How to speed-up hard problem resolution using GLPK ?

Onfroy B. & Cohen N.

September 27, 2010

Contents

1 Introduction

In this little document, I will try to explain how to interact with the *branch and cut* algorithm used by GLPK for LP resolution. In fact, a LP solver is able to solve P and NP problems, but in the case of NP-Complete problem resolution, it could be efficient to add cuts in order to speed-up the resolution. It is clear that the solver doesn't know your problem, it just knows its mathematical formulation.

But, in several cases we can provide heuristic algorithms for approximating the solution. On the other hand, we want an exact resolution providing the optimal solution. An heuristic cannot guarantee that it provides the optimal solution.

Thus, I will try to show how to find the optimal solution for a problem making exact and approximated resolution.

First of all, I will try to answer many questions about GLPK, about linear programming, about different solvers. In a second part, we will discover the famous *branch and cut* algorithm. Finally, we will explore the best way for interacting with the resolution algorithm processing in order to add cuts into the resolution (search) tree.

1.1 What is GLPK ?

The **GNU Linear Programming Kit**[1] (GLPK) is a software package intended for solving large-scale linear programming (LP), mixed integer programming (MIP), and other related problems. It is a set of routines written in ANSI C and organized in the form of a callable library. The package is part of the GNU Project and is released under the GNU General Public License. Problems can be modeled in the language GNU MathProg which shares many parts of the syntax with AMPL and solved with standalone solver GLPSOL. GLPK can also be used as a C library. GLPK uses the revised simplex method and the primal-dual interior point method for non-integer problems and the branch-and-bound algorithm together with Gomory's mixed integer cuts for (mixed) integer problems. An independent project has also been developed that provides a Java-based interface to GLPK (via JNI). This allows Java applications to call out to GLPK and is relatively transparent.

1.2 GLPK, the best one ?

In fact, GLPK is free but we can find several other solvers. In fact, GLPK is not the only one ! The website http://plato.asu.edu/bench.html presents benchmarks of many solvers and can help you to choose the good one for your problem.

1.3 What is a linear program ?

Linear programming[2] (LP) is a mathematical method for determining a way to achieve the best outcome (such as maximum profit or lowest cost) in a given mathematical model for some list of requirements represented as linear equations. More formally, linear programming is a technique for the optimization of a linear objective function, subject to linear equality and linear inequality constraints. Given a polytope and a real-valued affine function defined on this polytope, a linear programming method will find a point on the polytope where this function has the smallest (or largest) value if such point exists, by searching through the polytope vertices.

Ok, this topic is largely covered by a lot of books, so, if you are a newbie, ask Google for good books !

2 GLPK API

2.1 LP problem

GLPK assumes the following formulation of *linear programming (LP)* problem[3]:

[1] Source: Wikipedia
[2] Source: Wikipedia
[3] Source: GLPK documentation

minimize (or maximize)

$$z = c_1 x_{m+1} + c_2 x_{m+2} + \cdots + c_n x_{m+n} + c_0 \tag{1.1}$$

subject to linear constraints

$$\begin{aligned} x_1 &= a_{11} x_{m+1} + a_{12} x_{m+2} + \cdots + a_{1n} x_{m+n} \\ x_2 &= a_{21} x_{m+1} + a_{22} x_{m+2} + \cdots + a_{2n} x_{m+n} \\ &\cdots\cdots\cdots\cdots\cdots \\ x_m &= a_{m1} x_{m+1} + a_{m2} x_{m+2} + \cdots + a_{mn} x_{m+n} \end{aligned} \tag{1.2}$$

and bounds of variables

$$\begin{aligned} l_1 &\le x_1 \le u_1 \\ l_2 &\le x_2 \le u_2 \\ &\cdots\cdots \\ l_{m+n} &\le x_{m+n} \le u_{m+n} \end{aligned} \tag{1.3}$$

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

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

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

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

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

To solve the LP problem (1.1)—(1.3) is to find such values of all structural and auxiliary variables, which:

- satisfy to all the linear constraints (1.2), and
- are within their bounds (1.3), and
- provide the smallest (in case of minimization) or the largest (in case of maximization) value of the objective function (1.1).

2.2 MIP problem

Mixed integer linear programming (MIP) problem is LP problem in which some variables are additionally required to be integer.

GLPK assumes that MIP problem has the same formulation as ordinary (pure) LP problem (1.1)—(1.3), i.e. includes auxiliary and structural variables, which may have lower and/or upper bounds. However, in case of MIP problem some variables may be required to be integer. This additional constraint means that a value of each *integer variable* must be only integer number. (Should note that GLPK allows only structural variables to be of integer kind.)

2.3 Using the package

2.3.1 Brief example

In order to understand what GLPK is from the user's standpoint, consider the following simple LP problem:

maximize

$$z = 10x_1 + 6x_2 + 4x_3$$

subject to

$$\begin{aligned} x_1 + \ x_2 + \ x_3 &\leq 100 \\ 10x_1 + 4x_2 + 5x_3 &\leq 600 \\ 2x_1 + 2x_2 + 6x_3 &\leq 300 \end{aligned}$$

where all variables are non-negative

$$x_1 \geq 0, \ x_2 \geq 0, \ x_3 \geq 0$$

At first this LP problem should be transformed to the standard form (1.1)—(1.3). This can be easily done by introducing auxiliary variables, by one for each original inequality constraint. Thus, the problem can be reformulated as follows:

maximize

$$z = 10x_1 + 6x_2 + 4x_3$$

subject to

$$\begin{aligned} p &= \ \ x_1 + \ x_2 + \ x_3 \\ q &= 10x_1 + 4x_2 + 5x_3 \\ r &= \ 2x_1 + 2x_2 + 6x_3 \end{aligned}$$

and bounds of variables

$$\begin{aligned} -\infty < p \leq 100 \qquad & 0 \leq x_1 < +\infty \\ -\infty < q \leq 600 \qquad & 0 \leq x_2 < +\infty \\ -\infty < r \leq 300 \qquad & 0 \leq x_3 < +\infty \end{aligned}$$

where p, q, r are auxiliary variables (rows), and x_1, x_2, x_3 are structural variables (columns).

