimization and it returns one of parameters defined on each "interface.h".

QSopt The function *QSget_status()* is called to obtain the solution status for QSopt problem and then the correspondent parameter in CPLEX is returned.

Coin The functions *isProvenOptimal()*, *isIterationLimitReached()*, *isProvenPrimalInfeasible()*, *isProvenDualInfeasible()*, *isPrimalObjectiveLimitReached()*, *isDualObjectiveLimitReached()* and *isAbandoned()* are called to determine and to return at least one of possible solution status. If not is returned -1.

CPXmipopt — branch-and-bound method

Synopsis

int CPXmipopt (CPXENVptr env, CPXLPptr lpx);

Description The routine may be used, at any time after a mixed integer program has been created, to find a solution to that problem using branch-and-bound method. *CPXmipopt()* returns a 0 unless it encounters an error. Nonzero values are error codes indicating which type of failure occurred.

QSopt In QSopt this routine can be realized using *QSopt_strongbranch()* to implement the strong-branching rule used by Applegate, Bixby, Chvtal and Cook. The idea is to use strong LP information to determine a choice of branching variable that is likely to increase the LP bound for each of the two children created by the branching operation. To do this, after the initialization of necessary variables, a for cycle between 0 and "(number of columns)-1" is implemented to determine candidates for branching variable using strong-branching: now, obtained the current solution of the LP, the fractional part (t) for each variable is controlled and if $t \geq 0.1$ and $t \leq 0.9$ a candidate is found. Then the function *QSopt_strongbranch()* is called with correct parameters and at the end the routine returns zero.

Coin At first the variable *presolve* (global and initialized in "LocBra_fred.c") is analyzed: if *presolve* is equal to 1 *OsiDoPresolveInResolve* parameter is activated using the function *setHintParam()*. Then the function *branchAndBound()* is invoked to perform the branch-and-bound method. The routine returns zero.

CPXgetmipx — access a range of mixed integer solution 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 beginning and end of the range must be specified and the routine returns a zero on success, and a nonzero if an error occurs.

QSopt The function is called to obtain the solution of the LP problem and then the array x is set with the correct values. The routine returns zero. After to have obtained the number of columns is invoked the function *QSget_x_array()* that copy the solution values into an array. Then the array x is initialized into a for cycle between *begin* and *end* with the solution values. The routine returns zero.

Coin Into a for cycle between *begin* and *end* the array x is initialized calling the function *getColSolution()* to obtain the solution values. Then the routine returns zero.

CPXgetmipobjval — access the mixed integer solution objective value

Synopsis

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

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

QSopt The function *QSget_objval()* is called to obtain the current objective function value and then the routine returns zero.

Coin The function *getObjValue()* is called to obtain the solution objective value and then the routine returns zero.

CPXgetbestobjval — access the best objective function value

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

QSopt The same implementation of *CPXgetmipobjval()*: obviously this is not a perfect solution because the mixed integer solution objective value is not equal to the objective function value of the best remaining node for all cases, but it is impossible to find it with QSopt.

Coin As for QSopt and for the same reason, but calling the function *getObjValue()*.

CPXlpopt — find a solution

Synopsis

int CPXlpopt (CPXENVptr env, CPXLPptr lpx);

Description The routine may be used to find a solution to the problem using one of CPLEX's linear optimizers. The routine returns zero unless an error occurred during the optimization. Note that a zero return value does not necessarily mean that a solution exists.

QSopt The function *QSget_basis()* is called to obtain a copy of the current basis; if a basis is returned, the primal simplex method is invoked with the function *QSopt_primal()*, else the dual method with *QSopt_dual()*. Then the routine returns zero.

Coin At first the variable *presolve* global and initialized in “LocBra_fred.c”) is analyzed: if *presolve* is equal to 1 *OsiDoPresolveInResolve* parameter is activated using the function *setHintParam()*. Then if primal infeasibility has been proven, the dual simplex method is invoked (as for *CPXdualopt()*), else the primal (as for *CPXprimopt()*). The routine returns zero.

CPXdualopt — find a solution

Synopsis

int CPXdualopt (CPXENVptr env, CPXLPptr lpx);

Description The routine may be used to find a solution to the problem using the dual simplex optimizer. The routine returns zero unless an error occurred during the optimization. Note that a zero return value does not necessarily mean that a solution exists.

QSopt The dual simplex method is invoked with the function *QSopt_dual()* and the routine returns zero.

Coin At first the variable *presolve* (global and initialized in “LocBra_fred.c”) is analyzed: if *presolve* is equal to 1 *OsiDoPresolveInResolve* parameter is activated using the function *setHintParam()*. Then the function *resolve()* is invoked to perform the dual simplex method. The routine returns zero.

5.3 Problem modification routines

CPXchgsense — change the sense of a set of constraints

Synopsis

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

Description The routine is used to change the sense of a set of constraints of a problem object. When changing the sense of a row to ranged, *CPXchgsense()* sets the corresponding range value to 0. The routine returns a zero on success, and a nonzero if an error occurs.

QSopt The routine is implemented with a for cycle between *0* and *cnt-1* (number of constraints) and at each iteration, ith entry of sense is read and through the function *QSchange_sense()* the sense of each constraint is updated. Then the routine returns zero.

Coin Into a for cycle between *0* and *cnt-1*, the routine *getRightHandSide()* is called to obtain the right-hand-side coefficient of the current constraint and then *setRowType()* to update the sense of the row. The routine returns zero.

CPXchgobj — change the objective coefficients of a set of variables

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

QSopt Into a cycle for *i* from *0* to *cnt-1* the function *QSchange_objcoef()* is called at each iteration to update the objective coefficient of the variable *indices[i]* with the value *values[i].* At the end the routine returns zero.

Coin Into a cycle for *i* from *0* to *cnt-1* the function *setObjCoeff()* is called at each iteration to update the objective coefficient of the variable *indices[i]* with the value *values[i].* At the end the routine returns zero.

CPXchgrhs — change the right-hand side coefficients of a set of constraints

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

QSopt Into a cycle for *i* from *0* to *cnt-1* the function *QSchange_rhscoef()* is called at each iteration to update the right-hand-side coefficient of the variable *indices[i]* with the value *values[i].* At the end the routine returns zero.

Coin Into a cycle for *i* from *0* to *cnt-1* the function *setRowType()* is called at each iteration to update only the right-hand-side coefficient of the variable *indices[i]* with the value *values[i].* At the end the routine returns zero.

CPXchgprobtype — change the current problem

Synopsis

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

Description The routine is used to change the current problem to a related problem. There are six problem types that can be used: LP, MILP, FIXEDMIL, QP, MIQP and FIXEDMIQP problems.

QSopt The routine returns simply zero because with QSopt is impossible to change the problem.

Coin The routine returns simply zero because with Coin is impossible to change the problem.

CPXdelsetrows — delete a set of rows

Synopsis

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

Description The routine deletes a set of rows but it does not require the rows to be in a contiguous range. 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.

QSopt At first, the function *QSget_rowcount()* is called to obtain the number of rows (*nrows*) of the problem and an array (*delrows*) of size "number of rows + 1" is created. Then into a cycle for *i* from *0* to *nrows-1*, ith entry of *delrows* is initialized to *i* if *delstat[i]* is equal to 1. At the end of cycle if there are rows to be deleted the function *QSdelete_rows()* is invoked using array *delstat.* The routine returns zero.

Coin At first, the function *getNumRows()* is called to obtain the number of rows (*nrows*) of the problem and an array (*delrows*) of size “number of rows + 1” is created. Then into a cycle for *i* from *0* to *nrows-1*, ith entry of *delrows* is initialized to *i* if *delstat[i]* is equal to 1. At the end of cycle if there are rows to be deleted the function *deleteRows()* is invoked using array *delstat*. The routine returns zero.

CPXchgbds — change the upper or lower 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 and the routine returns a zero on success, and a nonzero if an error occurs.

QSopt Into a cycle, for *i* between *0* and *cnt-1* and using the function *QSchange_bound()*:

- if *lu[i]* ==*’U’* upper bound is changed to *bd[i]*;
- if *lu[i]* ==*’L’* lower bound is changed to *bd[i]*;
- if *lu[i]* ==*’B’* upper and lower bounds are both changed to *bd[i]*;

The routine returns zero.

Coin At first, with the functions *getColLower()* and *getColUpper()* current upper and lower bounds are stored into two variables. Then into a cycle, for *i* between *0* and *cnt-1* and using the function *setColBounds()*:

- if *lu[i]==’U’* upper bound is changed to *bd[i]*;
- if *lu[i]==’L’* lower bound is changed to *bd[i]*;
- if *lu[i]==’B’* upper and lower bounds are both changed to *bd[i]*;

The routine returns zero.

CPXaddcols — add columns

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 a problem object. *CPXaddcols()* is very similar to the routine *CPXaddrows()*. The primary difference is that *CPXaddcols()* cannot add coefficients in rows that do not already exist (that is, in rows with index greater than the number returned by *CPXgetnumrows()*); whereas *CPXaddrows()* can add coefficients in columns with index greater than the value returned by *CPXgetnumcols()*, by the use of the *ccnt* argument. Thus, *CPXaddcols()* has no variable *rcnt* and no array *rowname*. The routine returns a zero on success, and a nonzero if an error occurs.

QSopt After to have created the integer array *cmatcnt* of *nzcnt* elements, into a cycle for *k* from *0* to *ccnt-1*, each entry is initialized:

- if $k < ccnt - 1$ then $cmatcnt[k] = (int)(cmatbeg[k+1] - cmatbeg[k])$;
- else $cmatcnt[k] = (int)(ccnt - cmatbeg[k])$;

Now *cmatcnt* (that is an array of length *ncols* where the kth entry specifies the number of non-zeros in the kth column), is defined and calling the function *QSadd_cols()* the new columns are added to the problem. Then the routine returns zero.

Coin This routine is quite complex and following operations are performed into a cycle with *t* from *0* to *cnt-1* (*objcoeff*, *inf* and *sup* are double variables whereas *inizio*, *fine* and *len* are integer variables):

- if $obj == NULL$ then *objcoeff* is set to 0, else to $obj[t]$;
- *inizio* is set to $cmatbeg[t]$;
- if $t == (ccnt - 1)$ then $fine = (nzcnt - 1)$, else $fine = (cmatbeg[t+1] - 1)$;
- so $len = fine - inizio + 1$;
- now, array *index* (of integer) and *val* (of double) are created and initialized in this way (with *q* from 0 to *len-1*):
  - $index[q] = cmatind[inizio + q]$;
  - $val[q] = cmatval[inizio + q]$;
- an object *CoinPackedVector* is created with the constructor $CoinPackedVectorcol(len, index, val)$;
- if $lb == NULL$:
  - then $inf = 0$;
  - else if $lb[t] <= -CPX\_INFBOUND$ then $inf = -(*lpx).getInfinity()$, else $inf = lb[t]$;
- if $ub[t] >= CPX\_INFBOUND$ or $ub == NULL$ then $sup = (*lp).getInfinity()$ else $sup = ub[t]$;
- the new column is added: $(*lpx).addCol(col, inf, sup, objcoeff)$ and the cycle is closed;

Then the routine returns zero.

CPXaddrows — add rows

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 a specified problem object and it returns a zero on success, and a nonzero if an error occurs.

QSopt The routine is implemented in this way:

- if *ccnt* > 0 then begin a cycle (for *t* from *0* to *ccnt-1*) where if *colname !=NULL*:
  - then a new column is added calling the instruction $QSnew\_col(lpx, 0, 0, 0, colname[t])$;
  - else calling $QSnew\_col(lpx, 0, 0, 0, "")$;
- with another cycle (for *k* from *0* to *rcnt-1*):
  - if $k < rcnt - 1$ then $rmatcnt[k] = (int)(rmatbeg[k+1] - rmatbeg[k])$;
  - else $rmatcnt[k] = (int)(rcnt - rmatbeg[k])$;
- now the function *QSadd_rows()* is called using the new parameter *rmatcnt*.

The routine returns zero.

Coin The routine is implemented in this way:

- if *ccnt* > 0 then:
  - current number of rows (*nrows*) is obtained and a new array (*zeric*, of *nrows* elements) is defined;
  - a new object (*col*) is created with the constructor $CoinPackedVectorcol = newCoinPackedVector(nrows, zeric)$;
  - into a for cycle (with *t* from *0* to *ccnt-1*) is called the function $(*lpx).addCol(col, 0, (*lpx).getInfinity(), 0)$;

- current number of columns is obtained and integer variables *inizio*, *fine* and *len* are declared;
- into a for cycle (with t from *0* to *rcnt-1*)
  - $inizio = rmatbeg[t]$;
  - if $t == (rcnt - 1)$ then $fine = (nzcnt - 1)$ *else* $fine = (rmatbeg[t + 1] - 1)$;
  - $len = fine - inizio + 1$;
  - arrays *index* (of integer) and *val* (of double) are defined; integer variable c is initialized to 0;
  - into a for cycle (with q from *0* to *len-1*) if $rmatval[inizio + q]! = 0$ then
    - $index[c] = rmatind[inizio + q]$;
    - $val[c] = rmatval[inizio + q]$;
    - $c + +$;
  - $CoinPackedVectorrow(c, index, val)$;
  - now the function *addRow()* is called using as parameter the object row;

The routine returns zero.

5.4 Problem query routines

CPXgetprobname — access problem name

Synopsis

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

Description The routine *CPXgetprobname()* is used to access the name of the problem set via the call to *CPXcreateprob()*.

QSopt The routine is implemented simply calling *QSget_probname()* and it returns zero.

Coin It is a fake routine because *Clp* does not assign a name to the problem and so the routine returns simply zero.

