Sup Galilée - INFOA2 - Université Sorbonne Paris Nord
OLOC

Optimisation Combinatoire - Heuristiques, Méta-heuristiques et PLNE

VERSION Julia+JuMP+CPLEX

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 logiciel CPLEX. D'autres package Julia seront utilisés pour manipuler des courbes (Plots) ou des graphes.

Les exemples et exercices suivants présentent ainsi :

- Quelques exemples très simples d'heuristiques gloutonnes et de méta-heuristiques
- l'utilisation de PLNEs compacts en utilisant Julia et JuMP

Question 1 *Récupérer l'archive du TP qui se dézipe en créant un répertoire* `TP_OLOC` *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_OLOC`*.*

Table des matières

Première partie

Découverte de Julia et de JuMP

1 Cadre et mise en place

Le système utilisé pour ces TPs est LINUX sans environnement de développement, mais il est possible de l'adaptater à d'autres systèmes. Les principes généraux présentés ici pourraient tout aussi bien être mis en œuvre en utilisant Gurobi à la place de Cplex. Enfin, Cplex comme Gurobi possèdent des interfaces (partielles) dans de nombreux langages (C, C++, Java, python, langage dédié etc).

1.1 Le langage Julia

Julia est un langage de programmation conçu au MIT [1]. L'objectif de ce langage est d'être à la fois un langage performant et de haut niveau (d'une facilité de programmation similaire à Python), dans l'objectif d'être utilisé dans des projets scientifiques. Ainsi Julia comporte des bibliothèques rapides (par exemple pour l'algèbre codés en C et en Fortran) ainsi que de nombreux packages, tous en open source, permettant d'en étendre l'utilisation. A terme, les créateurs de Julia ambitionne qu'il remplace Python, mais aussi les langages/logiciels scientifiques comme Matlab, Scilab ou R.
Julia repose sur le principe de "compilation à la volée", en anglais Just-In-Time (JIT) compilation. C'est-à-dire que les lignes de code sont compilées en langage machine par le programme Julia avant d'être exécutée (comme le C) mais en le faisant nouvelle ligne par nouvelle ligne, ce qui permet une utilisation proche de Python mais bien plus rapide que ce dernier.

1.2 Lancer Julia

Similairement aux machines virtuelles (comme Java), Julia est un logiciel qui va compiler et exécuter votre programme écrit en Julia. Ainsi le lancement de Julia et la mise en place des packages requis est un peu longue. Une fois lancé, le logiciel Julia permet par contre de lancer plusieurs programmes dans une configuration chargée auparavant. De plus des fonctions déjà compilées sont déjà connues et ne sont plus à re-compiler...
Attention à l'ordre de compilation : si vous utilisez plusieurs fichiers, il faut faire attention à ce que tous les fichiers soient compilés à partir de leur dernières sauvegardes

1.2.1 Sur les machines de l'institut Galilée

Julia est déjà installé sur les machines de l'institut Galilée sur le compte pierre.fouilhoux.
Il vous suffit de taper

- d'ajouter à la fin de votre .bashrc :

1. The Massachusetts Institute of Technology (MIT) est une des lieux universitaires les moteurs dans l'innovation technonologique

`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"`

- de relancer votre terminal
- puis d'utiliser `julia-1.6.2` **ET NON** `julia` **tout court !**

Il se peut qu'un long message d'erreur vous parle de "log file" au départ, ce n'est pas grave : il se termine par une phrase vous signalant que ce genre de messages est désactivé pour la suite.

1.3 Si vous voulez l'avoir chez vous : installation de Julia

Sous Linux

Pour installer Julia sous Linux (sur x86 64 bits, ce qui est le plus courant), récupérer sur `https://julialang.org/downloads` l'archive de Julia `julia-xxxx-linux-x86_64.tar.gz` et décompresser l'archive `tar -zxvf julia-xxx-linux-x86_.tar.gz` à l'emplacement de votre choix, par exemple dans `/home/moi/logiciels`.
Le logiciel `julia` sera alors `/home/moi/logiciels/julia-xxxx/bin/julia`.
Il est utile d'ajouter à votre fichier .bashrc une ligne permettant d'accéder facilement à Julia :
Taper `gedit .basrc` dans votre répertoire utilisateur.
Ajouter tout à la fin `export PATH=$PATH:/home/moi/logiciels/julia-xxxx/bin`
Fermer et réouvrer votre terminal, à partir de maintenant, il vous suffira de taper `julia` pour lancer le programme.