The example C program shown below uses GLPK API routines in order to solve this LP problem.[4]

```
/* sample.c */

#include <stdio.h>
#include <stdlib.h>
#include <glpk.h>
int main(void)
{     glp_prob *lp;
      int ia[1+1000], ja[1+1000];
      double ar[1+1000], z, x1, x2, x3;
s1:   lp = glp_create_prob();
s2:   glp_set_prob_name(lp, "sample");
s3:   glp_set_obj_dir(lp, GLP_MAX);
s4:   glp_add_rows(lp, 3);
s5:   glp_set_row_name(lp, 1, "p");
s6:   glp_set_row_bnds(lp, 1, GLP_UP, 0.0, 100.0);
s7:   glp_set_row_name(lp, 2, "q");
s8:   glp_set_row_bnds(lp, 2, GLP_UP, 0.0, 600.0);
s9:   glp_set_row_name(lp, 3, "r");
s10:  glp_set_row_bnds(lp, 3, GLP_UP, 0.0, 300.0);
s11:  glp_add_cols(lp, 3);
```

[4] If you just need to solve LP or MIP instance, you may write it in MPS or CPLEX LP format and then use the GLPK stand-alone solver to obtain a solution. This is much less time-consuming than programming in C with GLPK API routines.

```
s12:  glp_set_col_name(lp, 1, "x1");
s13:  glp_set_col_bnds(lp, 1, GLP_LO, 0.0, 0.0);
s14:  glp_set_obj_coef(lp, 1, 10.0);
s15:  glp_set_col_name(lp, 2, "x2");
s16:  glp_set_col_bnds(lp, 2, GLP_LO, 0.0, 0.0);
s17:  glp_set_obj_coef(lp, 2, 6.0);
s18:  glp_set_col_name(lp, 3, "x3");
s19:  glp_set_col_bnds(lp, 3, GLP_LO, 0.0, 0.0);
s20:  glp_set_obj_coef(lp, 3, 4.0);
s21:  ia[1] = 1, ja[1] = 1, ar[1] =  1.0; /* a[1,1] =  1 */
s22:  ia[2] = 1, ja[2] = 2, ar[2] =  1.0; /* a[1,2] =  1 */
s23:  ia[3] = 1, ja[3] = 3, ar[3] =  1.0; /* a[1,3] =  1 */
s24:  ia[4] = 2, ja[4] = 1, ar[4] = 10.0; /* a[2,1] = 10 */
s25:  ia[5] = 3, ja[5] = 1, ar[5] =  2.0; /* a[3,1] =  2 */
s26:  ia[6] = 2, ja[6] = 2, ar[6] =  4.0; /* a[2,2] =  4 */
s27:  ia[7] = 3, ja[7] = 2, ar[7] =  2.0; /* a[3,2] =  2 */
s28:  ia[8] = 2, ja[8] = 3, ar[8] =  5.0; /* a[2,3] =  5 */
s29:  ia[9] = 3, ja[9] = 3, ar[9] =  6.0; /* a[3,3] =  6 */
s30:  glp_load_matrix(lp, 9, ia, ja, ar);
s31:  glp_simplex(lp, NULL);
s32:  z = glp_get_obj_val(lp);
s33:  x1 = glp_get_col_prim(lp, 1);
s34:  x2 = glp_get_col_prim(lp, 2);
s35:  x3 = glp_get_col_prim(lp, 3);
s36:  printf("\nz = %g; x1 = %g; x2 = %g; x3 = %g\n",
         z, x1, x2, x3);
s37:  glp_delete_prob(lp);
      return 0;
}

/* eof */
```

The statement `s1` creates a problem object. Being created the object is initially empty. The statement `s2` assigns a symbolic name to the problem object.

The statement `s3` calls the routine `glp_set_obj_dir` in order to set the optimization direction flag, where `GLP_MAX` means maximization.

The statement `s4` adds three rows to the problem object.

The statement `s5` assigns the symbolic name '`p`' to the first row, and the statement `s6` sets the type and bounds of the first row, where `GLP_UP` means that the row has an upper bound. The statements `s7`, `s8`, `s9`, `s10` are used in the same way in order to assign the symbolic names '`q`' and '`r`' to the second and third rows and set their types and bounds.

The statement `s11` adds three columns to the problem object.

The statement `s12` assigns the symbolic name '`x1`' to the first column, the statement `s13` sets the type and bounds of the first column, where `GLP_LO` means that the column has an lower bound, and the statement `s14` sets the objective coefficient for the first column. The statements `s15`—`s20` are used in the same way in order to assign the symbolic names '`x2`' and '`x3`' to the second and third columns and set their types, bounds, and objective coefficients.

The statements `s21`—`s29` prepare non-zero elements of the constraint matrix (i.e. constraint coefficients). Row indices of each element are stored in the array `ia`, column indices are stored in the array `ja`, and numerical values of corresponding elements are stored in the array `ar`. Then the statement `s30` calls the routine `glp_load_matrix`, which loads information from these three arrays into the problem object.

Now all data have been entered into the problem object, and therefore the statement `s31` calls the routine `glp_simplex`, which is a driver to the simplex method, in order to solve the LP problem. This routine finds an optimal solution and stores all relevant information back into the problem object.

The statement `s32` obtains a computed value of the objective function, and the statements `s33`—`s35` obtain computed values of structural variables (columns), which correspond to the optimal basic solution found by the solver.