CPXgetnumrows — retrieve number of rows

Synopsis

int CPXgetnumrows (CPXENVptr env, CPXLPptr lpx);

Description The routine *CPXgetnumrows()* is used to access the number of rows in the constraint matrix, not including the objective function or the bounds constraints on the variables.

Qsopt The routine is implemented simply calling *QSget_rowcount()* and it returns the number of rows.

Coin The routine is implemented simply calling *getNumRows()* and it returns the number of rows.

CPXgetnumcols — retrieve number of columns

Synopsis

int CPXgetnumcols (CPXENVptr env, CPXLPptr lp);

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

QSopt The routine is implemented simply calling *QSget_colcount()* and it returns the number of rows.

Coin The routine is implemented simply calling *getNumCols()* and it returns the number of rows.

CPXgetobjsen — access the objective function sense

Synopsis

int CPXgetobjsen (CPXENVptr env, CPXLPptr lpx);

Description The routine is used to access whether the objective function sense of a problem object is maximization or minimization. The value 1 is returned for minimization, -1 for maximization and 0 if the problem object or environment does not exist.

QSopt The routine returns simply 1 because the solver cannot able to verify the objective sense of the problem. Indeed the sense can be set as a parameter by users but then it is valid for all subsequent problems.

Coin The routine is implemented simply calling *getObjSense()* and it returns the objective function sense.

CPXgetctype — access the types of a range of variables

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 and these are passed through *xctype*. Begin and end of the range must be specified and a zero is returned on success, a nonzero if an error occurs.

QSopt In *QSopt* there is not a similar routine to obtain the same result. So, each variable between *begin* and *end* is analyzed in this way. At first, *QSget_intflags()* is used to determine the integer variables in the problem (information stored in an array of integer where the jth entry is 1 or 0 if the jth variable is integer or not). Then, with a for cycle between *begin* and *end*, are performed these operations:

- if the jth variable is continuous the (j-begin)th entry of *xctype* is set to *'C'*;
- else if the variable is integer, his bounds are controlled with the functions *QSget_bound()* and:
  - if upper bound is equals to 1 and lower to 0 the variable is binary and so (j-begin)th entry of *xctype* is set to *'B'*;
  - else (j-begin)th entry of *xctype* is set to *'I'*;

Coin For each variable between *begin* and *end* is called the routine *isContinuous()* to determine if that is continuous or not: if true the current entry of *xctype* is set to *'C'*; else if the variable is integer, as for *QSopt*, using *getColLower()* and *getColUpper()* his bounds are controlled and then the entry of *xctype* is set to *'B'* or *'I'* (in the same cases as for *QSopt*).

CPXgetobj — access a range of objective function coefficients

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 problem object. The beginning and end of the range must be specified and the routine returns a zero on success, and a nonzero if an error occurs.

QSopt The function *QSget_obj()* returns the coefficients of the objective function as an array of length, at least, the number of columns of the problem. Then *obj* is set with the coefficients (taken between *begin* and *end*) from former array.

Coin In the same way as for *QSopt*, the coefficients of the objective function are obtained by *getObjCoefficients()* and only those from *begin* and *end* are stored into *obj*.

CPXgetub — access a range of upper bounds on the variables

Synopsis

*int CPXgetub (CPXCENVptr env, CPXCLPptr lpx, double *ub, int begin, int end);*

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

QSopt Into a for cycle between *begin* and *end*, using the function *QSget_bound()*, is obtained the upper bound of each variable (*upper*).
If $upper <$ *QS_MAXDOUBLE* (value for infinity used by QSopt):

- if $upper >$ *CPX_INFBOUND* (value for infinity used by CPLEX) the correspondent entry of *ub* is set to *CPX_INFBOUND*;
- else the entry of *ub* is set to *upper*;

If $upper \geq$ *QS_MAXDOUBLE* the current element of *ub* is set to *CPX_INFBOUND*.
Finally the routine returns zero.

Coin Into a for cycle between *begin* and *end*, using the function *getColUpper()*, is obtained the upper bound of each variable (*upper*).
If $upper\ != $ *(*lpx).getInfinity()*(value for infinity used by Clp):

- if *upper* > *CPX_INFBOUND* (value for infinity used by CPLEX) the correspondent entry of *ub* is set to *CPX_INFBOUND*;
- else the entry of *ub* is set to *upper*;

If *upper* ≥ *(*lpx).getInfinity()* the current element of *ub* is set to *CPX_INFBOUND*. Finally the routine returns zero.

CPXgetsense — access the sense

Synopsis

*int CPXgetsense (CPXCENVptr env, CPXCLPptr lpx, char *sense, int begin, int end);*

Description The routine is used to access the sense for a range of constraints in a problem object. The beginning and end of the range must be specified and routine returns a zero on success, and a nonzero if an error occurs. Possible chars are:

- 'L' ≤ constraint;
- 'E' = constraint;
- 'G' ≥ constraint;
- 'R' for ranged constraints; (not used by FP)

QSopt After to have created and initialized few variables and pointers, *QSget_rowcount()* and *QSget_rows()* are called to obtain, respectively, the number of rows and a complete description of the constraints of the problem. Now, since *QSget_rows()* returns an array of length equal to "number of rows" where the ith entry specifies the sense of the ith constraint, into a for cycle between *begin* and *end* it is simple to set the array *sense* with the correct symbols. The routine returns zero.

Coin Into a for cycle between *begin* and *end* the function *getRowSense()* is called to obtain the sense of each row and the array *sense* is initialized. The routine returns zero.

CPXgetrhs — access the right-hand side coefficients

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 problem object. The beginning and end of the range must be specified and the routine returns a zero on success, and a nonzero if an error occurs.

QSopt After to have obtained the number of rows, the function *QSget_rhs()* returns the right-hand-side values of the constraints. Then into a for cycle between *begin* and *end* it is simple to set the array *rhs* with the values. The routine returns zero.

Coin Into a for cycle between *begin* and *end* the function *getRightHandSide()* is called to obtain the right-hand side coefficient of each row and the array *rhs* is initialized. The routine returns zero.

CPXgetcolname — access a range of 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 problem object. The beginning and end of the range, along with the length of the array in which the column names are to be returned, must be specified. The routine returns a zero on success, and a nonzero if an error occurs.

QSopt After to have obtained the number of columns is called the function *QSget_colnames()* that returns an array of strings (*colnames*) specifying the names of the columns (variables). Then *storespace* is stored in *tmp* (integer variable) and into a for cycle between *begin* and *end*:

- the length of each column-name is controlled;
- each entry of *namestore* is initialized with the correspondent string from *colnames*;
- tmp is updated subtracting the length of current column-name.

At the end, *surplus_p* is initialized to *tmp* and if it is negative the routine returns *CPXERR_NEGATIVE_SURPLUS*, else zero.

Coin The routine returns simply zero because Coin does not allow this operation.

CPXgetrows — access a range of rows

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 problem object. The beginning and end of the range, along with the length of the arrays in which the nonzero entries of these rows are to be returned, must be specified. The routine returns a zero on success, and a nonzero if an error occurs.

QSopt The function *QSget_rowcount()* is called to obtain number of rows and then *QSget_rows()* to define *rmatbeg*, *rmatind*, *rmatval* and *nzcnt*. The routine returns zero.

Coin First it is necessary to create the new object *CoinPackedMatrix matrice* with the instruction *const CoinPackedMatrix *matrice = (*lp).getMatrixByRow();* and to initialize a temporary integer variable (*tmp*) to *rmatspace*. Then into for cycle from *begin* to *end*, to obtain the size of the current row is called the function *getVectorSize()*; then if this size is superior to zero:

- *getVectorStarts()* is called to obtain the beginning of ith row
- *rmatbeg[i-begin] = nz;*
- into a new for cycle between zero and the size of the current row, *surplus* is decremented of 1 at each step and if *tmp* > 0:
  - *rmatval[nz] = ((*matrice).getElements())[start+len];*
  - *rmatind[nz] = (int)((*matrice).getIndices())[start+len];*
  - *nz++; tmp − −;*

At the end *nzcnt* is set to *nz* and the routine returns zero.

5.5 File reading and writing routines

CPXreadcopyprob — read a problem object from a file

Synopsis

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

Description The routine *CPXreadcopyprob()* 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", ".mps" or ".MPS".

QSopt After to have read name and extension of the source file, verifying all possible cases, with *QSread_prob()* the problem is read and then routine returns zero on success or 1 if an error occurs.

Coin This solver accept only MPS and so the routine is implemented simply calling *readMps()*; then the routine returns zero.

CPXwriteprob — write problem data

Synopsis

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

Description The routine *CPXwriteprob()* is used to write the current problem object to a file in MPS or LP format. *Filetype* define extension and format of the new file; if *filetype* is NULL the type is inferred from the *filename*. The routine returns a zero on success, and a nonzero if an error occurs.

QSopt After to have read name and extension, calling the routine *QSwrite_prob()* the new file is created with the problem data.

Coin The solver can write only in MPS format with the routine *writeMps()*; it returns zero.

5.6 Parameter setting and query routines

CPXsetintparam — set an integer parameter

Synopsis

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

Description The routine sets the value of a CPLEX parameter of type int and it returns a zero on success, and a nonzero if an error occurs.

QSopt The only parameter fixed is *CPX_PARAM_ITLIM* using the function *QSset_param()*. Other parameters are not fixed because similar parameters are not present in QSopt. The routine returns zero.

Coin The only parameters fixed are *CPX_PARAM_PREIND* and *CPX_PARAM_ITLIM* using the function *setHintParam()*. Other parameters are not fixed because similar parameters are not present in Coin. The routine returns zero.

CPXsetdblparam — set a double parameter

Synopsis

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

Description The routine sets the value of a CPLEX parameter of type double and it returns a zero on success, and a nonzero if an error occurs.

QSopt The routine returns simply zero because because similar parameters are not present in QSopt.

Coin As for QSopt and for the same reason.

5.7 Utility routines

CPXopenCPLEX — initializes a CPLEX environment

Synopsis

*CPXENVptr CPXopenCPLEX (int *status_p);*

Description The routine *CPXopenCPLEX()* initializes a CPLEX environment when accessing a license for CPLEX and it works only if the computer is licensed for Callable Library use. Since if an error occurs (including licensing problems), the value NULL is returned and the reason for the error is returned in the variable **status_p*, else it returns the pointer to the CPLEX environment.

QSopt In "ifc_imp.c" is defined an integer with value 1 and the routine returns a pointer to this integer (because *CPXENVptr* has been defined as a pointer to integer in "interface.h").

Coin In "ifc_imp.cpp" is defined an integer with value 1 and the routine returns a pointer to this integer (because *CPXENVptr* has been defined as a pointer to integer in "interface.h").

CPXflushstdchannels — flushes the output

Synopsis

int CPXflushstdchannels (CPXENVptr env);

Description This routine flushes the output buffers of the four standard channels *cpxresults*, *cpxwarning*, *cpxerror*, and *cpxlog*. It returns a zero on success, and a nonzero if an error occurs.

QSopt The routine returns simply zero because QSopt does not allow this operation.

Coin As for QSopt and for the same reason.

CPXgetcallbackinfo — access information

Synopsis

*int CPXgetcallbackinfo (CPXENVptr env, void *cbdata, int wherefrom, int which-info, void *result_p);*

Description The routine is used to access information about the current optimization process and it is the only routine that can access optimization status information from within a user-written callback function. It is also the only Callable Library routine that may be called from within a user-written callback function, and indeed, may only be called from the callback function. The routine returns a zero on success, and a nonzero if an error occurs.

QSopt The routine retrieve simply objective function value and returns zero.

Coin As for QSopt.

CPXcloseCPLEX — close CPLEX environment

Synopsis

*int CPXcloseCPLEX (CPXENVptr *env_p);*

Description The routine closes CPLEX environment, release the license and it returns a zero on success, a nonzero if an error occurs.

QSopt The routine returns simply zero.

Coin The routine returns simply zero.

Chapter 6

Computational results

6.1 Performance

This chapter is dedicated to the comparison of the performance of Feasibility Pump through the solvers: ILOG-CPLEX 8.1, Coin-Or (updated in September 2004) and QSopt 1.0 . The results reported on the following two tables concern to problems of set "MIPLIB 2003" and to some instances provided by Fischetti and Lodi [7, 8, 9, 15, 18, 19, 22, 1] . The tests have been obtained with the same FP configuration file (param.txt) and command line, for all the problems and for each solver.
In detail, in the command line parameters have been set to:

- totalTime (total time limit) = 5000
- T (size of the set to update) = 20
- maxIter (max number of internal iterations) = 100000
- TT (tabu tenure) = 0
- wh (0 no heuristic, 1 RINS, 2 DIST) = 0
- SAT (0 NO, 1 YES, 2 STOP at first feasible solution) = 2.

In this way the focus of the tests has been to measure the capability of the solvers (using the FP) to converge to an initial feasible solution, stopping the solvers as soon as the first feasible solution was found.

As to file “param.txt”, we have tested the solvers with `presolve` activated: with this parameter set to 1, the preprocessing/presolve is activated for ILOG-CPLEX and Coin-Or, but not for QSopt that does not allow this process to reduce problems size and therefore total run time.

Then into the files “LocBra.h” and “LocBra.hpp”, we have enabled the line *#defineOPT*. This function allows all iterations the solvers needs to perform the simplex algorithm, 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 a stall situation occurs but the solver does not notice it, the FP algorithm enter an infinite loop.