Sous Max Os et Sous Windows

Vous pouvez trouver des indications pour ces deux systèmes : n'hésitez pas faire remonter vos expériences pour alimenter cette section.

1.4 Un premier programme en Julia

Téléchargez sur le site du TP, le fichier `exemple_1er_pgm.jl` qui propose un premier exemple avec une fonction et une boucle for. Les lignes qui ne sont pas dans une fonction seront exécutées et sont donc une sorte de programme principal.

1.5 Prise en main du logiciel Julia

Il existe trois modes d'utilisation du logiciel Julia

- **en ligne de commande Linux :** `julia exemple_1er_pgm.jl`.
  Ici le programme Julia se lance, compile le programme `exemple_1er_pgm.jl`, l'éxécute, puis Julia s'arrête. Comme le lancement de Julia peut être lent s'il y a des ajouts de package. Ce mode n'est pas à utiliser pour les gros projets.
- **en ligne de commande Julia :** en lançant tout d'abord `julia`. Puis en tapant `include(‘‘exemple_1er_pgm.jl’’)`. Là, Julia a été lancé une fois pour toute, le programme est compilé et exécuté. Julia reste alors actif... ainsi que toutes les fonctions que vous venez de compiler dans `exemple_1er_pgm.jl` (pour vous en convaincre, tapez par exemple : `Affiche_n_fois_une_chaine(5,"Salut")` qui va être exécutée dans la console Julia.

- **Jupyter notebook** : (pour ceux qui sont intéressés, voire l'annexe).

Que choisir en ces trois modes : à vous de voir, mais le choix **en ligne de commande Julia** sera celui utilisé dans la suite du document... Il est très pratique. En particuler, en lançant `julia` votre terminal devient le terminal du logiciel Julia. Les commandes sont donc celle du langage Julia : ce qui permet de lancer des programmes ou des lignes à la volée.

1.6 Un deuxième programme : dessiner des courbes

Le deuxième exmple `exemple_dessiner_courbe.jl` propose 2 fonctions
- la première permet de lire un fichier de données texte et de le charger dans deux tableaux
- la deuxième permet de lancer la première fonction et de créer une courbe dans un fichier pdf

La première ligne de `exemple_dessiner_courbe.jl` est `using Plots` : ce qui permet de signaler que l'on va utiliser le package Plots pour dessiner des courbes...

Mais si vous êtes sur votre machine : ce package doit être chargé auparavant.
Donc lancez les lignes permettant de charger Plots :
```
import Pkg; Pkg.add("Plots")
```

Vous pouvez exécuter
`using Plots` Vous pouvez noter que cela prend un certain temps au premier lancement. Il est donc important **en ligne de commande Julia** de le taper au début (ou de lancer une première fois un programme comprenant cette ligne), il sera alors chargé une bonne fois pour toute votre session.

Question 2 *A présent, la commande* `include("exemple_dessiner_courbe.jl")` *arrive à la fin de la compilation... Mais rien ne se passe car ce programme ne contient aucune ligne composant un programme principal.*
Pour lancer le programme, il faut lancer la fonction `Dessine_courbe` *avec le fichier de votre choix : cela peut se faire donc en* **ligne de commande Julia** *en tapant* `Dessine_courbe("donnees.txt")` *par exemple.*

1.7 Mesure du temps

L'exemple suivant pris sur `https://juliapackages.com/p/cputime` permet de comprendre comment mesurer le temps en Julia.
Si vous êtes sur votre machine, il faut ajouter `Pkg.add("CPUTime");`

```
using CPUTime
```

```
function Fonction_a_chronométrer()
    x = 0
    for i in 1:10_000_000
        x += i
    end
    sleep(1)
end
```

La fonction exemple `Fonction_a_chronométrer()` contient un temps effectif et un temps de repos forcé avec la commande `sleep`.
La commande `@time @CPUtime Fonction_a_chronométrer()` renvoie alors

```
elapsed CPU time: 0.000174 seconds
  1.002640 seconds (32 allocations: 912 bytes)
```