The statement `s36` writes the optimal solution to the standard output. The printout may look like follows:

```
*     0:   objval =   0.000000000e+00   infeas =   0.000000000e+00 (0)
*     2:   objval =   7.333333333e+02   infeas =   0.000000000e+00 (0)
OPTIMAL SOLUTION FOUND

z = 733.333; x1 = 33.3333; x2 = 66.6667; x3 = 0
```

Finally, the statement `s37` calls the routine `glp_delete_prob`, which frees all the memory allocated to the problem object.

2.4 Compiling

The GLPK package has the only header file `glpk.h`, which should be available on compiling a C (or C++) program using GLPK API routines.

If the header file is installed in the default location `/usr/local/include`, the following typical command may be used to compile, say, the example C program described above with the GNU C compiler:

```
$ gcc -c sample.c
```

If `glpk.h` is not in the default location, the corresponding directory containing it should be made known to the C compiler through `-I` option, for example:

```
$ gcc -I/foo/bar/glpk-4.15/include -c sample.c
```

In any case the compilation results in an object file `sample.o`.

2.5 Linking

The GLPK library is a single file `libglpk.a`. (On systems which support shared libraries there may be also a shared version of the library `libglpk.so`.)

If the library is installed in the default location `/usr/local/lib`, the following typical command may be used to link, say, the example C program described above against with the library:

```
$ gcc sample.o -lglpk -lm
```

If the GLPK library is not in the default location, the corresponding directory containing it should be made known to the linker through `-L` option, for example:

```
$ gcc -L/foo/bar/glpk-4.15 sample.o -lglpk -lm
```

Depending on configuration of the package linking against with the GLPK library may require the following optional libraries:

`-lgmp` the GNU MP bignum library;
`-lz` the zlib data compression library;
`-lltdl` the GNU ltdl shared support library.

in which case corresponding libraries should be also made known to the linker, for example:

```
$ gcc sample.o -lglpk -lz -lltdl -lm
```

2.6 Basic API Routines

This chapter describes GLPK API routines intended for using in application programs.

Library header All GLPK API data types and routines are defined in the header file `glpk.h`. It should be included in all source files which use GLPK API, either directly or indirectly through some other header file as follows:

```
#include <glpk.h>
```

Error handling If some GLPK API routine detects erroneous or incorrect data passed by the application program, it writes appropriate diagnostic messages to the terminal and then abnormally terminates the application program. In most practical cases this allows to simplify programming by avoiding numerous checks of return codes. Thus, in order to prevent crashing the application program should check all data, which are suspected to be incorrect, before calling GLPK API routines.

Should note that this kind of error handling is used only in cases of incorrect data passed by the application program. If, for example, the application program calls some GLPK API routine to read data from an input file and these data are incorrect, the GLPK API routine reports about error in the usual way by means of the return code.

Thread safety Currently GLPK API routines are non-reentrant and therefore cannot be used in multi-threaded programs.

Array indexing Normally all GLPK API routines start array indexing from 1, not from 0 (except the specially stipulated cases). This means, for example, that if some vector x of the length n is passed as an array to some GLPK API routine, the latter expects vector components to be placed in locations `x[1]`, `x[2]`, ..., `x[n]`, and the location `x[0]` normally is not used.

In order to avoid indexing errors it is most convenient and most reliable to declare the array `x` as follows:

```
double x[1+n];
```

or to allocate it as follows:

```
double *x;
. . .
x = calloc(1+n, sizeof(double));
```

In both cases one extra location `x[0]` is reserved that allows passing the array to GLPK routines in a usual way.

2.7 Problem object

All GLPK API routines deal with so called *problem object*, which is a program object of type `glp_prob` and intended to represent a particular LP or MIP instance.

The type `glp_prob` is a data structure declared in the header file `glpk.h` as follows:

```
typedef struct { ... } glp_prob;
```

Problem objects (i.e. program objects of the `glp_prob` type) are allocated and managed internally by the GLPK API routines. The application program should never use any members of the `glp_prob` structure directly and should deal only with pointers to these objects (that is, `glp_prob *` values).

The problem object consists of five segments, which are:

- problem segment,
- basis segment,
- interior point segment,
- MIP segment, and
- control parameters and statistics segment.

Problem segment

The *problem segment* contains original LP/MIP data, which corresponds to the problem formulation (1.1)—(1.3). It includes the following components:

- rows (auxiliary variables),
- columns (structural variables),
- objective function, and
- constraint matrix.

Rows and columns have the same set of the following attributes:

- ordinal number,
- symbolic name (1 up to 255 arbitrary graphic characters),
- type (free, lower bound, upper bound, double bound, fixed),
- numerical values of lower and upper bounds,
- scale factor.

Ordinal numbers are intended for referencing rows and columns. Row ordinal numbers are integers $1, 2, \ldots, m$, and column ordinal numbers are integers $1, 2, \ldots, n$, where m and n are, respectively, the current number of rows and columns in the problem object.

Symbolic names are intended for informational purposes. They also can be used for referencing rows and columns.

Types and bounds of rows (auxiliary variables) and columns (structural variables) are explained above (see Section 2.1, page 2).

Scale factors are used internally for scaling rows and columns of the constraint matrix.

Information about the *objective function* includes numerical values of objective coefficients and a flag, which defines the optimization direction (i.e. minimization or maximization).

The *constraint matrix* is a $m \times n$ rectangular matrix built of constraint coefficients a_{ij}, which defines the system of linear constraints (1.2) (see Section 2.1, page 2). This matrix is stored in the problem object in both row-wise and column-wise sparse formats.

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

Basis segment

The *basis segment* of the problem object keeps information related to the current basic solution. It includes:

- row and column statuses,
- basic solution statuses,
- factorization of the current basis matrix, and
- basic solution components.