The two tables report, for each problem and for each solver, the value of the first feasible solution found (“value”), the number of iterations performed (“nIT”) and the corresponding computing time (“time”) in seconds. In case of failure, “N/A” is reported if a solution has not been found and “S/F” in case of segmentation fault (core dumped) error. Computing times are expressed in CPU seconds and refer to a Pentium III 933 MHz with 256 MByte of RAM memory. All the problems have been converted in MPS format because Coin-Or does not recognized LP format; to convert the instances we have used ILOG-CPLEX Interactive Optimizer.

6.2 Comments on results

Own order of business has been 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 with all the solvers we used. Indeed, on 81 problems tested, with ILOG-CPLEX a first feasible solution has not been found in only 3 instances, with Coin in 10, and with QSopt in 11 problems. In detail, ILOG-CPLEX has reached the time limit in 3 cases, while Coin-Or and QSopt, respectively 9 and 4 times. For other problems for which Feasibility Pump has not found a feasible solution, the execution is terminated with a segmentation fault error (core dumped) in just one case for Coin-Or, and 7 cases for QSopt.

Also interesting is the comparison of the quality of the FP solution found with the solvers: for 25 problems, the best solution has been found by ILOG-CPLEX solver, while QSopt and Coin-Or ranked first for, respectively, 19 and 17 instances (for remaining problems the solvers found the same solution values).

	ILOG-CPLEX			Coin-Or			QSopt		
Name	value	nIT	time	value	nIT	time	value	nIT	time
10teams	992.00	53	22.6	1060.00	113	88.5	988.00	120	99.5
A1C1S1	18377.24	5	16.3	18015.92	3	19.3	16815.62	5	55.7
aflow30a	4398.00	16	0.4	4033.00	6	2.3	4254.00	7	1.1
aflow40b	6859.00	7	1.6	7137.00	12	8.8	4893.00	5	10.3
air04	58950.00	6	239.4	67533.00	16	1047.9	59199.00	2	46.7
air05	29937.00	2	33.1	31744.00	12	210.4	34167.00	6	66.3
cap6000	-2354320.00	2	1.7	-2354320.00	2	4.5	-2354320.00	2	1.1
dano3mip	2882022.00	2	256.7	1000.00	9	2875.7	756.62	1	2755.4
danoint	77	3	0.7	77	2	3.1	77	3	3.0
ds	N/A	18	5000.0	N/A	15	5000.0	N/A	22	5000.0
fast0507	181.00	4	117.8	183.00	3	483.8	188.00	3	370.4
fiber	1911617.79	2	0.1	1324277.64	2	0.9	1759273.96	2	0.1
fixnet6	9131.00	4	0.1	9194.00	2	0.9	10839.00	4	0.3
glass4	5600050250.00	124	1.4	12700177300.00	3	0.9	N/A	S/F	?
harp2	-43856974.00	654	15.8	-45846186.00	2765	1169.4	N/A	S/F	?
liu	6262.00	0	0.3	6450.00	0	0.5	5930.00	0	0.3
markshare1	1114.00	9	0.1	852.00	1	0.1	268.00	0	0.0
markshare2	1738.00	8	0.1	1507.00	1	0.4	585.00	0	0.1
mas74	52429700.59	1	0.1	18003.39	1	0.3	38123.74	2	0.1
mas76	194527859.06	1	0.0	47541.14	1	0.4	62310.32	1	0.1
misc07	3700.00	29	0.6	4415.00	41	11.7	4130.00	102	5.1
mkc	-164.56	2	1.0	-85.85	2	2.0	-234.69	2	53.7
modglob	35147088.88	0	0.0	35147088.88	0	0.2	35147088.88	0	0.1
momentum1	462127.33	502	3316.7	N/A	226	5000.0	N/A	39	5000.0
net12	337.00	346	170.5	N/A	S/F	?	N/A	S/F	?
nsrand_ipx	340800.00	3	2.2	412000.00	2	6.2	313760.00	3	2.9
nw04	19882.00	1	7.8	19882.00	1	97.6	19812.00	1	50.4
opt1217	-12	0	0.1	-14	0	0.3	-14	0	0.1
p2756	N/A	79552	5000.0	N/A	9299	5000.0	N/A	S/F	?
pk1	57.00	0	0.1	86.00	0	0.1	35.00	0	0.1
pp08a	11150.00	2	0.1	10420.00	3	1.3	12970.00	3	0.1
pp08aCUTS	10940.00	2	0.1	10700.00	2	0.7	11600.00	3	0.1
protfold	-10.00	367	2341.6	N/A	125	5000.0	N/A	77	5000.0
qiu	389.36	3	1.1	386.53	3	2.8	422.93	3	2.9
rd-rplusc-21	N/A	378	5000.0	N/A	4	5000.0	N/A	16	5000.0
set1ch	76951.50	2	0.2	107554.50	2	1.0	87450.75	2	0.2
seymour	452.00	9	13.4	457.00	8	17.9	462.00	10	41.2
sp97ar	1398705728.00	6	13.6	1397935023.68	4	88.2	1175257518.72	4	23.2
swath	19221.42	49	7.8	45119.00	556	529.0	N/A	S/F	?
t1717	826848.00	42	2523.9	N/A	6	5000.0	N/A	S/F	?
tr12-30	277218.00	9	0.5	263011.00	9	3.9	250302.00	9	1.1
van	8.21	4	693.7	8.91	2	1071.6	7.11	3	1204.3
vpm2	19.25	3	0.1	22.5	3	1.3	22.00	2	0.1

Table 6.1: Performance of Feasibility Pump applied to ILOG-CPLEX, Coin-Or and QSopt.

These numbers are quite comparable, also taking into account that ILOG-CPLEX is a commercial product while QSopt and Coin-Or are open-source software.

A different scenario concerns computing times. Looking at the tables, it is very simple to observe that for almost the entire set of problems, the Feasibility Pump implemented with ILOG-CPLEX finds a feasible solution earlier than those implemented with QSopt and Coin-Or. Between those with Qsopt and Coin-Or, instead, it is not too simple to establish which is the fastest one. Excluding the

	ILOG-CPLEX			COIN-OR			QSopt		
Name	value	nIT	time	value	nIT	time	value	nIT	time
biella1	3537959.54	5	28.2	3536921.85	6	178.4	3436393.28	6	1435.3
dc1c	27348312.19	4	60.2	26305768.12	5	526.6	6104043.29	4	4448.7
dc1l	8256022.49	5	335.9	N/A	0	5000.0	N/A	S/F	?
dolom1	298684615.17	7	94.0	1040541381.20	4	874.7	N/A	4	1501.3
siena1	104004996.99	5	281.5	N/A	2	5000.0	252696707.35	5	4938.8
trento1	356179003.01	2	57.2	177331065.02	3	929.3	256667686.02	3	199.8
rail507	178.00	2	175.8	187.00	3	523.1	178.00	3	461.8
rail2536c	715.00	4	85.4	730.00	3	618.7	719.00	3	241.9
rail2586c	1007.00	5	289.4	1002.00	5	709.0	1023.00	5	814.9
rail4284c	1124.00	3	4343.7	N/A	0	5000.0	1137.00	3	2998.5
rail4872	1614.00	5	993.1	N/A	2	5000.0	1628.00	4	2728.8
A2C1S1	19879.93	5	17.2	15222.13	4	17.5	17897.46	5	49.2
B1C1S1	38530.65	7	23.2	46705.76	5	24.0	40908.09	6	53.0
B2C1S1	48279.95	6	18.8	56377.22	6	27.5	35118.52	5	63.2
sp97ic	1280793707.52	3	8.6	1134148364.64	2	34.5	907630990.40	2	15.1
sp98ar	988402511.36	4	13.7	1056785441.12	5	129.6	10020402074.40	9	43.2
sp96ic	959924716.00	3	7.0	1094092452.80	2	21.5	885483915.04	4	17.9
blp-ar98	25094.03	161	72.7	24014.3	204	925.2	N/A	S/F	?
blp-ic97	7874.87	4	2.4	7911.40	22	46.9	7072.96	13	10.2
blp-ic98	14848.96	6	4.8	13614.59	8	37.8	17399.89	19	77.7
blp-ir98	6208.74	4	1.4	6580.38	3	6.8	9476.30	9	4.3
CMS750_4	606.00	131	67.9	555.00	140	483.0	679.00	182	2384.4
berlin_5_8_0	79.00	10	0.5	77.00	13	6.4	77.00	15	1.7
railway_8_1_0	440.00	13	1.0	435.00	17	10.1	450.00	27	5.3
usAbbrv.8.25_70	164.00	34	2.6	158.00	31	20.9	159.00	34	9.0
bg512142	120738665.00	0	0.5	120738665.00	0	1.1	120738665.00	0	1.7
dg012142	153406945.50	0	3.6	153406945.50	0	4.9	153409621.50	0	20.0
manpower1	8.00	66	106.2	11.00	125	629.8	9.00	51	2113.2
manpower2	7.00	148	507.5	6.00	33	608.8	N/A	79	5000.0
manpower3	6.00	49	156.6	7.00	26	492.1	6.00	36	1860.2
manpower3a	6.00	73	236.8	10.00	157	2579.6	7.00	55	3242.9
manpower4	7.00	192	322.4	7.00	28	375.8	8.00	46	2800.9
manpower4a	7.00	53	237.4	10.00	127	2568.5	7.00	42	2622.3
ljb2	7.24	0	0.2	7.24	0	0.5	7.24	0	0.3
ljb7	8.61	0	2.0	8.61	0	4.0	8.61	0	12.1
ljb9	9.48	0	2.6	9.48	0	5.7	9.48	0	16.7
ljb10	7.31	0	3.6	7.31	0	9.8	7.31	0	23.8
ljb12	6.20	0	2.6	6.20	0	6.1	6.20	0	16.7

Table 6.2: Performance of Feasibility Pump applied to ILOG-CPLEX, Coin-Or and QSopt.

instances for which the solvers required less than 5 seconds, the Feasibility Pump implemented with Coin-Or was faster than that with QSopt in 28 cases, while in 19 cases the FP with QSopt was faster than that with Coin-Or. Excluding ILOG-CPLEX, using the FP the faster solver seems to be Coin-Or, but it is necessary to notice that for the problems solved in less than 5 seconds, the FP with QSopt was faster than that with Coin-Or.

Indeed, the tables 6.3 and 6.4 report the ratio of the average times per iteration of the FP with CPLEX is normalized to 1.0 for all instances (the higher the ratio, the slower the FP implementation with respect to the one based on CPLEX). The average ratio of FP with Coin-Or is 8.45, while for QSopt is 8.13: hence, on

	ILOG-CPLEX		Coin-Or		QSopt	
Name	value	ratio	value	ratio	value	ratio
10teams	992.00	(1.0)	1060.00	1.8	988.00	1.9
A1C1S1	18377.24	(1.0)	18015.92	2.0	16815.62	3.4
aflow30a	4398.00	(1.0)	4033.00	15.3	4254.00	6.3
aflow40b	6859.00	(1.0)	7137.00	3.2	4893.00	9.0
air04	58950.00	(1.0)	67533.00	1.6	59199.00	0.6
air05	29937.00	(1.0)	31744.00	1.1	34167.00	0.7
cap6000	-2354320.00	(1.0)	-2354320.00	2.6	-2354320.00	0.6
dano3mip	2882022.00	(1.0)	1000.00	21.8	756.62	21.5
danoint	77	(1.0)	77	6.6	77	4.3
ds	N/A	(1.0)	N/A	1.2	N/A	0.8
fast0507	181.00	(1.0)	183.00	5.5	188.00	4.2
fiber	1911617.79	(1.0)	1324277.64	9.0	1759273.96	1.0
fixnet6	9131.00	(1.0)	9194.00	18.0	10839.00	3.0
glass4	5600050250.00	(1.0)	12700177300.00	26.6	N/A	?
harp2	-43856974.00	(1.0)	-45846186.00	17.5	N/A	?
liu	6262.00	(1.0)	6450.00	1.6	5930.00	1.0
markshare1	1114.00	(1.0)	852.00	9.0	268.00	0.0
markshare2	1738.00	(1.0)	1507.00	32.0	585.00	0.0
mas74	52429700.59	(1.0)	18003.39	3.0	38123.74	0.5
mas76	194527859.06	(1.0)	47541.14	4.0	62310.32	1.0
misc07	3700.00	(1.0)	4415.00	13.8	4130.00	2.4
mkc	-164.56	(1.0)	-85.85	2.0	-234.69	53.7
modglob	35147088.88	(1.0)	35147088.88	2.0	35147088.88	1.0
momentum1	462127.33	(1.0)	N/A	3.3	N/A	19.4
nsrand_ipx	340800.00	(1.0)	412000.00	4.2	313760.00	1.3
nw04	19882.00	(1.0)	19882.00	12.5	19812.00	6.5
opt1217	-12	(1.0)	-14	3.0	-14	1.0
p2756	N/A	(1.0)	N/A	8.6	N/A	?
pk1	57.00	(1.0)	86.00	1.0	35.00	1.0
pp08a	11150.00	(1.0)	10420.00	13.0	12970.00	1.0
pp08aCUTS	10940.00	(1.0)	10700.00	7.0	11600.00	0.7
protfold	-10.00	(1.0)	N/A	6.3	N/A	10.2
qiu	389.36	(1.0)	386.53	2.5	422.93	2.6
rd-rplusc-21	N/A	(1.0)	N/A	94.5	N/A	23.6
set1ch	76951.50	(1.0)	107554.50	5.0	87450.75	1.0
seymour	452.00	(1.0)	457.00	1.5	462.00	2.8
sp97ar	1398705728.00	(1.0)	1397935023.68	9.7	1175257518.72	2.6
swath	19221.42	(1.0)	45119.00	6.0	N/A	?
t1717	826848.00	(1.0)	N/A	13.9	N/A	?
tr12-30	277218.00	(1.0)	263011.00	7.8	250302.00	2.2
van	8.21	(1.0)	8.91	3.1	7.11	2.3
vpm2	19.25	(1.0)	22.5	13.0	22.00	6.0

Table 6.3: Performance of Feasibility Pump applied to ILOG-CPLEX, Coin-Or and QSopt.

average, an iteration of 1 second of the FP with CPLEX requires 8.45 seconds with Coin-Or and 8.13 with QSopt.

