Université Sorbonne Paris Nord
Master EUR
Mathematical Optimization and Polyhedral Approaches

Practical Session 1&2

Solving Mixed Linear Integer Programs with Julia and GLPK

Ces Travaux pratiques ont pour but d'introduire la mise en œuvre de résolution de problèmes d'optimisation combinatoire en utilisant des programmes linéaires en nombres entiers (Mixed Integer Programs, MIPs). Ils sont proposés en langage Julia utilisant le package JuMP qui sert d'interface au solveur MIP. D'autres package Julia seront utilisés pour manipuler des graphes.

Q 0.1**.** Récupérer l'archive du TP qui se dézipe en créant un répertoire `TP_MOPA` contenant des fichiers .jl (langage Julia) et quelques autres fichiers.
Le sous-répertoire Instances propose des instances pour les différents exemples et exercices.
Les répertoires cités dans le TP seront à partir de la racine du répertoire `TP_MOPA`.

1 Julia framework

The system used for these practical sessions is LINUX without a development environment, but it is possible to adapt it to other systems. The general principles presented here could just as well be implemented using several solvers such as GLPK, Gurobi, or Cplex. Indeed, these solvers have (partial) interfaces in many languages (C, C++, Java, Python, dedicated language, etc.).

1.1 The Julia Language

Julia is a programming language designed at MIT [1]. The goal of this language is to be both a high-performance and high-level language (with a programming ease similar to Python), with the aim of being used in scientific projects. Thus Julia includes fast libraries (for example for algebra coded in C and Fortran) as well as many packages, all open source, allowing to extend its use. In the long term, the creators of Julia aim for it to replace Python, as well as scientific languages/software like Matlab, Scilab, or R.

Julia is based on the principle of "just-in-time compilation" (JIT compilation). That is, the lines of code are compiled into machine language by the Julia program before being executed (like C) but by doing it line by line, which allows for a usage close to Python but much faster than the latter.

1. The Massachusetts Institute of Technology (MIT) is one of the university places that drives technological innovation

1.2 Launching Julia

Similar to virtual machines (like Java), Julia is a software that will compile and execute your program written in Julia. Thus, launching Julia and setting up the required packages is a bit long. Once launched, however, the Julia software allows you to run several programs in a previously loaded configuration. Moreover, functions that are already compiled are already known and do not need to be recompiled... Pay attention to the compilation order : if you use several files, you must ensure that all files are compiled from their latest saved versions.

Julia is already installed on the machines of the Galilée institute on the account pierre.fouilhoux. You just need to type
- add at the end of your .bashrc :

```
export PATH=$PATH:/export/home/users/Enseignants/pierre.fouilhoux/julia-1.6.2/bin/
export JULIA_DEPOT_PATH="/export/home/users/Enseignants/pierre.fouilhoux/.julia-1.6.2"
```

- restart your terminal
- then use `julia-1.6.2` **AND NOT** `julia` **alone !**

A long error message might appear at the beginning talking about "log file", it's not serious : it ends with a sentence indicating that this kind of message is disabled for the rest.

1.3 A first program in Julia

Download from the practical session website the file `example1.jl` which provides a first example with a function and a for loop.

We will use Julia in **Julia command line mode :**
by first launching `julia-1.6.2`.
Then by typing `include(''example1.jl'')`.
The function contained in the .jl file is now uploaded. You can use it !
For instance by typing `Print_n_times(10,''Hello'')`.

There, Julia has been launched once and for all, the program is compiled and executed. Julia then remains active... as well as all the functions you have just loaded and compiled.

2 MILP with the JuMP Package

The Julia language has given rise to many open-source projects created all over the world.
We will use the **JuMP** package `https://jump.dev` which allows
- modeling linear programs (among others)
- easy interfacing with existing linear solvers like GLPK, CPLEX, Gurobi, CBC,...

2.1 Introduction to GLPK

GLPK is a set of algorithms for solving various problems ranging from linear programming to integer programming. It is an open-source software under the GNU free license. This software is not very powerful, more geared towards teaching, but you may prefer commercial solvers like CPLEX or GUROBI, or academic products like SCIP or HIGHS.

These solvers can be called in several ways, here we use the Julia language as an interface with GLPK.

Algorithms inside the solver GLPK solves linear programs (LP) and mixed integer linear programs (MILP). These two types of problems are very similar in appearance but lead to quite different algorithms.

- Solving an LP can be done in polynomial time. GLPK uses the simplex method which is not polynomial but is very efficient.
- Solving a MILP is an NP-hard problem. GLPK uses the only known algorithm capable of solving all MILPs effectively : the branch-and-bound method which is exponential in the worst case.

2.2 A first LP example in JuMP

The program `example_PL.jl` provides you with a first program to manipulate a linear program with a solver.

Read and execute the program to understand the general operation.

Some remarks :

- the `Model()` command of JuMP returns a variable creating an LP in the chosen solver : thus JuMP is an interface with this solver
- multiple executions of `Model()` will return as many models : we can manage several LPs side by side
- commands starting with @ are Julia macro-commands designed for JuMP, they launch a series of functions to create the model
- the `optimize!()` command is a JuMP command but it will actually launch the optimization algorithm of the solver you chose above (GLPK or CPLEX) : it is not JuMP that solves but the solver. Thus the performance will depend on the chosen solver

3 Three LP exercises

Exercice 1 : Résolution d'un programme linéaire

Q 1.1**.** Solve the following linear program using Julia and GLPK :

$$
\begin{array}{lrcrcrcl}
max\ z = & 3x_1 & + & 2x_2 & + & 4x_3 & & \\
 & x_1 & + & x_2 & + & 2x_3 & \leq & 4 \\
 & 2x_1 & + & x_2 & + & 3x_3 & \leq & 7 \\
 & 2x_1 & & & + & 3x_3 & \leq & 5 \\
 & x_i \geq 0, & \forall & i \in & \{1,\ldots,3\} & & &
\end{array}
$$

Q 1.2. Give the solution and the value of an optimal solution. Indicate which inequalities of the problem are tight (i.e., have reached their bound).