The *row and column statuses* define which rows and columns are basic and which are non-basic. These statuses may be assigned either by the application program of by some API routines. Note that these statuses are always defined independently on whether the corresponding basis is valid or not.

The *basic solution statuses* include the *primal status* and the *dual status*, which are set by the simplex-based solver once the problem has been solved. The primal status shows whether a primal basic solution is feasible, infeasible, or undefined. The dual status shows the same for a dual basic solution.

The *factorization of the basis matrix* is some factorized form (like LU-factorization) of the current basis matrix (defined by the current row and column statuses). The factorization is used by the simplex-based

solver and kept when the solver terminates the search. This feature allows efficiently reoptimizing the problem after some modifications (for example, after changing some bounds or objective coefficients). It also allows performing the post-optimal analysis (for example, computing components of the simplex table, etc.).

The *basic solution components* include primal and dual values of all auxiliary and structural variables for the most recently obtained basic solution.

Interior point segment

The *interior point segment* is automatically allocated after the problem has been solved using the interior point solver. It contains interior point solution components, which include the solution status, and primal and dual values of all auxiliary and structural variables.

MIP segment

The *MIP segment* is used only for MIP problems. This segment includes:

- column kinds,
- MIP solution status, and
- MIP solution components.

The *column kinds* define which columns (i.e. structural variables) are integer and which are continuous.

The *MIP solution status* is set by the MIP solver and shows whether a MIP solution is integer optimal, integer feasible (non-optimal), or undefined.

The *MIP solution components* are computed by the MIP solver and include primal values of all auxiliary and structural variables for the most recently obtained MIP solution.

Note that in case of MIP problem the basis segment corresponds to the optimal solution of LP relaxation, which is also available to the application program.

Currently the search tree is not kept in the MIP segment. Therefore if the search has been terminated, it cannot be continued.

2.8 Using the callback routine

The GLPK MIP solver based on the branch-and-cut method allows the application program to control the solution process. This is attained by means of the user-defined callback routine, which is called by the solver at various points of the branch-and-cut algorithm.

The callback routine passed to the MIP solver should be written by the user and has the following specification:[5]

```
void foo_bar(glp_tree *tree, void *info);
```

where `tree` is a pointer to the data structure `glp_tree`, which should be used on subsequent calls to branch-and-cut interface routines, and `info` is a transit pointer passed to the routine `glp_intopt`, which may be used by the application program to pass some external data to the callback routine.

The callback routine is passed to the MIP solver through the control parameter structure `glp_iocp` (see Chapter "Basic API Routines", Section "Mixed integer programming routines", Subsection "Solve MIP problem with the branch-and-cut method") as follows:

```
glp_prob *mip;
glp_iocp parm;
. . .
glp_init_iocp(&parm);
. . .
parm.cb_func = foo_bar;
```

[5] The name `foo_bar` used here is a placeholder for the callback routine name.

```
parm.cb_info = ... ;
ret = glp_intopt(mip, &parm);
. . .
```

To determine why it is being called by the MIP solver the callback routine should use the routine `glp_ios_reason` (described in this section below), which returns a code indicating the reason for calling. Depending on the reason the callback routine may perform necessary actions to control the solution process.

The reason codes, which correspond to various point of the branch-and-cut algorithm implemented in the MIP solver, are described in Subsection "Reasons for calling the callback routine" below.

To ignore calls for reasons, which are not processed by the callback routine, it should just return to the MIP solver doing nothing. For example:

```
void foo_bar(glp_tree *tree, void *info)
{     . . .
      switch (glp_ios_reason(tree))
      {  case GLP_IBRANCH:
            . . .
            break;
         case GLP_ISELECT:
            . . .
            break;
         default:
            /* ignore call for other reasons */
            break;
      }
      return;
}
```

To control the solution process as well as to obtain necessary information the callback routine may use the branch-and-cut API routines described in this chapter. Names of all these routines begin with '`glp_ios_`'.

2.9 Branch and cut algorithm

This section gives a schematic description of the branch-and-cut algorithm as it is implemented in the GLPK MIP solver .

1. Initialization

Set $L := \{P_0\}$, where L is the *active list* (i.e. the list of active sub-problems), P_0 is the original MIP problem to be solved.

Set $z^{best} := +\infty$ (in case of minimization) or $z^{best} := -\infty$ (in case of maximization), where z^{best} is *incumbent value*, i.e. an upper (minimization) or lower (maximization) global bound for z^{opt}, the optimal objective value for P^0.

2. Sub-problem selection

If $L = \varnothing$ then GO TO 9. Select $P \in L$, i.e. make active sub-problem P current.

3. Solving LP relaxation

Solve P^{LP}, which is LP relaxation of P.

If P^{LP} has no primal feasible solution then GO TO 8.

Let z^{LP} be the optimal objective value for P^{LP}.

If $z^{LP} \geq z^{best}$ (in case of minimization) or $z^{LP} \leq z^{\text{best}}$ (in case of maximization) then GO TO 8.

4. Adding "lazy" constraints

Let x^{LP} be the optimal solution to P^{LP}.

If there are "lazy" constraints (i.e. essential constraints not included in the original MIP problem P_0), which are violated at the optimal point x^{LP}, add them to P, and GO TO 3.

5. Check for integrality

Let x_j be a variable, which is required to be integer, and let $x_j^{LP} \in x^{LP}$ be its value in the optimal solution to P^{LP}.

If x_j^{LP} are integral for all integer variables, then a better integer feasible solution is found. Store its components, set $z^{best} := z^{LP}$, and GO TO 8.

6. Adding cutting planes

If there are cutting planes (i.e. valid constraints for P), which are violated at the optimal point x^{LP}, add them to P, and GO TO 3.