About problems for which FP did found any feasible solution with all the solvers used, it is necessary to make considerations, especially for p2756.

As written in [25], p2756 is a pathological instance for FP, 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.

	ILOG-CPLEX		COIN-OR		QSopt	
Name	value	ratio	value	ratio	value	ratio
biella1	3537959.54	(1.0)	3536921.85	5.3	3436393.28	42.4
dc1c	27348312.19	(1.0)	26305768.12	7.0	6104043.29	73.9
dolom1	298684615.17	(1.0)	1040541381.20	16.3	N/A	27.9
siena1	104004996.99	(1.0)	N/A	44.4	252696707.35	17.6
trento1	356179003.01	(1.0)	177331065.02	10.8	256667686.02	2.3
rail507	178.00	(1.0)	187.00	2.0	178.00	1.8
rail2536c	715.00	(1.0)	730.00	9.7	719.00	3.8
rail2586c	1007.00	(1.0)	1002.00	2.5	1023.00	2.8
rail4284c	1124.00	(1.0)	N/A	?	1137.00	0.7
rail4872	1614.00	(1.0)	N/A	12.6	1628.00	3.4
A2C1S1	19879.93	(1.0)	15222.13	1.3	17897.46	2.9
B1C1S1	38530.65	(1.0)	46705.76	1.5	40908.09	2.7
B2C1S1	48279.95	(1.0)	56377.22	1.5	35118.52	4.0
sp97ic	1280793707.52	(1.0)	1134148364.64	6.0	907630990.40	2.6
sp98ar	988402511.36	(1.0)	1056785441.12	7.6	10020402074.40	1.4
sp96ic	959924716.00	(1.0)	1094092452.80	4.6	885483915.04	1.9
blp-ar98	25094.03	(1.0)	24014.3	10.0	N/A	?
blp-ic97	7874.87	(1.0)	7911.40	3.6	7072.96	1.3
blp-ic98	14848.96	(1.0)	13614.59	5.9	17399.89	5.1
blp-ir98	6208.74	(1.0)	6580.38	6.5	9476.30	1.4
CMS750_4	606.00	(1.0)	555.00	6.7	679.00	25.3
berlin_5_8_0	79.00	(1.0)	77.00	9.8	77.00	2.3
railway_8_1_0	440.00	(1.0)	435.00	7.7	450.00	2.6
usAbbrv.8.25_70	164.00	(1.0)	158.00	8.8	159.00	3.5
bg512142	120738665.00	(1.0)	120738665.00	2.2	120738665.00	3.4
dg012142	153406945.50	(1.0)	153406945.50	1.4	153409621.50	5.6
manpower1	8.00	(1.0)	11.00	5.9	9.00	19.8
manpower2	7.00	(1.0)	6.00	5.4	N/A	18.5
manpower3	6.00	(1.0)	7.00	5.9	6.00	16.2
manpower3a	6.00	(1.0)	10.00	5.1	7.00	18.2
manpower4	7.00	(1.0)	7.00	8.0	8.00	36.3
manpower4a	7.00	(1.0)	10.00	4.5	7.00	13.9
ljb2	7.24	(1.0)	7.24	2.5	7.24	1.5
ljb7	8.61	(1.0)	8.61	2.0	8.61	6.0
ljb9	9.48	(1.0)	9.48	2.2	9.48	6.4
ljb10	7.31	(1.0)	7.31	2.7	7.31	6.6
ljb12	6.20	(1.0)	6.20	2.3	6.20	6.4

Table 6.4: Performance of Feasibility Pump applied to ILOG-CPLEX, Coin-Or and QSopt.

About `ds` and `rd-rplusc-21`, instead, the Feasibility Pump did not find a solution with all the solvers used, probably because these are instances particularly complex and the time limit of 5000 seconds is not sufficient on our hardware. The same considerations can be done about the problems that Coin-Or and QSopt cannot solve and for which the time limit has been reached before finding a solution.

In the end, especially for QSopt, but also for Coin-Or in one case, there are problems for which the execution terminates without finding a solution and before to reach time limit: with these instances a segmentation fault (core dumped) occurs, because the programs tries to access memory locations that have not been allocated by the program. The segmentation fault, is not likely to be caused by bug of our code, because only for few problems this kind of error occurs, while the routine implemented into the files "ifc_imp" are (almost all) called for each instance. On the other hand, both Coin-Or and QSopt are less sophisticated than ILOG-CPLEX, hence the presence of a bug is possible.

Chapter 7

Conclusion

The results obtained by the Feasibility Pump through the two open-source software have been very satisfactory: for all solvers FP has been able to find a first feasible solution in the large majority of instances (only with QSopt the FP encountered some problems). Obviously the best performance have been obtained with CPLEX, the commercial solver that is, in absolute, one of the most efficient. Between Coin-Or and QSopt, we would suggest the use of Coin-Or, because it has been able to find a solution for more instances respect to QSopt. Examining carefully the entire set of results, FP obtained excellent results with all the solvers. Looking to the problems for which the computing time is lower than 5 seconds, the FP performance with QSopt is very similar to that with CPLEX. So, after CPLEX that is at the first place, the FP with Coin-Or and QSopt have their merits and faults. The FP with Coin-Or can find a solution on more instances than that with Qsopt, but the FP with Qsopt is, perhaps, faster on "simple" problems.

As to the libraries of functions, it must be stressed that the interfaces are not complete, because only routines called by FP have been implemented. Nevertheless, the current libraries can be applied on a large number of programs because the routines implemented may be sufficient in many cases.

Finally, the libraries we have developed could be very useful because they can be applied to all the software and programs based and written for CPLEX which does not need a too powerful solver. In this case, without rewriting any line of code, one can immediately use one of this free software.

Appendix A

Source code: “from ILOG-CPLEX to QSopt”

A.1 interface.h

```
#include "c:/cygwin/qsopt/qsopt.h"

typedef int * CPXENVptr;

typedef QSprob CPXLPptr;

#define CPXoptimize     CPXlpopt

#define CPX_ALG_NONE                    -1
#define CPX_ALG_AUTOMATIC                0
#define CPX_ALG_PRIMAL                   1
#define CPX_ALG_DUAL                     2
#define CPX_ALG_NET                      3
#define CPX_ALG_BARRIER                  4
#define CPX_ALG_SIFTING                  5
#define CPX_ALG_CONCURRENT               6
#define CPX_ALG_BAROPT                   7
#define CPX_ALG_PIVOTIN                  8
#define CPX_ALG_PIVOTOUT                 9
#define CPX_ALG_PIVOT                   10
#define CPX_ALG_ANY                      BIGINT

#define CPX_MAX                         -1
#define CPX_MIN                          1

#define CPXPROB_LP  0
#define CPXPROB_MILP  1
#define CPXPROB_FIXEDMILP  3
#define CPXPROB_QP  5
#define CPXPROB_MIQP 7
#define CPXPROB_FIXEDMIQP 8

#define CPX_INFBOUND  1.0E+20

#define CPXERR_NEGATIVE_SURPLUS 1207

#define CPX_AT_LOWER                     0
#define CPX_BASIC                        1
#define CPX_AT_UPPER                     2
```

```
#define CPX_FREE_SUPER                   3

//per wherefrom
#define CPX_CALLBACK_PRIMAL               1
#define CPX_CALLBACK_DUAL                 2
#define CPX_CALLBACK_NETWORK              3
#define CPX_CALLBACK_PRIMAL_CROSSOVER     4
#define CPX_CALLBACK_DUAL_CROSSOVER       5
#define CPX_CALLBACK_BARRIER              6
#define CPX_CALLBACK_PRESOLVE             7
#define CPX_CALLBACK_QPBARRIER            8
#define CPX_CALLBACK_QPSIMPLEX            9

#define CPX_CALLBACK_MIP                101
#define CPX_CALLBACK_MIP_BRANCH         102
#define CPX_CALLBACK_MIP_NODE           103
#define CPX_CALLBACK_MIP_HEURISTIC      104
#define CPX_CALLBACK_MIP_SOLVE          105
#define CPX_CALLBACK_MIP_CUT            106
#define CPX_CALLBACK_MIP_PROBE          107
#define CPX_CALLBACK_MIP_FRACCUT        108
#define CPX_CALLBACK_MIP_DISJCUT        109
#define CPX_CALLBACK_MIP_FLOWMIR        110
#define CPX_CALLBACK_MIP_INCUMBENT      111
#define CPX_CALLBACK_MIP_DELETENODE     112

#define CPX_CALLBACK_INFO_BEST_INTEGER         101

/* MIP Parameter numbers */
#define CPX_PARAM_BRDIR                 2001
#define CPX_PARAM_BTTOL                 2002
#define CPX_PARAM_CLIQUES               2003
#define CPX_PARAM_COEREDIND             2004
#define CPX_PARAM_COVERS                2005
#define CPX_PARAM_CUTLO                 2006
#define CPX_PARAM_CUTUP                 2007
#define CPX_PARAM_EPAGAP                2008
#define CPX_PARAM_EPGAP                 2009
#define CPX_PARAM_EPINT                 2010
#define CPX_PARAM_HEURISTIC             2011
#define CPX_PARAM_MIPDISPLAY            2012
#define CPX_PARAM_MIPINTERVAL           2013
#define CPX_PARAM_MIPTHREADS            2014
#define CPX_PARAM_INTSOLLIM             2015
#define CPX_PARAM_NODEFILEIND           2016
#define CPX_PARAM_NODELIM               2017
#define CPX_PARAM_NODESEL               2018
#define CPX_PARAM_OBJDIF                2019
#define CPX_PARAM_MIPORDIND             2020
#define CPX_PARAM_RELOBJDIF             2022
#define CPX_PARAM_STARTALG              2025
#define CPX_PARAM_SUBALG                2026
#define CPX_PARAM_TRELIM                2027
#define CPX_PARAM_VARSEL                2028
#define CPX_PARAM_BNDSTRENIND           2029
#define CPX_PARAM_HEURFREQ              2031
#define CPX_PARAM_MIPORDTYPE            2032
#define CPX_PARAM_CUTSFACTOR            2033
#define CPX_PARAM_RELAXPREIND           2034
#define CPX_PARAM_MIPSTART              2035
#define CPX_PARAM_PRESLVND              2037
#define CPX_PARAM_BBINTERVAL            2039
#define CPX_PARAM_FLOWCOVERS            2040
#define CPX_PARAM_IMPLBD                2041
#define CPX_PARAM_PROBE                 2042
```

```
#define CPX_PARAM_GUBCOVERS           2044
#define CPX_PARAM_STRONGCANDLIM       2045
#define CPX_PARAM_STRONGITLIM         2046
#define CPX_PARAM_STRONGTHREADLIM     2047
#define CPX_PARAM_FRACCAND            2048
#define CPX_PARAM_FRACCUTS            2049
#define CPX_PARAM_FRACPASS            2050
#define CPX_PARAM_FLOWPATHS           2051
#define CPX_PARAM_MIRCUTS             2052
#define CPX_PARAM_DISJCUTS            2053
#define CPX_PARAM_AGGCUTLIM           2054
#define CPX_PARAM_MIPCBREDLP          2055
#define CPX_PARAM_CUTPASS             2056
#define CPX_PARAM_MIPEMPHASIS         2058
#define CPX_PARAM_SYMMETRY            2059
#define CPX_PARAM_DIVETYPE            2060

/* CPLEX Parameter numbers */

#define CPX_PARAM_ADVIND              1001
#define CPX_PARAM_AGGFILL             1002
#define CPX_PARAM_AGGIND              1003
#define CPX_PARAM_BASINTERVAL         1004
#define CPX_PARAM_CFILEMUL            1005
#define CPX_PARAM_CLOCKTYPE           1006
#define CPX_PARAM_CRAIND              1007
#define CPX_PARAM_DEPIND              1008
#define CPX_PARAM_DPRIIND             1009
#define CPX_PARAM_PRICELIM            1010
#define CPX_PARAM_EPMRK               1013
#define CPX_PARAM_EPOPT               1014
#define CPX_PARAM_EPPER               1015
#define CPX_PARAM_EPRHS               1016
#define CPX_PARAM_FASTMIP             1017
#define CPX_PARAM_IISIND              1018
#define CPX_PARAM_SIMDISPLAY          1019
#define CPX_PARAM_ITLIM               1020
#define CPX_PARAM_ROWREADLIM          1021
#define CPX_PARAM_NETFIND             1022
#define CPX_PARAM_COLREADLIM          1023
#define CPX_PARAM_NZREADLIM           1024
#define CPX_PARAM_OBJLLIM             1025
#define CPX_PARAM_OBJULIM             1026
#define CPX_PARAM_PERIND              1027
#define CPX_PARAM_PERLIM              1028
#define CPX_PARAM_PPRIIND             1029
#define CPX_PARAM_PREIND              1030
#define CPX_PARAM_REINV               1031
#define CPX_PARAM_REVERSEIND          1032
#define CPX_PARAM_RFILEMUL            1033
#define CPX_PARAM_SCAIND              1034
#define CPX_PARAM_SCRIND              1035
#define CPX_PARAM_SIMTHREADS          1036
#define CPX_PARAM_SINGLIM             1037
#define CPX_PARAM_SINGTOL             1038
#define CPX_PARAM_TILIM               1039
#define CPX_PARAM_XXXIND              1041
#define CPX_PARAM_PREDUAL             1044
#define CPX_PARAM_ROWGROWTH           1046
#define CPX_PARAM_COLGROWTH           1047
#define CPX_PARAM_NZGROWTH            1048
#define CPX_PARAM_EPOPT_H             1049
#define CPX_PARAM_EPRHS_H             1050
#define CPX_PARAM_PREPASS             1052
#define CPX_PARAM_DATACHECK           1056
```

```
#define CPX_PARAM_REDUCE               1057
#define CPX_PARAM_PRELINEAR            1058
#define CPX_PARAM_LPMETHOD             1062
#define CPX_PARAM_QPMETHOD             1063
#define CPX_PARAM_WORKDIR              1064
#define CPX_PARAM_WORKMEM              1065
#define CPX_PARAM_PRECOMPRESS          1066
#define CPX_PARAM_THREADS              1067
#define CPX_PARAM_SIFTDISPLAY          1076
#define CPX_PARAM_SIFTALG              1077
#define CPX_PARAM_SIFTITLIM            1078

/* Values returned for 'stat' by solution () */

#define CPX_STAT_OPTIMAL                  1
#define CPX_STAT_UNBOUNDED                2
#define CPX_STAT_INFEASIBLE               3
#define CPX_STAT_INForUNBD                4
#define CPX_STAT_OPTIMAL_INFEAS           5
#define CPX_STAT_NUM_BEST                 6
#define CPX_STAT_ABORT_IT_LIM            10
#define CPX_STAT_ABORT_TIME_LIM          11
#define CPX_STAT_ABORT_OBJ_LIM           12
#define CPX_STAT_ABORT_USER              13

/* Solution type return values from CPXsolninfo() */
#define CPX_NO_SOLN       0
#define CPX_BASIC_SOLN    1
#define CPX_NONBASIC_SOLN  2

/* Values of presolve 'stats' for columns and rows */
#define CPX_PRECOL_LOW               -1
#define CPX_PRECOL_UP                -2
#define CPX_PRECOL_FIX               -3
#define CPX_PRECOL_AGG               -4
#define CPX_PRECOL_OTHER             -5
#define CPX_PREROW_RED               -1
#define CPX_PREROW_AGG               -2
#define CPX_PREROW_OTHER             -3

/* MIP Problem status codes */

#define CPXMIP_OPTIMAL                 101
#define CPXMIP_OPTIMAL_TOL             102
#define CPXMIP_INFEASIBLE              103
#define CPXMIP_SOL_LIM                 104
#define CPXMIP_NODE_LIM_FEAS           105
#define CPXMIP_NODE_LIM_INFEAS         106
#define CPXMIP_TIME_LIM_FEAS           107
#define CPXMIP_TIME_LIM_INFEAS         108
#define CPXMIP_FAIL_FEAS               109
#define CPXMIP_FAIL_INFEAS             110
#define CPXMIP_MEM_LIM_FEAS            111
#define CPXMIP_MEM_LIM_INFEAS          112
#define CPXMIP_ABORT_FEAS              113
#define CPXMIP_ABORT_INFEAS            114
#define CPXMIP_OPTIMAL_INFEAS          115
#define CPXMIP_FAIL_FEAS_NO_TREE       116
#define CPXMIP_FAIL_INFEAS_NO_TREE     117
#define CPXMIP_UNBOUNDED               118
#define CPXMIP_INForUNBD               119

char *nomefile;
char *tipofile;
```

```
CPXENVptr CPXopenCPLEX (int  *status_p);
int CPXcloseCPLEX (CPXENVptr *env_p);
CPXLPptr CPXcreateprob (CPXENVptr env, int *status_p, char *probname);
int CPXreadcopyprob (CPXENVptr env, CPXLPptr lpx, char *filename_str, char *filetype_str);
int CPXwriteprob (CPXENVptr env, CPXLPptr lpx, char *filename, char *filetype);
CPXLPptr CPXcloneprob (CPXENVptr env, CPXLPptr lpx, int *status_p);
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 CPXprimopt ( 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 CPXgetphase1cnt ( CPXENVptr env, CPXLPptr lpx);
int CPXgetstat (CPXENVptr env, CPXLPptr lpx);
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 CPXmipopt( CPXENVptr env, CPXLPptr lpx);
int CPXgetmipx (CPXENVptr env, CPXLPptr lpx, double *x, int begin, int end );
int CPXgetmipobjval (CPXENVptr env, CPXLPptr lpx, double *objval_p);
int CPXgetbestobjval (CPXENVptr env, CPXLPptr lpx, double *objval_p);
int CPXchgprobtype ( CPXENVptr env, CPXLPptr lpx, int type);
int CPXflushstdchannels (CPXENVptr env);
int CPXgetcolname (CPXENVptr env, CPXLPptr lpx, char **name, char *namestore,
        int storespace, int *surplus_p, int begin, int end);

int CPXdelsetrows (CPXENVptr env, CPXLPptr lpx, int *delstat);
int CPXchgbds(CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, char *lu, double *bd);
int CPXgetrows (CPXENVptr env, CPXLPptr lp, int *nzcnt, int *rmatbeg, int *rmatind,
        double *rmatval, int rmatspace, int *surplus, int begin, int end);

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 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 CPXgetcallbackinfo ( CPXENVptr env, void *cbdata, int wherefrom, int whichinfo, void *result_p);
int CPXsetintparam (CPXENVptr env, int whichparam, int newvalue);
int CPXsetdblparam (CPXENVptr env, int whichparam, double newvalue);
int CPXcloseCPLEX (CPXENVptr *env_p);
int CPXdualopt( CPXENVptr env, CPXLPptr lp);
int CPXlpopt ( CPXENVptr env, CPXLPptr lplocale);
```