La macro `@CPUtime` affiche en premier le temps effectif d'utilisation du CPU par la fonction.
La macro `@time` affiche ensuite le temps réel observé de la fonction.
L'un est plus grand que l'autre : il vaut donc mieux utiliser le temps CPU car le temps réel dépend de la charge d'utilisation de votre ordinateur.
En revanche, le temps CPU est difficile à établir sur une machine multi-cœur... car il s'agit souvent d'une fonction du fabriquant de processeur qui cherche à donner le meilleur temps possible... ce qui n'est que rarement le cumul du temps des différents cœur mais plutôt une valeur bien inférieure (une sorte de moyenne, un max ?).

1.8 Pour aller plus loin en Julia

Il y a de nombreux documents synthétiques pour tout comprendre en Julia. En voici certains très utiles :
Tout en une page et en français :
`https://juliadocs.github.io/Julia-Cheat-Sheet/fr`
Un peu plus complet avec un menu pratique :
`https://syl1.gitbook.io/julia-language-a-concise-tutorial`
Et bien sûr la doc officielle
`https://docs.julialang.org/en/v1`

Deuxième partie

PL et PLNE en Julia avec JuMP et Cplex

Le langage Julia a donné lieu à de nombreux projets open-source créé un peu partout dans le monde.
Nous allons utiliser le package **JuMP** `https://jump.dev` qui permet

- de modéliser des programmes linéaires (entre autres)
- d'interfacer facilement avec des solveurs linéaires existants comme GLPL, CPLEX, Gurobi, CBC,...

2 Installation

Si vous êtes à Galilée, il n'y a rien à faire.

Mais si vous êtes sur votre machine, pour utiliser JuMP, il faut bien entendu taper une fois au début
`import Pkg; Pkg.add(''JuMP'')`
Pour que JuMP ne soit pas qu'un outil vide d'algorithme de résolution, il faut aussi ajouter un solveur : par exemple ici ce sera CPLEX.
Pour utiliser votre licence CPLEX :
Il faut auparavant avoir installer CPLEX en payant une licence (très élevée) ou en ayant une licence académique (voir annexe).
Par exemple vous pouvez le mettre dans le répertoire `/home/moi/logiciels/CPLEX_studioxxx/cplex`.
Puis définir une variable Julia vers le répertoire contenant CPLEX.

```
ENV["CPLEX_STUDIO_BINARIES"] =
"/home/moi/logiciels/CPLEX_studioxxx/cplex/bin/x86-64_linux"
Pkg.add("CPLEX")
Pkg.build("CPLEX")
```

Au début d'un programme, n'oubliez pas `using CPLEX`.

3 Un premier exemple de PL en JuMP

Le programme `exemple_PL_cplex.jl` vous propose un premier programme pour manipuler un programme linéaire avec Cplex.

Lisez et exécutez le programme pour comprendre le fonctionnement général.
Quelques remarques :

- la commande `Model()` de JuMP renvoie une variable créant un PL dans le solveur choisi : JuMP est donc une interface avec ce solveur
- plusieurs exécutions de `Model()` renveront autant de modèles : on peut gérer côte à côte plusieurs PL

- les commande commençant par un @ sont des macro-commande Julia conçu pour JuMP, elles lancent une série de fonctions permettant de créer le modèle
- la commande `optimize!()` est une commande JuMP mais elle va en fait lancer l'algorithme d'optimisation du solveur que vous avez choisi plus haut (GLPK ou CPLEX) : ce n'est pas JuMP qui résoud mais bien le solveur. Ainsi les performances vont dépendre du solveur choisi

4 A vous de jouer

4.1 Courbes paramétrées

On considère ici le programme $P(t)$, qui est très proche du PL précédent, sauf qu'il est paramétré par la valeur t.

$$(P(t)) \begin{cases} \max & 5x_1 & + & 3x_2 & & \\ s.c. & x_1 & + & x_2 & \geq & 2 \\ & 2x_1 & + & tx_2 & \leq & 6 \\ & x_1 & + & 5x_2 & \leq & 10 \\ & x_1 & & & \geq & 0 \\ & & & x_2 & \geq & 0 \end{cases}$$