7. Branching

Select *branching variable* x_j, i.e. a variable, which is required to be integer, and whose value $x_j^{LP} \in x^{LP}$ is fractional in the optimal solution to P^{LP}.

Create new sub-problem P^D (so called *down branch*), which is identical to the current sub-problem P with exception that the upper bound of x_j is replaced by $\lfloor x_j^{LP} \rfloor$. (For example, if $x_j^{LP} = 3.14$, the new upper bound of x_j in the down branch will be $\lfloor 3.14 \rfloor = 3$.)

Create new sub-problem P^U (so called *up branch*), which is identical to the current sub-problem P with exception that the lower bound of x_j is replaced by $\lceil x_j^{LP} \rceil$. (For example, if $x_j^{LP} = 3.14$, the new lower bound of x_j in the up branch will be $\lceil 3.14 \rceil = 4$.)

Set $L := (L \backslash \{P\}) \cup \{P^D, P^U\}$, i.e. remove the current sub-problem P from the active list L and add two new sub-problems P^D and P^U to it. Then GO TO 2.

8. Pruning

Remove from the active list L all sub-problems (including the current one), whose local bound $\widetilde{z}$ is not better than the global bound z^{best}, i.e. set $L := L \backslash \{P\}$ for all P, where $\widetilde{z} \geq z^{best}$ (in case of minimization) or $\widetilde{z} \leq z^{best}$ (in case of maximization), and then GO TO 2.

The local bound $\widetilde{z}$ for sub-problem P is an lower (minimization) or upper (maximization) bound for integer optimal solution to *this* sub-problem (not to the original problem). This bound is local in the sense that only sub-problems in the sub-tree rooted at node P cannot have better integer feasible solutions. Note that the local bound is not necessarily the optimal objective value to LP relaxation P^{LP}.

9. Termination

If $z^{best} = +\infty$ (in case of minimization) or $z^{best} = -\infty$ (in case of maximization), the original problem P_0 has no integer feasible solution. Otherwise, the last integer feasible solution stored on step 5 is the integer optimal solution to the original problem P_0 with $z^{opt} = z^{best}$. STOP.

2.10 The search tree

On the branching step of the branch-and-cut algorithm the current sub-problem is divided into two[6] new sub-problems, so the set of all sub-problems can be represented in the form of a rooted tree, which is called the *search* or *branch-and-bound* tree. An example of the search tree is shown on Fig. 1. Each node of the search tree corresponds to a sub-problem, so the terms 'node' and 'sub-problem' may be used synonymously.

In GLPK each node may have one of the following four statuses:

[6] In more general cases the current sub-problem may be divided into more than two sub-problems. However, currently such feature is not used in GLPK.

- *current node* is the active node currently being processed;
- *active node* is a leaf node, which still has to be processed;
- *non-active node* is a node, which has been processed, but not fathomed;
- *fathomed node* is a node, which has been processed and fathomed.

In the data structure representing the search tree GLPK keeps only current, active, and non-active nodes. Once a node has been fathomed, it is removed from the tree data structure.

Being created each node of the search tree is assigned a distinct positive integer called the *sub-problem reference number*, which may be used by the application program to specify a particular node of the tree. The root node corresponding to the original problem to be solved is always assigned the reference number 1.

2.11 Current sub-problem

The current sub-problem is a MIP problem corresponding to the current node of the search tree. It is represented as the GLPK problem object (`glp_prob`) that allows the application program using API routines to access its content in the standard way. If the MIP presolver is not used, it is the original problem object passed to the routine `glp_intopt`; otherwise, it is an internal problem object built by the MIP presolver.

Note that the problem object is used by the MIP solver itself during the solution process for various purposes (to solve LP relaxations, to perform branching, etc.), and even if the MIP presolver is not used, the current content of the problem object may differ from its original content. For example, it may have additional rows, bounds of some rows and columns may be changed, etc. In particular, LP segment of the problem object corresponds to LP relaxation of the current sub-problem. However, on exit from the MIP solver the content of the problem object is restored to its original state.

To obtain information from the problem object the application program may use any API routines, which do not change the object. Using API routines, which change the problem object, is restricted to stipulated cases.

2.12 The cut pool

The *cut pool* is a set of cutting plane constraints maintained by the MIP solver. It is used by the GLPK cut generation routines and may be used by the application program in the same way, i.e. rather than to add cutting plane constraints directly to the problem object the application program may store them to the cut pool. In the latter case the solver looks through the cut pool, selects efficient constraints, and adds them to the problem object.

2.13 Reasons for calling the callback routine

The callback routine may be called by the MIP solver for the following reasons.

Request for sub-problem selection The callback routine is called with the reason code `GLP_ISELECT` if the current sub-problem has been fathomed and therefore there is no current sub-problem.

In response the callback routine may select some sub-problem from the active list and pass its reference number to the solver using the routine `glp_ios_select_node`, in which case the solver continues the search from the specified active sub-problem. If no selection is made by the callback routine, the solver uses a backtracking technique specified by the control parameter `bt_tech`.

To explore the active list (i.e. active nodes of the branch-and-bound tree) the callback routine may use the routines `glp_ios_next_node` and `glp_ios_prev_node`.

Request for preprocessing The callback routine is called with the reason code `GLP_IPREPRO` if the current sub-problem has just been selected from the active list and its LP relaxation is not solved yet.

In response the callback routine may perform some preprocessing of the current sub-problem like tightening bounds of some variables or removing bounds of some redundant constraints.

Request for row generation The callback routine is called with the reason code `GLP_IROWGEN` if LP relaxation of the current sub-problem has just been solved to optimality and its objective value is better than the best known integer feasible solution.