A.2 ifc_imp.c

```
#include "LocBra.h"

CPXENVptr CPXopenCPLEX (int *status_p)
{
    int valore = 1;
    int *falso;
    falso = &valore;
    return falso;
}

CPXLPptr CPXcreateprob (CPXENVptr env, int *status_p, char *probname)
{
    QSprob *temp;
    temp = QScreate_prob(probname, 1);
    return temp;
}

int CPXgetprobname (CPXENVptr env, CPXLPptr lp, char *buf_str, int bufspace, int *surplus_p)
{
    char *nome = QSget_probname(lp);
    char *tmp = buf_str;
    while (*nome != '\0')
    {
        *tmp = *nome;
        tmp++;
        nome++;
    }
    *tmp = '\0';
    return 0;
}

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

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

int CPXreadcopyprob (CPXENVptr env, CPXLPptr prob,  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 = QSread_prob(filename_str,"LP");
            return 0;
        }

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

    else
    {
        int l = strlen(ext);
        if (l == 2)
        {
            if( Controllo(ext,"lp","LP") )
            {
                lp = QSread_prob(filename_str,"LP");
                return 0;
            }
            else
            {
                return 1;
            }
        }
        if (l == 3)
        {
            if( Controllo(ext,"mps","MPS") )
            {
                lp = QSread_prob(filename_str, "MPS");
                return 0;
            }
            else
            {
                return 1;
            }
        }
    }
    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;
}

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;

    if (tipofile == NULL)
    {
        tmp = (tmp + size) - 3;
        s = Controllo(tmp,".lp",".LP");
        if (s == 1)
        {
            copia = QSread_prob(nomefile, "LP");
            return copia;
        }
        if (s == 0)
        {
            tmp = (nomefile + size) - 4;
            s = Controllo(tmp,".mps",".MPS");
            if (s == 1)
            {
                copia = QSread_prob(nomefile, "MPS");
                return copia;
            }
        }
        return NULL;
    }
    else
    {
        int l = strlen(ext);
        if (l == 2)
        {
            if( Controllo(ext,"lp","LP") )
            {
                copia = QSread_prob(nomefile, "LP");
                return copia;
            }
            else
            {
                return NULL;
            }
        }
        if (l == 3)
        {
            if( Controllo(ext,"mps","MPS") )
            {
                copia = QSread_prob(nomefile, "MPS");
                return NULL;
            }
            else
            {
                return NULL;
            }
        }
    }
    return NULL;
}

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

int CPXgetobjsen (CPXENVptr env, CPXLPptr lp)
{
    return 1;
}

int CPXgetctype( CPXENVptr env, CPXLPptr lpx, char *xctype, int begin, int end)
{
    int i;
    int *intflags;
    int ncols;
    ncols = QSget_colcount (lpx);
    intflags = (int *) malloc (ncols * sizeof (int));
    int rval = QSget_intflags (lpx, intflags);

    for( i = begin ; i < (end + 1); i++ )
    {
        int t = intflags[i];
        if ( t != 1 )
        {
            *xctype = 'C';
            xctype++;
        }
        else
        {
            if ( t == 1 )
            {
                int stat;
                double ub,lb;
                stat = QSget_bound (lpx, i, 'U', &ub);
                stat = QSget_bound (lpx, i, 'L', &lb);
                if (( lb == 0) && (ub == 1) )
                {
                    *xctype = 'B';
                    xctype++;
                }
                else
                {
                    *xctype = 'I';
                    xctype++;
                }
            }
        }
    }
    return 0;
}

int CPXgetobj (CPXENVptr env, CPXLPptr lpx, double *obj, int begin, int end)
{
    int rval, ncols;
    double *o;
    ncols = QSget_colcount (lpx);
    o = (double *) malloc (ncols * sizeof (double));

    rval = QSget_obj (lpx, o);

    int i;
    for( i = begin ; i < (end+ 1) ; i++ )
    {
        obj[i-begin] = o[i];
    }
    return 0;
}

int CPXgetub (CPXENVptr env, CPXLPptr lpx, double *ub, int begin, int end)
{
    int i;
    for( i = begin ; i < (end + 1) ; i++ )
    {
        int rval;
        double upper,lower;
        rval = QSget_bound (lpx, i, 'U', &upper);
        if (upper < QS_MAXDOUBLE)
        {
            if (upper > CPX_INFBOUND ) *ub = CPX_INFBOUND;
            else *ub = upper;
            ub++;
        }
        else
        {
            *ub = CPX_INFBOUND;
            ub++;
        }
    }
    return 0;
}

int CPXgetsense (CPXENVptr env, CPXLPptr lpx, char *sense, int begin, int end)
{
    int type;
    double *rowval = NULL, *rhs = NULL;
    int *rowcnt = NULL, *rowbeg = NULL, *rowind = NULL;
    char *sen = NULL, **names = NULL;
    int nrows, i, j, rval;

    nrows = QSget_rowcount (lpx);
    rval = QSget_rows (lpx, &rowcnt, &rowbeg, &rowind, &rowval, &rhs, &sen, &names);

    for (i = begin ; i< (end + 1); i++)
    {
        type = sen[i];
        if (type == 'E')
        {
            *sense = 'E';
            sense++;
        }
        if (type == 'G')
        {
            *sense = 'G';
            sense++;
        }
        if (type == 'L')
        {
            *sense = 'L';
            sense++;
        }
    }
    return 0;
}

int CPXgetrhs ( CPXENVptr env, CPXLPptr lpx, double *rhs, int begin, int end)
{
    int rval, nrows;
    double *r;

    nrows = QSget_rowcount (lpx);
    r = (double *) malloc (nrows * sizeof (double));
    rval = QSget_rhs (lpx, r);

    int i;
    for (i = begin ; i < (end + 1) ; i++)
    {
        *rhs = r[i];
        rhs++;
    }
    return 0;
}
```

```
int CPXprimopt ( CPXENVptr env, CPXLPptr lpx)
{
    int r;
    int status;
    r = QSopt_primal(lpx,&status);
    return 0;
}

int CPXgetobjval ( CPXENVptr env, CPXLPptr lpx, double *objval_p)
{
    double soluzione;
    int r = QSget_objval(lpx, &soluzione);
    *objval_p = soluzione;
    return 0;
}

int CPXgetx ( CPXENVptr env, CPXLPptr lpx, double *x, int begin, int end)
{
    int i;
    int rval, ncols;
    double *xx;
    ncols = QSget_colcount (lpx);
    xx = (double *) malloc (ncols * sizeof (double));

    rval = QSget_x_array (lpx, xx);

    for (i = begin; i < (end+1) ; i++)
    {
        x[i-begin] = xx[i];
    }
    return 0;
}

int CPXgetdj ( CPXENVptr env, CPXLPptr lpx, double *dj, int begin, int end)
{
    int rval, ncols;
    double *rc;

    ncols = QSget_colcount (lpx);
    rc = (double *) malloc (ncols * sizeof (double));

    rval = QSget_rc_array (lpx, rc);

    int i;
    for (i = begin ; i < (end + 1) ; i++)
    {
        *dj = rc[i];
        dj++;
    }
    return 0;
}

int CPXgetpi ( CPXENVptr env, CPXLPptr lpx, double *pi, int begin, int end)
{
    int rval, nrows;
    double *pix;
    nrows = QSget_rowcount (lpx);
    pix = (double *) malloc (nrows * sizeof (double));

    rval = QSget_pi_array (lpx, pix);

    int i;
    for (i = begin ; i < (end + 1) ; i++)
    {
        *pi = pix[i];
        pi++;
    }
    return 0;
}

int CPXgetphase1cnt ( CPXENVptr env, CPXLPptr lpx)
{
    int iter;
    int rval;
    rval = QSget_param (lpx, QS_PRICE_PDANTZIG, &iter);  //???
    return iter;
}

int CPXgetstat (CPXENVptr env, CPXLPptr lpx) {
    int res;
    int rval, status;
    rval = QSget_status (lpx, &status);

    switch(status)
    {
        case QS_LP_OPTIMAL:
        res = CPX_STAT_OPTIMAL;
        break;
        case QS_LP_INFEASIBLE:
        res = CPX_STAT_INFEASIBLE;
        break;
        case QS_LP_UNSOLVED:
        res = CPX_STAT_INFEASIBLE;
        break;
        case QS_LP_UNBOUNDED:
        res = CPX_STAT_INForUNBD;
        break;
        default:
        res = QS_LP_ABORTED;
        break;
    }
    return res;
}

int CPXchgsense (CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, char *sense)
{
    int i;
    int stat;
    for (i = 0 ; i < cnt ; i++ )
    {
        char tipo  = sense[i];
        switch(tipo)
        {
            case ’L’:
                stat = QSchange_sense (lpx, indices[i], ’L’);
                break;
            case ’G’:
                stat = QSchange_sense (lpx, indices[i],’G’);
                break;
            case ’E’:
                stat = QSchange_sense (lpx, indices[i],’E’);
                break;
            default:
            break;
        }
    }
    return 0;
}

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