Question 3 *Que peut-on dire de $P(3)$ et de $P(t)$ pour un t fixé en terme de terminologie ?*

Question 4 *En couplant l'exemple `exemple_courbe.jl` qui permet de dessiner des courbes et JuMP, dessiner la courbe de $P(t)$ en fonction de t pour t prenant des valeurs entre -10 et 10.*

Troisième partie

Le problème du stable de cardinalité maximale

Si vous êtes sur les machines de Galilée, tout est en place.
Mais si vous êtes sur votre machine : pour obtenir l'ensemble des packages utilisant des graphes : `import Pkg; Pkg.add("Graphs"); Pkg.add("SimpleWeightedGraphs"); Pkg.add("Cairo"); Pkg.add("Compose"); Pkg.add("Fontconfig"); Pkg.add("GraphPlot"); Pkg.add("Colors");`.

5 Regardons un peu `Graphe_Manip.jl`

Question 5 *Regardez et charger les fichiers* `Graphe_Manip.jl` *et* `Graph_StableSet.jl`.

Vous pouvez trouver des fichiers au format DIMACS dans le répertoire `Instances/DIMACS` *(ces fichiers sont issus de la librairie d'instances du problème de coloration de graphe).*

Question 6 *Lancer la commande de lecture d'un graphe de format DIMACS avec la fonction de lecture.*
Visualiser ces graphes avec la function créant un pdf représentant le graphe.

6 Problème du stable : Heuristiques gloutonnes et heuristique de descentes itérées

Dans cette section, on s'intéresse au problème du stable de cardinalité maximale (pour des poids égaux à 1 pour chaque sommet) : ce problème est NP-difficile. Voir le cours OLOC_6 et OLOC_7 pour une définition. Le but de cette section est de prendre en main à la fois les graphes en Julia et aussi les heuristiques/métaheuristiques

Question 7 *Lisons en détails* `Graphe_StableSet.jl`. *Il contient une heuristique gloutonne et une métatheuristique. Lancer la fonction lançant les deux méthodes l'une derrière l'autre sur quelques instances.*
Un fichier pdf donne le stable obtenu.

Question 8 *Lire attentivement le code pour comprendre ce qu'il fait. N'hésitez pas à écrire l'algorithme à la main etc.*

7 Problème du stable : PLNE

Question 9 *Regardez le modèle PLNE compact dit "aux arêtes" pour le problème du stable de cardinalité maximale dans "*`PLNE_compact_StableSet.jl`*"*

Question 10 *Résoudre des instances du répertoire d'instance* `Instances/DIMACS`. *Comparer les résultats obtenus avec l'heuristique.*

Annexes

Annexe : Jupyter Notebook

Il faut avoir préalablement installer Jupyter notebook. Puis taper dans Julia :

```
import Pkg
Pkg.add("IJulia")
notebook()
```

Annexe : Obtenir une licence académique CPLEX

Pour cela :

- *créer un compte IBMid sur la base de votre addresse universitaire*
  (attention la validation par mail avec envoi de code est lente... or il y a un délai entre les 2) `https://www.ibm.com/account/reg/fr-fr/signup?formid=urx-19776`
- *aller sur internet à l'ACADEMIC INITIATIVE d'IBM (et non à la verson free qui est limitée)* `https://www.ibm.com/academic/home`
- *loguez vous et singalez que vous êtes dans le cadre de l'ACADEMIC INITIATIVE*

- *aller à* `https://www.ibm.com/academic/technology/data-science`
  Puis sur l'onglet de milieu de page Software
  A gauche, il y a alors un quart de page CPLEX Cliquer sur Download (Il y a des browsers qui peuvent rester bloqué... si trop de protection par exemple (firefox), tenter sur des browsers plus "ouverts" (chromium-browser par exemple)
- *vous arrivez alors à la page de téléchargement* **en haut, il y a des choix dans un onglet entre IBM Download Director ou http : prendre http ! ! ! !**
- *Choisir votre version (linux, windows, MacOs...) et download*
- *Prévoyez un emplacement pour installer Cplex pour lequel vous aurez le chemin car il servira à JuMP par la suite*
- *Sur linux, il s'agit d'un binaire à lancer qui se dézipera et installera Cplex (penser à rendre le binaire exécutable si besoin avec* `chmod +x`