In response the callback routine may add one or more "lazy" constraints (rows), which are violated by the current optimal solution of LP relaxation, using API routines `glp_add_rows`, `glp_set_row_name`, `glp_set_row_bnds`, and `glp_set_mat_row`, in which case the solver will perform re-optimization of LP relaxation. If there are no violated constraints, the callback routine should just return.

Optimal solution components for LP relaxation can be obtained with API routines `glp_get_obj_val`, `glp_get_row_prim`, `glp_get_row_dual`, `glp_get_col_prim`, and `glp_get_col_dual`.

Request for heuristic solution The callback routine is called with the reason code `GLP_IHEUR` if LP relaxation of the current sub-problem being solved to optimality is integer infeasible (i.e. values of some structural variables of integer kind are fractional), though its objective value is better than the best known integer feasible solution.

In response the callback routine may try applying a primal heuristic to find an integer feasible solution,[7] which is better than the best known one. In case of success the callback routine may store such better solution in the problem object using the routine `glp_ios_heur_sol`.

Request for cut generation The callback routine is called with the reason code `GLP_ICUTGEN` if LP relaxation of the current sub-problem being solved to optimality is integer infeasible (i.e. values of some structural variables of integer kind are fractional), though its objective value is better than the best known integer feasible solution.

In response the callback routine may reformulate the *current* sub-problem (before it will be splitted up due to branching) by adding to the problem object one or more *cutting plane constraints*, which cut off the fractional optimal point from the MIP polytope.[8]

Adding cutting plane constraints may be performed in two ways. One way is the same as for the reason code `GLP_IROWGEN` (see above), in which case the callback routine adds new rows corresponding to cutting plane constraints directly to the current sub-problem.

The other way is to add cutting plane constraints to the *cut pool*, a set of cutting plane constraints maintained by the solver, rather than directly to the current sub-problem. In this case after return from the callback routine the solver looks through the cut pool, selects efficient cutting plane constraints, adds them to the current sub-problem, drops other constraints, and then performs re-optimization.

Request for branching The callback routine is called with the reason code `GLP_IBRANCH` if LP relaxation of the current sub-problem being solved to optimality is integer infeasible (i.e. values of some structural variables of integer kind are fractional), though its objective value is better than the best known integer feasible solution.

In response the callback routine may choose some variable suitable for branching (i.e. integer variable, whose value in optimal solution to LP relaxation of the current sub-problem is fractional) and pass its ordinal number to the solver using the routine `glp_ios_branch_upon`, in which case the solver splits the current sub-problem in two new sub-problems and continues the search. If no choice is made by the callback routine, the solver uses a branching technique specified by the control parameter `br_tech`.

Better integer solution found The callback routine is called with the reason code `GLP_IBINGO` if LP relaxation of the current sub-problem being solved to optimality is integer feasible (i.e. values of all structural variables of integer kind are integral within the working precision) and its objective value is better than the best known integer feasible solution.

Optimal solution components for LP relaxation can be obtained in the same way as for the reason code `GLP_IROWGEN` (see above).

Components of the new MIP solution can be obtained with API routines `glp_mip_obj_val`, `glp_mip_row_val`, and `glp_mip_col_val`. Note, however, that due to row/cut generation there may be additional rows in the problem object.

[7] Integer feasible to the original MIP problem, not to the current sub-problem.

[8] Since these constraints are added to the current sub-problem, they may be globally as well as locally valid.

The difference between optimal solution to LP relaxation and corresponding MIP solution is that in the former case some structural variables of integer kind (namely, basic variables) may have values, which are close to nearest integers within the working precision, while in the latter case all such variables have exact integral values.

The reason `GLP_IBINGO` is intended only for informational purposes, so the callback routine should not modify the problem object in this case.

3 How to provide heuristic solution using callback routine ?

GLPK provide a lot of routines and methods for problem resolution but we will focus on the callback system offered by GLPK for adding cuts (or sub-problems) into its search tree.

3.1 Three functions

In order to do this, we need to understand three functions of GLPK:

- *glp_init_iocp(&param)* this routine initializes control parameters, which are used by the branch-and-cut solver, with default values. Default values of the control parameters are stored in a *glp_iocp* structure, which the parameter *param* points to. These parameters can been changed and can specify the callback routine defined by the user with the following code: *param.cb_func = callback* where *callback* is the name of the callback function.
- *glp_ios_reason*: this routine returns a code, which indicates why the userdefined callback routine is being called:
  - GLP_ISELECT request for sub-problem selection;
  - GLP_IPREPRO request for preprocessing;
  - GLP_IROWGEN request for row generation;
  - GLP_IHEUR request for heuristic solution;
  - GLP_ICUTGEN request for cut generation;
  - GLP_IBRANCH request for branching;
  - GLP_IBINGO better integer solution found.
- *glp_ios_heur_sol*: this routine can be called from the user-defined callback routine in response to the reason GLP_IHEUR to provide an integer feasible solution found by a primal heuristic. Primal values of all variables (columns) found by the heuristic should be placed in locations $x[1], ..., x[n]$, where n is the number of columns in the original problem object. Note that the routine *glp_ios_heur_sol* does not check primal feasibility of the solution provided.

  Using the solution passed in the array x the routine computes value of the objective function. If the objective value is better than the best known integer feasible solution, the routine computes values of auxiliary variables (rows) and stores all solution components in the problem object.