```
{
    int t;
    int stat;
    for( t = 0 ; t < cnt ; t++ )
    {
        stat = QSchange_objcoef (lpx, indices[t], values[t]);
    }
    return 0;
}

int CPXchgrhs (CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, double *values)
{
    int i, rval;
    for ( i = 0 ; i < cnt ; i++ )
        rval = QSchange_rhscoef (lpx, indices[i], values[i] );

    return 0;
}

int CPXmipopt( CPXENVptr env, CPXLPptr lpx)
{
    int rval, i, ncols, ncand, branch = -1;
    int *candidatelist = (int *) NULL;
    double *xlist, *down_vals, *up_vals;
    double bestval;
    ncols = QSget_colcount (lpx);

    double *x;
    x = (double *) malloc (ncols * sizeof (double));
    rval = QSget_x_array (lpx, x);

    candidatelist = (int *) malloc (ncols * (sizeof(int)));
    xlist = (double *) malloc (ncols * (sizeof(double)));

    ncand = 0;
    for (i = 0; i < ncols; i++) {
        double t = x[i] - floor (x[i]);      /* t is the fractional part of x[i]  */
        if (t >= 0.1 && t <= 0.9) {   /* x[i] is at least 0.1 from integer */
            candidatelist[ncand] = i;
            xlist[ncand++] = x[i];
        }
    }

    if (ncand == 0) {
        free (candidatelist);
        free (xlist);
        return -1;
    }

    down_vals = (double *) malloc (ncand * sizeof(double));
    up_vals = (double *) malloc (ncand * sizeof(double));

    double soluzione;
    int r = QSget_objval(lpx, &soluzione);

    rval = QSopt_strongbranch (lpx, ncand, candidatelist, xlist, down_vals, up_vals, 50, soluzione);

    free (candidatelist);
    free (xlist);
    free (down_vals);
    free (up_vals);

    return 0;
}
```

```
int CPXgetmipx (CPXENVptr env, CPXLPptr lpx, double *x, int begin, int end )
{
    int i;
    int rval, ncols;
    double *xx;
    ncols = QSget_colcount (lpx);
    xx = (double *) malloc (ncols * sizeof (double));

    rval = QSget_x_array (lpx, xx);

    for (i = begin; i < (end+1) ; i++)
    {
        x[i-begin] = xx[i];
    }
    return 0;
}

int CPXgetmipobjval (CPXENVptr env, CPXLPptr lpx, double *objval_p)
{
    double soluz;
    int r = QSget_objval(lpx, &soluz);
    *objval_p = soluz;
    return 0;
}

int CPXgetbestobjval (CPXENVptr env, CPXLPptr lpx, double *objval_p)
{
    double soluz;
    int r = QSget_objval(lpx, &soluz);
    *objval_p = soluz;
    return 0;
}

int CPXchgprobtype ( CPXENVptr env, CPXLPptr lpx, int type)
{
    return 0;
}

int CPXflushstdchannels (CPXENVptr env)
{
    return 0;
}

int CPXgetcolname ( CPXENVptr env, CPXLPptr lpx, char **name, char
            *namestore, int storespace, int *surplus_p, int begin, int end)
{
    int i;
    int tmp = storespace;
    int rval, ncols, j;
    char **colnames;

    ncols = QSget_colcount (lpx);
    colnames = (char **) malloc (ncols * sizeof (char *));

    rval = QSget_colnames (lpx, colnames);

    for ( i = begin ; i < (end+1) ; i++)
    {
        char *nametmp = colnames[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 CPXdelsetrows (CPXENVptr env, CPXLPptr lpx, int *delstat)
{
    int nrows = QSget_rowcount(lpx);
    int delrows[nrows];
    int i,stat;
    int nrs = 0;
    for( i = 0 ; i < nrows ; i++ )
    {
        if (delstat[i] == 1)
        {

            delrows[nrs] = i;
            nrs++;
        }
    }
    if (nrs > 0)
    {
        stat = QSdelete_rows(lp, nrs, delrows);
    }
    return 0;
}

int CPXchgbds(CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, char *lu, double *bd)
{
    int i;
    int indice, rval;
    for ( i = 0 ; i < cnt ; i++ )
    {
        indice = indices[i];
        switch(lu[i])
        {
            case 'U':
                rval = QSchange_bound(lpx,indice,'U',bd[i]);
                break;
            case 'L':
                rval = QSchange_bound(lpx,indice,'L',bd[i]);
                break;
            case 'B':
                rval = QSchange_bound(lpx,indice,'L',bd[i]);
                rval = QSchange_bound(lpx,indice,'U',bd[i]);
                break;
            default:
            break;
        }
    }
    return 0;
}

int CPXgetrows (CPXENVptr env, CPXLPptr lpx, int *nzcnt, int
                *rmatbeg, int *rmatind, double *rmatval, int rmatspace, int
                    *surplus, int begin, int end)
{
    double *rowval = NULL, *rhs = NULL;
    int *rowcnt = NULL, *rowbeg = NULL, *rowind = NULL;
    char *sense = NULL, **names = NULL;
    int nrows, rval;

    nrows = QSget_rowcount (lpx);
    rval = QSget_rows (lpx, &rowcnt, &rowbeg, &rowind, &rowval, &rhs, &sense, &names);

    rmatbeg = rowbeg;
    rmatind = rowind;
    rmatval = rowval;
    nzcnt = rowcnt;

    return 0;
}

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 rval,k;
    const char **names = colname;
    int *cmatcnt;
    cmatcnt = (int *) malloc (nzcnt * sizeof (int));
    k = 0;
    for (k = 0; k < ccnt; k++)
    {
        if (k < ccnt-1) cmatcnt[k] = (int) (cmatbeg[k+1] - cmatbeg[k]);
        else cmatcnt[k] = (int) (ccnt - cmatbeg[k]);
    }

    rval = QSadd_cols (lpx, ccnt, cmatcnt, cmatbeg, cmatind, cmatval, obj, lb, ub, names);
    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 r;
        for ( t = 0 ; t < ccnt ; t++ )
        {
            if (colname != NULL) r = QSnew_col(lpx,0,0,0,colname[t]);
            else r = QSnew_col(lpx,0,0,0,"");
        }
    }

    int rval,k;
    const char **names = colname;
    int *rmatcnt;
    rmatcnt = (int *) malloc (nzcnt * sizeof (int));
    k = 0;
    for (k = 0; k < rcnt; k++)
    {
        if (k < rcnt-1) rmatcnt[k] = (int) (rmatbeg[k+1] - rmatbeg[k]);
        else rmatcnt[k] = (int) (rcnt - rmatbeg[k]);
    }

    rval = QSadd_rows(lpx, rcnt, rmatcnt, rmatbeg, rmatind, rmatval, rhs, sense, names);

    return 0;
}
```

```
int CPXgetcallbackinfo ( CPXENVptr env, void *cbdata, int wherefrom, int whichinfo, void *result_p)
{
    cbdata = lp;
    double r;
    int rv = QSget_objval(lp,&r);
    result_p = &r;
    return 0;
}

int CPXsetintparam (CPXENVptr env, int whichparam, int newvalue)
{
    switch(whichparam)
    {
        case CPX_PARAM_MIPINTERVAL:
            //????
        break;
        case CPX_PARAM_PREIND:
            //?????????
        case CPX_PARAM_SCRIND:
        //??????????????
        break;
        case CPX_PARAM_ITLIM:
            QSset_param(lp, QS_PARAM_SIMPLEX_MAX_ITERATIONS, newvalue);
            QSset_param(lp2, QS_PARAM_SIMPLEX_MAX_ITERATIONS, newvalue);
            QSset_param(lp3, QS_PARAM_SIMPLEX_MAX_ITERATIONS, newvalue);
            QSset_param(lpf, QS_PARAM_SIMPLEX_MAX_ITERATIONS, newvalue);
        break;
        case CPX_PARAM_INTSOLLIM:
        //?????
        break;
        case CPX_PARAM_MIPSTART:
        //?????
        break;
        default:
        break;
    }
    return 0;
}

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:
            //??????????
        break;
        default:
        break;
    }
    return 0;
}

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

int CPXlpopt ( CPXENVptr env, CPXLPptr lpx)
{
    int r;
    int status;

    QSbas B;

    B = QSget_basis (lpx);
    if (B !=  NULL)
    {
            r = QSopt_primal(lpx,&status);
    }
    else
    r = QSopt_dual(lpx,&status);

    return 0;
}

int CPXdualopt ( CPXENVptr env, CPXLPptr lpx)
{
    int r;
    int status;
    r = QSopt_dual(lpx,&status);
    return 0;
}
```

Appendix B

Source code: "from ILOG-CPLEX to Coin-Or"

B.1 interface.h

```
#include "c:/cygwin/COIN/include/OsiClpSolverInterface.hpp"
#include "c:/cygwin/COIN/include/CoinPackedVector.hpp"
#include "c:/cygwin/COIN/include/CoinDistance.hpp"
#include "c:/cygwin/COIN/include/CoinPackedMatrix.hpp"

typedef int * CPXENVptr;

typedef OsiSolverInterface * CPXLPptr;

typedef struct ogg {
 double s;
 int    i;
}ogg;

typedef struct move {
 int BIG;
 int SMALL;
 int hash;
 double val;
 double lb;
}move;

#define CPXoptimize     CPXlpopt

#define CPX_ALG_NONE                    -1
#define CPX_ALG_AUTOMATIC                0
#define CPX_ALG_PRIMAL                   1
#define CPX_ALG_DUAL                     2
#define CPX_ALG_NET                      3
#define CPX_ALG_BARRIER                  4
#define CPX_ALG_SIFTING                  5
#define CPX_ALG_CONCURRENT               6
#define CPX_ALG_BAROPT                   7
#define CPX_ALG_PIVOTIN                  8
#define CPX_ALG_PIVOTOUT                 9
#define CPX_ALG_PIVOT                   10
#define CPX_ALG_ANY                      BIGINT
```

```
#define CPX_MAX                        -1
#define CPX_MIN                         1

#define CPXPROB_LP  0
#define CPXPROB_MILP  1
#define CPXPROB_FIXEDMILP  3
#define CPXPROB_QP  5
#define CPXPROB_MIQP 7
#define CPXPROB_FIXEDMIQP 8

#define CPX_INFBOUND  1.0E+20

#define CPXERR_NEGATIVE_SURPLUS 1207

#define CPX_AT_LOWER                    0
#define CPX_BASIC                       1
#define CPX_AT_UPPER                    2
#define CPX_FREE_SUPER                  3

//per wherefrom
#define CPX_CALLBACK_PRIMAL              1
#define CPX_CALLBACK_DUAL                2
#define CPX_CALLBACK_NETWORK             3
#define CPX_CALLBACK_PRIMAL_CROSSOVER    4
#define CPX_CALLBACK_DUAL_CROSSOVER      5
#define CPX_CALLBACK_BARRIER             6
#define CPX_CALLBACK_PRESOLVE            7
#define CPX_CALLBACK_QPBARRIER           8
#define CPX_CALLBACK_QPSIMPLEX           9

#define CPX_CALLBACK_MIP               101
#define CPX_CALLBACK_MIP_BRANCH        102
#define CPX_CALLBACK_MIP_NODE          103
#define CPX_CALLBACK_MIP_HEURISTIC     104
#define CPX_CALLBACK_MIP_SOLVE         105
#define CPX_CALLBACK_MIP_CUT           106
#define CPX_CALLBACK_MIP_PROBE         107
#define CPX_CALLBACK_MIP_FRACCUT       108
#define CPX_CALLBACK_MIP_DISJCUT       109
#define CPX_CALLBACK_MIP_FLOWMIR       110
#define CPX_CALLBACK_MIP_INCUMBENT     111
#define CPX_CALLBACK_MIP_DELETENODE    112

#define CPX_CALLBACK_INFO_BEST_INTEGER        101

/* MIP Parameter numbers */
#define CPX_PARAM_BRDIR                2001
#define CPX_PARAM_BTTOL                2002
#define CPX_PARAM_CLIQUES              2003
#define CPX_PARAM_COEREDIND            2004
#define CPX_PARAM_COVERS               2005
#define CPX_PARAM_CUTLO                2006
#define CPX_PARAM_CUTUP                2007
#define CPX_PARAM_EPAGAP               2008
#define CPX_PARAM_EPGAP                2009
#define CPX_PARAM_EPINT                2010
#define CPX_PARAM_HEURISTIC            2011
#define CPX_PARAM_MIPDISPLAY           2012
#define CPX_PARAM_MIPINTERVAL          2013
#define CPX_PARAM_MIPTHREADS           2014
#define CPX_PARAM_INTSOLLIM            2015
#define CPX_PARAM_NODEFILEIND          2016
#define CPX_PARAM_NODELIM              2017
#define CPX_PARAM_NODESEL              2018
#define CPX_PARAM_OBJDIF               2019
```