Q 1.3. Increase the right-hand side of constraint 2 by 1, what do you notice about the solution and its value ?

Exercice 2 : Decision aid for a dairy farm

noindent A dairy farmer has 60 kilo-litres (kl) of milk that he can either process into dairy products (cheeses and yogurts) or sell as cream. The dairy products are either cheeses or yogurts. The farmer has only 2600 euros to invest for the processing of his 60 kl.
The operating costs and expected gains for these three types of production are given in the following table :

	Operating costs in euro/kl	Gains in euro/kl
Cheeses	65	90
Yogurts	70	90
Cream	40	70

Q 2.1. **a)** Taking into account that the operating costs are to be deducted from the gains, write (with justification) the linear program to maximize the company's profit. **b)** Indicate the solution and verify that the profit obtained is 1800 € .

Q 2.2. Is it profitable to use at least 10 kl for dairy products ?

Q 2.3. From what selling price (per kl) does the farmer have an interest in producing cheeses ?

4 The Maximum Cardinality Stable Set Problem

4.1 A Look at `Graphe_Manip.jl`

Q 2.4. Load and read the file `Graphe_Manip.jl`.
You can find DIMACS format files in the `Instances/DIMACS` directory (these files come from the instance library for the graph coloring problem).

Q 2.4. To read a DIMACS format graph with the reading function and then visualize this graph, two functions are provided. The visualization creates a PDF representing the graph by nicely placing the vertices using the `gplot` command.

```
G=Read_undirected_Graph_DIMACS(‘‘graph_file_name.dim’’)
WritePdf_visualization_Graph(G,"graphe_file_name")
```

Do this for several graph instances.

4.2 Stable Set Problem : Integer Linear Programming (ILP)

Q 2.4. The file `‘‘MIP_StableSet.jl’’` proposes an implementation of the compact "edge" ILP model for the maximum cardinality stable set problem, seen in class. Look at the code, which specifies that the variables are binary.

Q 2.4. Solve instances from the `Instances/DIMACS` instance directory. Compare the results obtained for various sizes of graphs.

5 Exercise : The Graph Coloring Problem

Given an undirected graph $G(V, E)$, a coloration of G is to give a color to nodes so that two adjacent nodes have distinct colors.
Formally, a coloration is then a function $f : V \leftarrow \{1, \ldots, n\}$ so that $f(u) \neq f(v)$ for every edge $uv \in E$.

The graph coloring problem is to find a coloration using the less possible colors.
Let us define the binary variables :
$x_{ul} = 1$ if and only if color l is given to node u for every node $u \in V$ and colors in $\{1, \ldots, n\}$.
Note that x is a matrix with lines corresponding to nodes and columns to colors.

We can prove that x encodes a coloring if and only if
$\sum_{l=1}^{n} x_u^l = 1 \quad \forall u \in V \qquad$ *(exactly one color per edge)*
$x_u^l + x_v^l \leq 1 \quad \forall uv \in E$ and $\forall l \in \{1, \ldots, n\}$ *(not the same color for the nodes of each edge)*

Let us add the additional binary variables w_l for every color $l = 1, ..., n$ indicating wether color l has been used in a solution x.
The two variables sets are linked by the inequalities
$x_{ul} \leq w_l \quad \forall uv \in E$ and $\forall l \in \{1, \ldots, n\}$ *(w_l is set to 1 is l is used for node u)*

The following formulation is equivalent to the coloring problem.

$$\text{Min} \sum_{l=1}^{n} w_l$$

$$\sum_{l=1}^{n} x_u^l = 1 \quad \forall u \in V \tag{1}$$

$$x_u^l + x_v^l \leq w_l \quad \forall uv \in E \text{ and } \forall l \in \{1, \ldots, n\} \tag{2}$$

$$x_{ul} \leq w_l \quad \forall uv \in E \text{ and } \forall l \in \{1, \ldots, n\} \tag{3}$$

$$x_u^l \in \{0, 1\}, \quad \text{pour tout } u \in V \text{and } \forall l \in \{1, \ldots, n\}. \tag{4}$$

Q 2.4**.** Based on the stable set visualizer, create a visualization for a graph coloring for a given solution given by x. And test your heuristic.

Q 2.4**.** Create a Julia program for this MILP formulation.
Visualize the solution on several instances using the Julia help below.
Here is an example of code with double-indexed variables.

```
m = Model(CPLEX.Optimizer)
Binary variable with double indices

@variable(m, x[1:G.nb_points, 1:G.nb_points], Bin)
Integer variable with single indices

@variable(m, u[1:G.nb_points], Int)
Objective function with double sum

@objective(m, Min, sum( (sum(c[i, j] * x[i, j]) for j = 1:G.nb_points)
for i = 1:G.nb_points ) )
Constraint: "only one x_ij equals 1 among all possible x_ij with fixed i"

for i in 1:G.nb_points
@constraint(m, (sum(x[i, j] for j in 1:G.nb_points))== 1)
end
```

Q 2.4**.** Test your MIP on various sizes of graphs.