3.2 Sample code

```
// Callback function example
void callback(glp_tree *tree, void *info){
    switch(glp_ios_reason(tree))    {
        case GLP_IHEUR: glp_ios_heur_sol(tree, an_array);break;
        case GLP_IBINGO: printf("Bingo"); break;
        default: break;
    }}

// Main function code sample
```

```
int ret;
// Create the problem
lp = glp_create_prob();
tran = glp_mpl_alloc_wksp();
// Read the model from a_file.mod
ret = glp_mpl_read_model(tran, "a_file.mod", 1);
if (ret != 0) fprintf(stderr, "Error on translating model\n");
// Read data from a_file.dat
ret = glp_mpl_read_data(tran, "a_file.dat");
if (ret != 0) fprintf(stderr, "Error on reading data\n");
// Populate the problem
ret = glp_mpl_generate(tran, NULL);
if (ret != 0) fprintf(stderr, "Error on generating model\n");
// Build the problem
glp_mpl_build_prob(tran, lp);
glp_init_iocp(&param);
// Specify the callback function
param.cb_func = callback;
// Use simplex method
glp_simplex(lp, NULL);
// Compute integer optimization
glp_intopt(lp, &param);
ret = glp_mpl_postsolve(tran, lp, GLP_MIP);
if (ret != 0) fprintf(stderr, "Error on postsolving model\n");
```

3.3 How to presolve ?

Solvers use a presolve algorithm in order to determine an initial lower/upper bound. We can easily replace the presolve algorithm provided by GLPK with a userdefined one. In fact, we have to check if the search tree contains only one sub-problem. If it is true, we need to apply the userdefined algorithm providing an initial lower/upper bound.

```
// Callback function example
void callback(glp_tree *tree, void *info){
    if(glp_ios_reason(tree) == GLP_IHEUR && glp_ios_curr_node(tree) == 1) {
    // find an integer feasible solution with heuristic
    // and give it to the solver
    glp_ios_heur_sol(tree, an_array);break;
}}
```

4 How to use relaxed formulation of a NP-Complete problem for exact resolution ?

Ok, in this part I will be considered as a theory terrorist !! But, I will show with an example of TSP problem formulation and heuristics how you can reduce the time of resolution for this problem. It is not the best way but it can be applied for other problems.

The TSP problem is a well known problem and, in fact, it is NP-Complete.

4.1 TSP problem

The Travelling Salesman Problem (TSP) is an NP-hard problem in combinatorial optimization studied in operations research and theoretical computer science. Given a list of cities and their pairwise distances, the task is to find a shortest possible tour that visits each city exactly once. The problem was first formulated as a mathematical problem in 1930 and is one of the most intensively studied problems in optimization.

It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, a large number of heuristics and exact methods are known, so that some instances with tens of thousands of cities can be solved.

The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept city represents, for example, customers, soldering points, or DNA fragments, and the concept distance represents travelling times or cost, or a similarity measure between DNA fragments. In many applications, additional constraints such as limited resources or time windows make the problem considerably harder. In the theory of computational complexity, the decision version of the TSP belongs to the class of NP-complete problems. Thus, it is assumed that there is no efficient algorithm for solving TSPs. In other words, it is likely that the worst case running time for any algorithm for the TSP increases exponentially with the number of cities, so most instances with only one hundred cities will take upwards of 10^{148} CPU years to solve exactly.

4.2 Relaxed problem formulation

Here, we want to relaxe the TSP problem in order to speed-up the resolution. A specific algorithm will apply the removed constraints after each LP resolution given by GLPK.

So, we will use this very simple formulation ! We have a list of cities C and a distance vector $distance[i \in C, j \in C]$ as input. We introduce a binary vector $visited[i \in C, j \in C]$ which represents roads between cities (1 if the road is used, 0 otherwise) and an integer vector $neighbor[i \in C]$ which represents the degree of each city. We add a constraint which specifies that each city has to have a degree equal to 2. The goal is to find the shortest tour containing all cities. So, the MathProg formulation is the following one:

```
param n, integer, >= 3;
/* node count */
set C, default {1..n};
/* nodes (cities) */
set E, within C cross C;
/* edges (roads) */
param distance{(i,j) in E};
/* distance between i and j */
var visited{(i,j) in E}, binary;
/* visited[i,j] = 1 if i neighbor of j */
minimize length: sum{(i,j) in E} distance[i,j] * visited[i,j];
/* minimize the tour length */
s.t. neighbor{i in C}: sum{(k,i) in E} visited[k,i]
                         + sum{(i,j) in E} visited[i,j] = 2;
/* 2 neighbors for each node */ end;
```

It is clear that this formulation will implies disjoint cycles as solution, but this problem is very simple to solve.

4.3 Branch & Bound

In order to solve this problem, we will use a simple branch and bound algorithm as follow:

```
1  (1) Initial solution
2      Solve the LP with GLPK
3      Save initial solution with sub-cycle
4      Save the lower bound


5  (2) Branching
6      Find the shortest sub-cycle of the current sub-problem
7      For each edge of the sub-cycle
8          Add a constraint which bans this edge
9          Add this new sub-problem into the resolution tree
10     End


11 (3) Branch and Bound
12     While it exists sub-problems
13         Select the best sub-problem
14         Evaluate with GLPK
15         If the solution is greater than the current lower bound
16             Remove the sub-problem
17         Else
18             Update the new lower bound
19             If the solution contains sub-cycle
20                 Go to (2)
21             Else
22                 It is a new feasible solution !
23             End
24         End
25     End
26     Return the best solution
```

Figure 1: **Branch & Bound** algorithm

5 Complete example with MFAS problem

5.1 Minimum Feedback Arc Set problem

In graph theory, a directed graph may contain directed cycles, a one-way loop of edges. In some applications, such cycles are undesirable, and we wish to eliminate them and obtain a directed acyclic graph (DAG). One way to do this is simply to drop edges from the graph to break the cycles. A feedback arc set (FAS) or feedback edge set is a set of edges which, when removed from the graph, leave a DAG. Put another way, it's a set containing at least one edge of every cycle in the graph.

Closely related are the feedback vertex set, which is a set of vertices containing at least one vertex from every cycle in the directed graph, and the minimum spanning tree, which is the undirected variant of the feedback arc set problem.