```
#define CPX_PARAM_MIPORDIND            2020
#define CPX_PARAM_RELOBJDIF            2022
#define CPX_PARAM_STARTALG             2025
#define CPX_PARAM_SUBALG               2026
#define CPX_PARAM_TRELIM               2027
#define CPX_PARAM_VARSEL               2028
#define CPX_PARAM_BNDSTRENIND          2029
#define CPX_PARAM_HEURFREQ             2031
#define CPX_PARAM_MIPORDTYPE           2032
#define CPX_PARAM_CUTSFACTOR           2033
#define CPX_PARAM_RELAXPREIND          2034
#define CPX_PARAM_MIPSTART             2035
#define CPX_PARAM_PRESLVND             2037
#define CPX_PARAM_BBINTERVAL           2039
#define CPX_PARAM_FLOWCOVERS           2040
#define CPX_PARAM_IMPLBD               2041
#define CPX_PARAM_PROBE                2042
#define CPX_PARAM_GUBCOVERS            2044
#define CPX_PARAM_STRONGCANDLIM        2045
#define CPX_PARAM_STRONGITLIM          2046
#define CPX_PARAM_STRONGTHREADLIM      2047
#define CPX_PARAM_FRACCAND             2048
#define CPX_PARAM_FRACCUTS             2049
#define CPX_PARAM_FRACPASS             2050
#define CPX_PARAM_FLOWPATHS            2051
#define CPX_PARAM_MIRCUTS              2052
#define CPX_PARAM_DISJCUTS             2053
#define CPX_PARAM_AGGCUTLIM            2054
#define CPX_PARAM_MIPCBREDLP           2055
#define CPX_PARAM_CUTPASS              2056
#define CPX_PARAM_MIPEMPHASIS          2058
#define CPX_PARAM_SYMMETRY             2059
#define CPX_PARAM_DIVETYPE             2060

/* CPLEX Parameter numbers */

#define CPX_PARAM_ADVIND               1001
#define CPX_PARAM_AGGFILL              1002
#define CPX_PARAM_AGGIND               1003
#define CPX_PARAM_BASINTERVAL          1004
#define CPX_PARAM_CFILEMUL             1005
#define CPX_PARAM_CLOCKTYPE            1006
#define CPX_PARAM_CRAIND               1007
#define CPX_PARAM_DEPIND               1008
#define CPX_PARAM_DPRIIND              1009
#define CPX_PARAM_PRICELIM             1010
#define CPX_PARAM_EPMRK                1013
#define CPX_PARAM_EPOPT                1014
#define CPX_PARAM_EPPER                1015
#define CPX_PARAM_EPRHS                1016
#define CPX_PARAM_FASTMIP              1017
#define CPX_PARAM_IISIND               1018
#define CPX_PARAM_SIMDISPLAY           1019
#define CPX_PARAM_ITLIM                1020
#define CPX_PARAM_ROWREADLIM           1021
#define CPX_PARAM_NETFIND              1022
#define CPX_PARAM_COLREADLIM           1023
#define CPX_PARAM_NZREADLIM            1024
#define CPX_PARAM_OBJLLIM              1025
#define CPX_PARAM_OBJULIM              1026
#define CPX_PARAM_PERIND               1027
#define CPX_PARAM_PERLIM               1028
#define CPX_PARAM_PPRIIND              1029
#define CPX_PARAM_PREIND               1030
#define CPX_PARAM_REINV                1031
```

```
#define CPX_PARAM_REVERSEIND           1032
#define CPX_PARAM_RFILEMUL             1033
#define CPX_PARAM_SCAIND               1034
#define CPX_PARAM_SCRIND               1035
#define CPX_PARAM_SIMTHREADS           1036
#define CPX_PARAM_SINGLIM              1037
#define CPX_PARAM_SINGTOL              1038
#define CPX_PARAM_TILIM                1039
#define CPX_PARAM_XXXIND               1041
#define CPX_PARAM_PREDUAL              1044
#define CPX_PARAM_ROWGROWTH            1046
#define CPX_PARAM_COLGROWTH            1047
#define CPX_PARAM_NZGROWTH             1048
#define CPX_PARAM_EPOPT_H              1049
#define CPX_PARAM_EPRHS_H              1050
#define CPX_PARAM_PREPASS              1052
#define CPX_PARAM_DATACHECK            1056
#define CPX_PARAM_REDUCE               1057
#define CPX_PARAM_PRELINEAR            1058
#define CPX_PARAM_LPMETHOD             1062
#define CPX_PARAM_QPMETHOD             1063
#define CPX_PARAM_WORKDIR              1064
#define CPX_PARAM_WORKMEM              1065
#define CPX_PARAM_PRECOMPRESS          1066
#define CPX_PARAM_THREADS              1067
#define CPX_PARAM_SIFTDISPLAY          1076
#define CPX_PARAM_SIFTALG              1077
#define CPX_PARAM_SIFTITLIM            1078

/* Values returned for 'stat' by solution () */

#define CPX_STAT_OPTIMAL                  1
#define CPX_STAT_UNBOUNDED                2
#define CPX_STAT_INFEASIBLE               3
#define CPX_STAT_INForUNBD                4
#define CPX_STAT_OPTIMAL_INFEAS           5
#define CPX_STAT_NUM_BEST                 6
#define CPX_STAT_ABORT_IT_LIM            10
#define CPX_STAT_ABORT_TIME_LIM          11
#define CPX_STAT_ABORT_OBJ_LIM           12
#define CPX_STAT_ABORT_USER              13

/* Solution type return values from CPXsolninfo() */
#define CPX_NO_SOLN        0
#define CPX_BASIC_SOLN     1
#define CPX_NONBASIC_SOLN  2

/* Values of presolve 'stats' for columns and rows */
#define CPX_PRECOL_LOW               -1
#define CPX_PRECOL_UP                -2
#define CPX_PRECOL_FIX               -3
#define CPX_PRECOL_AGG               -4
#define CPX_PRECOL_OTHER             -5
#define CPX_PREROW_RED               -1
#define CPX_PREROW_AGG               -2
#define CPX_PREROW_OTHER             -3

/* MIP Problem status codes */

#define CPXMIP_OPTIMAL                 101
#define CPXMIP_OPTIMAL_TOL             102
#define CPXMIP_INFEASIBLE              103
#define CPXMIP_SOL_LIM                 104
#define CPXMIP_NODE_LIM_FEAS           105
#define CPXMIP_NODE_LIM_INFEAS         106
```

```
#define CPXMIP_TIME_LIM_FEAS           107
#define CPXMIP_TIME_LIM_INFEAS         108
#define CPXMIP_FAIL_FEAS               109
#define CPXMIP_FAIL_INFEAS             110
#define CPXMIP_MEM_LIM_FEAS            111
#define CPXMIP_MEM_LIM_INFEAS          112
#define CPXMIP_ABORT_FEAS              113
#define CPXMIP_ABORT_INFEAS            114
#define CPXMIP_OPTIMAL_INFEAS          115
#define CPXMIP_FAIL_FEAS_NO_TREE       116
#define CPXMIP_FAIL_INFEAS_NO_TREE     117
#define CPXMIP_UNBOUNDED               118
#define CPXMIP_INForUNBD               119

CPXENVptr CPXopenCPLEX (int  *status_p);
int CPXcloseCPLEX (CPXENVptr *env_p);
CPXLPptr CPXcreateprob (CPXENVptr env, int *status_p, char *probname);
int CPXreadcopyprob (CPXENVptr env, CPXLPptr lpx, char *filename_str, char *filetype_str);
CPXLPptr CPXcloneprob (CPXENVptr env, CPXLPptr lpx, int *status_p);
int CPXsetintparam (CPXENVptr env, int whichparam, int newvalue);
int CPXsetdblparam (CPXENVptr env, int whichparam, double newvalue);
int CPXwriteprob (CPXENVptr env, CPXLPptr lpx, char *filename, char *filetype);
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 CPXgetphase1cnt ( CPXENVptr env, CPXLPptr lpx);
int CPXgetstat (CPXENVptr env, CPXLPptr lpx);
int CPXgetx (CPXENVptr env, CPXLPptr lpx, double *x, int begin, int end);
int CPXgetobjval ( CPXENVptr env, CPXLPptr lpx, double *objval_p);
int CPXgetub (CPXENVptr env, CPXLPptr lpx, double *ub, int begin, int end);
int CPXgetdj (CPXENVptr env, CPXLPptr lpx, double *dj, int begin, int end);
int CPXgetrhs (CPXENVptr env, CPXLPptr lpx, double *rhs, int begin, int end);
int CPXgetpi (CPXENVptr env, CPXLPptr lpx, double *pi, int begin, int end);
int CPXgetcolname (CPXENVptr env, CPXLPptr lpx, char **name, char *namestore,
        int storespace, int *surplus_p, int begin, int end);

int CPXgetmipobjval (CPXENVptr env, CPXLPptr lpx, double *objval_p);
int CPXgetbestobjval (CPXENVptr env, CPXLPptr lpx, double *objval_p);
int CPXflushstdchannels (CPXENVptr env);
int CPXgetcallbackinfo ( CPXENVptr env, void *cbdata, int wherefrom, int whichinfo, void *result_p);
int CPXgetrows (CPXENVptr env, CPXLPptr lp, int *nzcnt, int *rmatbeg, int *rmatind, double *rmatval,
        int rmatspace, int *surplus, int begin, int end);

int CPXgetsense (CPXENVptr env, CPXLPptr lpx, char *sense, int begin, int end);
int CPXprimopt ( CPXENVptr env, CPXLPptr lpx);
int CPXdualopt( CPXENVptr env, CPXLPptr lpx);
int CPXlpopt ( CPXENVptr env, CPXLPptr lpx);
int CPXmipopt( CPXENVptr env, CPXLPptr lpx);
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 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 CPXgetmipx (CPXENVptr env, CPXLPptr lpx, double *x, int begin, int end );
int CPXchgprobtype ( CPXENVptr env, CPXLPptr lpx, int type);
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);
```

B.2 ifc_imp.cpp

```
#include "interface.h"
#include <string> using namespace std;

CPXENVptr  env;
CPXLPptr   lp;
CPXLPptr   lp2;
CPXLPptr   lp3;
CPXLPptr   lpf;

int        ncols, nrows;   /* ncols, nrows:     ORIGINAL size of the problem   */
int        nBcols;         /* number of binary variables                       */
int        nIcols;         /* number of integer variables                      */
int        nCcols;         /* number of continuous variables                   */
int        MinMax;         /* 1/-1 MINimization/MAXimization problem           */
int Inf;
int debug;
int dmax;
int dBest, dFirst;
int iBest, iFirst;
int dv;
int kSize;
int original;
int lastCardS;
int nNewCols;
int feas;
int nSlackBest;

int emphasis;
int presolve;
int heurFreq;
int prec;
int mipInt;
int nIntervals;
int video;

double     ts, tss, time4Best, timeSlot, time4First, TL, addTime;
double     Eps;
double     dInf;
double     overallBestValue;
double     slack;

int        maxNode, N1, IMP, EMERGENZA, R, TT, itLim, WRITE;
int        nIter, firstTabu, nodes, SAT, nH, STALLED, changed, maxIter, addIter;
double     delta, pdgap;
double     bestBound, firstValue;
double     tt;
double     soglia, Hdistance;
double     divv;
move       *old, *cur, *new1;
char       *ctype;
int        *RINS;

int        *slot;
int        *tabu;


CPXENVptr CPXopenCPLEX (int *status_p)
{
    *status_p = 1;
    int r = 1;
    return &r;
}
```

```
CPXLPptr CPXcreateprob (CPXENVptr env, int *status_p, char *probname)
{
    OsiSolverInterface *temp = new OsiClpSolverInterface;
    return temp;
}

int CPXreadcopyprob (CPXENVptr env, CPXLPptr lpx, char *filename_str, char *filetype_str)
{
    (*lpx).readMps(filename_str);
    return 0;
}

CPXLPptr CPXcloneprob (CPXENVptr env, CPXLPptr lpx, int *status_p)
{
    OsiSolverInterface *temp = (*lpx).clone();
    return temp;
}

int CPXgetnumrows (CPXENVptr env, CPXLPptr lpx)
{
    return (*lpx).getNumRows();
}

int CPXgetnumcols (CPXENVptr env, CPXLPptr lpx)
{
    return (*lpx).getNumCols();
}

int CPXgetobjsen (CPXENVptr env, CPXLPptr lpx)
{
    return (int)(*lpx).getObjSense();
}

int CPXgetctype (CPXENVptr env, CPXLPptr lpx, char *xctype, int begin, int end)
{
    int i;
    for( i = begin ; i < (end + 1); i++ )
    {
        if ( (*lpx).isContinuous(i ))
        {
            *xctype = 'C';
            xctype++;
        }
        else
        {
            if (( ((*lpx).getColLower())[i] == 0) && (((*lpx).getColUpper())[i] == 1 ) )
            {
                *xctype = 'B';
                xctype++;
            }
            else
            {
                *xctype = 'I';
                xctype++;
            }
        }
    }
    return 0;
}

int CPXgetprobname (CPXENVptr env, CPXLPptr lpx, char *buf_str, int bufspace, int *surplus_p)
{
    return 0;
}
```

```
int CPXgetobj (CPXENVptr env, CPXLPptr lpx, double *obj, int begin, int end)
{
    for(int i = begin ; i < (end+ 1) ; i++ )
    {
        obj[i-begin] = ((*lpx).getObjCoefficients())[i];
    }
    return 0;
}

int CPXprimopt ( CPXENVptr env, CPXLPptr lpx)
{
    if (presolve == 1) (*lpx).setHintParam(OsiDoPresolveInResolve, 1);
    else (*lpx).setHintParam(OsiDoPresolveInResolve, 0);

    (*lpx).setHintParam(OsiDoDualInResolve, 0);
    (*lpx).resolve();
    return 0;
}

int CPXgetphase1cnt ( CPXENVptr env, CPXLPptr lpx)
{
    return (*lpx).getIterationCount();
}

int CPXgetstat (CPXENVptr env, CPXLPptr lpx)
{
    if ( (*lpx).isProvenOptimal() ) return CPX_STAT_OPTIMAL;
    if ( (*lpx).isIterationLimitReached() ) return CPX_STAT_ABORT_IT_LIM;
    if ( (*lpx).isProvenPrimalInfeasible() || (*lpx).isProvenDualInfeasible() )
        return CPX_STAT_INForUNBD ;
    if ( (*lpx).isPrimalObjectiveLimitReached() || (*lpx).isDualObjectiveLimitReached() )
        return CPX_STAT_ABORT_OBJ_LIM;
    if ( (*lpx).isAbandoned() ) return CPX_STAT_UNBOUNDED;

    return -1;
}

int CPXgetx (CPXENVptr env, CPXLPptr lpx, double *x, int begin, int end)
{
    for ( int i = begin ; i < (end+1) ; i++ )
    {
        x[i-begin] =   ((*lpx).getColSolution())[i];
    }
    return 0;
}

int CPXchgobj (CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, double *values)
{
    for(int t = 0 ; t < cnt ; t++ )
    {
        (*lpx).setObjCoeff( indices[t], values[t] );
    }
    return 0;
}

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

int CPXgetub (CPXENVptr env, CPXLPptr lpx, double *ub, int begin, int end)
{
    for(int i = begin ; i < (end + 1) ; i++ )
    {
        double bound = ((*lpx).getColUpper())[i];
        if( bound != (*lpx).getInfinity() )
        {
            if ( bound > CPX_INFBOUND ) *ub = CPX_INFBOUND;
            else *ub = bound;
            ub++;
        }
        else
        {
            *ub = CPX_INFBOUND;
            ub++;
        }
    }
    return 0;
}

int CPXchgprobtype ( CPXENVptr env, CPXLPptr lpx, int type)
{
    return 0;
}

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

int CPXgetrhs (CPXENVptr env, CPXLPptr lpx, double *rhs, int begin, int end)
{
    for (int i = begin ; i < (end + 1) ; i++)
    {
            rhs[i-begin] = ((*lpx).getRightHandSide())[i];
    }
    return 0;
}

int CPXgetdj ( CPXENVptr env, CPXLPptr lpx, double *dj, int begin, int end)
{
    int i;
    for (i = begin ; i < (end + 1) ; i++)
    {
        *dj = ((*lpx).getReducedCost())[i];
        dj++;
    }
    return 0;
}

int CPXgetpi ( CPXENVptr env, CPXLPptr lpx, double *pi, int begin, int end)
{
    int i;
    for (i = begin ; i < (end + 1) ; i++)
    {
        *pi = ((*lpx).getRowPrice())[i];
        pi++;
    }
    return 0;
}

int CPXchgsense (CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, char *sense)
{
    for ( int i = 0 ; i < cnt ; i++ )
    {
        int y = indices[i];
        double oldrhs = ((*lpx).getRightHandSide())[y];
        if ( ((*lpx).getRowSense())[i] == 'R' )
        {
            double oldrange = ((*lpx).getRowRange())[y];
            (*lpx).setRowType( y , sense[i] , oldrhs , oldrange );
        }
        else
        {
            (*lpx).setRowType( y , sense[i] , oldrhs , -1);
        }

    }
    return 0;
}

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

int CPXchgrhs (CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, double *values)
{
    int i,k;
    for ( i = 0 ; i < cnt ; i++ )
    {
        k = indices[i];
        (*lpx).setRowType(k, ((*lpx).getRowSense())[k], *values, ((*lpx).getRowRange())[k] );
    }
    return 0;
}

int CPXgetsense (CPXENVptr env, CPXLPptr lpx, char *sense, int begin, int end)
{
    int i;
    char s;
    for (i = begin; i< (end+1); i++)
    {
        s = ((*lpx).getRowSense())[i];
        if ( s != 'N')
        {
            *sense = s;
            sense++;
        }
    }
    return 0;
}

int CPXdelsetrows (CPXENVptr env, CPXLPptr lpx, int *delstat)
{
    int nrows = (*lpx).getNumRows();
    int delrows[nrows];
    int t = 0;
    for ( int i = 0; i < nrows; i++)
    {
        if (delstat[i] == 1)
        {
            delrows[t] = i;
            t++;
        }
    }
    (*lpx).deleteRows(t, delrows);
    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)
    {
        int nrows = (*lpx).getNumRows();
        double zeric[nrows];
        CoinPackedVector col = new CoinPackedVector(nrows, zeric);
        for ( int t = 0 ; t < ccnt ; t++ )
        {
            (*lpx).addCol(col, 0, (*lpx).getInfinity(), 0);
        }
    }
    int inizio;
    int fine;
    int len;

    int ncols = (*lpx).getNumCols();
    for ( int t = 0 ; t < rcnt ; t++)
    {
        inizio = rmatbeg[t];
        if (t == (rcnt-1)) fine = (nzcnt-1);
        else fine = (rmatbeg[t+1]-1);
        len = fine - inizio + 1;
        int index[len];
        double val[len];
        int c = 0;
        for( int q = 0 ; q < len ; q++ )
        {
            if(rmatval[inizio + q] != 0)
            {
                index[c] = rmatind[inizio + q];
                val[c] = rmatval[inizio + q];
                c++;
            }
        }
        CoinPackedVector row(c, index, val);
        if (sense != NULL)
            {
            char tipo = sense[t];
            switch(tipo)
            {
                case 'L':
                    if (rhs != NULL) (*lpx).addRow(row,  'L' , rhs[t], rhs[t]);
                    else (*lpx).addRow(row, 0, 0);
                    break;
                case 'G':
                    if (rhs != NULL) (*lpx).addRow(row, 'G', rhs[t],  rhs[t]);
                    else (*lpx).addRow(row, 0, 0);
                    break;
                case 'R':
                    if (rhs != NULL) (*lpx).addRow(row, 'R', rhs[t], rhs[t]);
                    else (*lpx).addRow(row, 0, 0);
                    break;
                case 'E':
                    if (rhs != NULL) (*lpx).addRow(row,'E', rhs[t], rhs[t]);
                    else (*lpx).addRow(row, 0, 0);
                    break;
                default:
                    (*lpx).addRow(row, 0, 0);
                    break;
            }
        }
    }
    return 0;
}
```

```
int CPXchgbds(CPXENVptr env, CPXLPptr lpx, int cnt, int *indices, char *lu, double *bd)
{
    int indice;
    for ( int i = 0 ; i < cnt ; i++ )
    {
        indice = indices[i];
        double oldlb = ((*lpx).getColLower())[indice];
        double oldub = oldub = ((*lpx).getColUpper())[indice];
        switch(lu[i])
        {
            case 'U':
                (*lpx).setColBounds(indice, oldlb, bd[i]);
                break;
            case 'L':
                (*lpx).setColBounds(indice, bd[i], oldub);
                break;
            case 'B':
                (*lpx).setColBounds(indice, bd[i],bd[i]);
                break;
            default:
                break;
        }
    }
    return 0;
}

int CPXmipopt( CPXENVptr env, CPXLPptr lpx)
{
    if (presolve == 1) (*lpx).setHintParam(OsiDoPresolveInResolve, 1);
    else (*lpx).setHintParam(OsiDoPresolveInResolve, 0);

    (*lpx).branchAndBound();
    return 0;
}

int CPXgetmipobjval (CPXENVptr env, CPXLPptr lpx, double *objval_p)
{
    *objval_p = (*lpx).getObjValue();
    return 0;
}

int CPXgetmipx (CPXENVptr env, CPXLPptr lpx, double *x, int begin, int end )
{
    int i;
    for ( i = begin ; i < (end+1) ; i++ )
    {
        x[i-begin] = ((*lpx).getColSolution())[i];
    }
    return 0;
}

int CPXgetbestobjval (CPXENVptr env, CPXLPptr lpx, double *objval_p)
{
    *objval_p = (*lpx).getObjValue();
    return 0;
}

int CPXflushstdchannels (CPXENVptr env)
{
    return 0;
}

int CPXwriteprob (CPXENVptr env, CPXLPptr lpx, char *filename, char *filetype)
{
    (*lpx).writeMps(filename,"mps",(*lpx).getObjSense());
    return 0;
}

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 inizio;
    int fine;
    int len;
    double inf;
    double sup;
    for ( int t = 0 ; t < ccnt ; t++ )
    {
        double objcoeff;
        if (obj == NULL) objcoeff = 0;
        else objcoeff = obj[t];

        inizio = cmatbeg[t];
        if (t == (ccnt-1)) fine = (nzcnt-1);
        else fine = (cmatbeg[t+1]-1);
        len = fine - inizio + 1;

        int nrows = (*lpx).getNumRows();
        int index[len];
        double val[len];
        for( int q = 0 ; q < len ; q++ )
        {
            index[q] = cmatind[inizio + q];
            val[q] = cmatval[inizio + q];
        }
        CoinPackedVector col(len, index, val);

        if (lb == NULL)
        {
            inf = 0;
        }
        else if (lb[t] <= -CPX_INFBOUND )
             {
                    inf = -(*lpx).getInfinity();
             }
             else
             {
                    inf = lb[t];
             }
        if ( (ub[t] >= CPX_INFBOUND) || (ub == NULL) )
        {
            sup = (*lpx).getInfinity();
        }
        else
        {
            sup = ub[t];
        }
        (*lpx).addCol(col, inf, sup, objcoeff);
    }
    return 0;
}

int CPXsetintparam (CPXENVptr env, int whichparam, int newvalue)
{
    switch (whichparam)
    {
        case CPX_PARAM_PREIND:
            (*lp).setHintParam(OsiDoPresolveInResolve, newvalue);
            (*lp2).setHintParam(OsiDoPresolveInResolve, newvalue);
            (*lp3).setHintParam(OsiDoPresolveInResolve, newvalue);
            (*lpf).setHintParam(OsiDoPresolveInResolve, newvalue);
            break;
        case  CPX_PARAM_ITLIM :
            (*lp).setIntParam(OsiMaxNumIteration, newvalue);
            (*lp2).setIntParam(OsiMaxNumIteration, newvalue);
            (*lp3).setIntParam(OsiMaxNumIteration, newvalue);
            (*lpf).setIntParam(OsiMaxNumIteration, newvalue);
            break;
        default:
        break;
    }
    return 0;
}

int CPXsetdblparam (CPXENVptr env, int whichparam, double newvalue)
{
    return 0;
}

int CPXgetcallbackinfo ( CPXENVptr env, void *cbdata, int wherefrom, int whichinfo, void *result_p)
{
    cbdata = lp;
    double r;
    r = (*lp).getObjValue();
    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)
{
    int nz = 0;
    int tmp = rmatspace;
    int len;
    const CoinPackedMatrix *matrice = (*lpx).getMatrixByRow();
    for( int i = begin ; i < (end + 1 ) ; i++ )
    {
        len = (*matrice).getVectorSize(i);
        if (len >0)
        {
            int start = ((*matrice).getVectorStarts())[i];
            rmatbeg[i-begin] = nz;
            for ( int t = 0 ; t < len ; t++ )
            {
                if ( tmp > 0 )
                {
                    rmatval[nz] = ((*matrice).getElements())[start+len];
                    rmatind[nz] = (int)((*matrice).getIndices())[start+len];
                    nz++;
                    tmp--;
                }
                *surplus--;
            }
        }
    }
    *nzcnt = nz;
    return 0;
}

int CPXlpopt ( CPXENVptr env, CPXLPptr lpx)
{
    if (presolve == 1) (*lpx).setHintParam(OsiDoPresolveInResolve, 1);
    else (*lpx).setHintParam(OsiDoPresolveInResolve, 0);

    if (!(*lpx).isProvenPrimalInfeasible())
    {
        (*lpx).setHintParam(OsiDoDualInResolve, 0);
        (*lpx).resolve();
    }
    else
    {
        (*lpx).setHintParam(OsiDoDualInResolve, 1);
        (*lpx).resolve();
    }

    return 0;
}

int CPXdualopt (  CPXENVptr env, CPXLPptr lpx)
{
    if (presolve == 1) (*lpx).setHintParam(OsiDoPresolveInResolve, 1);
    else (*lpx).setHintParam(OsiDoPresolveInResolve, 0);

    (*lpx).setHintParam(OsiDoDualInResolve, 1);
    (*lpx).resolve();
    return 0;
}
```

Bibliography

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

[2] 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.

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

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

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

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

[7] 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.

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

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

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

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

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

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

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

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

[16] 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.

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

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

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

[20] 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.

[21] 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.

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

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

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

[25] M. Fischetti, A. Lodi and F. Glover. The Feasibility Pump. May 8, 2004.

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

[27] Coin-or: Embedded documentation (Doxygen) in source code (http://sagan.ie.lehigh.edu/coin/)

[28] QSopt: QSopt 1.0, QSopt Reference Manual 1.0, October 27, 2003 (http://www.isye.gatech.edu/∼wcook/qsopt/index.html)

[29] Cygwin: http://www.cygwin.com/