5.2 MIP formulation

For this MIP formuation, we introduce several variables:

- n an integer variable which represents the number of vertex
- V an integer vector $V_0, \dots V_n$ which represents the set of vertices

- E an integer matrix $E_{i,j} \forall i, j \in V \mid E_{i,j}$ exists which represents the set of arcs
- x a binary vector $x_{(i,j)} \forall (i,j) \in E$ which is set to 1 if (i,j) is a feedback arc, 0 otherwise
- k a binary vector $k_v \forall v \in V$ which is an order over the vertices

It is known that a digraph $G = (V, E)$ is acyclic if and only if its vertices can be assigned numbers from 1 to $|V|$ in such a way that $k[i] + 1 \leq k[j]$ for every arc $(i,j) \in E$, where $k[i]$ is a number assigned to vertex i. We may use this condition to require that the digraph $G = (V, E \backslash E')$, where E' is a subset of feedback arcs, is acyclic.

LP formulation

- Minimize $\sum_{(i,j) \in E} x_{(i,j)}$
- Such that:
  - vertices ordering:
    $\forall v \in V$,
    $$1 \leq k_v \leq |V|,$$
  - feedback arc detection:
    $\forall (i,j) \in E$,
    $$r_{(i,j)} = k_i - k_j,$$
    $$r_{(i,j)} \geq 1 - |V| \times x_{(i,j)},$$

5.3 Optimization scheme

In order to solve this problem, we will use the previous MIP formulation and use the following heuristic for providing initial feasible solution. The heuristic will be computed only one time at the beginning of the resolution.

5.4 Heuristic for providing initial solution

For a given digraph $G = (V, A)$
Let *mip_col* be an array which will contain values for all columns of the initial MIP
Let *parent* be an array containing for each vertex its parent in DFS search

rank $\leftarrow 1$
for all vertex $v \in V$ **do**
  $mip\_col[k_v] \leftarrow rank$
  $rank \leftarrow +1$
  compute DFS(v, *parent*)
end for

for all vertex $v \in V$ **do**
  for all $u \in N^+(v)$ **do**
    if $u \in parent[v]$ **then**
      $mip\_col[x_{u,v}] \leftarrow 1$
    end if
  end for
end for

Figure 2: Heuristic used to provide initial solution

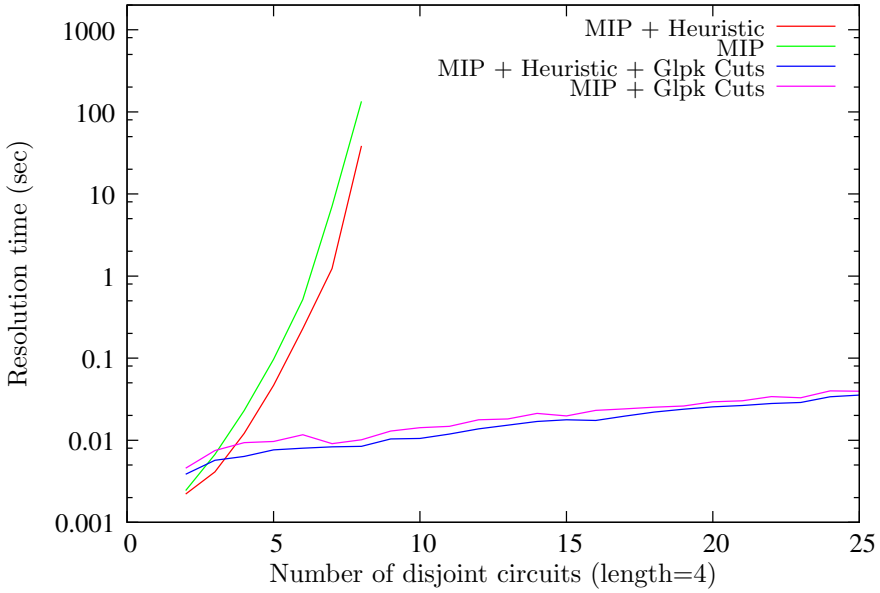
5.5 Impact of bad lower bounds - relaxed problem

While this formulation of the MFAS problem as a Mixed Integer Linear Program is perfectly sound, one quickly notices that very simple instances require an unreasonable amount of computing time. The explanation lies in the fact that if the solvers are usually able to find a satisfiable (and sometimes optimal) solution very early in their computations, they are not willing to return it until they get a proof of optimality, which is what drains their energy. When tested against a circuit (for which the MFAS is or cardinality exactly 1) the relaxed version of this formulation will give the correct lower bound. It would, however, give the same answer for the disjoint union of 10 circuits, an instance for which solving the problem would already take time. Such poor results are probably enough to justify the time very easy computations require, and in any case a reason to look for ways to improve the formulation by producing lower bounds during the solving process (through the addition of new constraints or heuristics). Alternatively, one could also prefer to solve the MFAS problem through the following formulation (at the cost of an exponential number of constraint, and so using constraint generation).

5.6 Results

In this part, we focus on the efficiency of this method. In fact, we use an heuristic for providing initial integer feasible solution.

The MIP formulation that we use implies very bad lower bound and larger the number of circuit in the instance is, harder is the resolution. So, the instances created for testing this method are based on disjoint circuits.



6 Tips related to GLPK

6.1 How to get a column by its name

Sometimes it could be useful to access a column by its name, for example $x[1,3]$. The following sample code show how to do this:

```
lp = glp_create_prob(); // create the glpk problem object
... // read model, data, ...
glp_create_index(lp); // create the index of columns for searching them by name
...
sprintf(col_name, "x[%d,%d]", u, i); // create the char * which contains the name of the column
glp_find_col(lp, col_name); // return the index of the column
```