Université de Nantes — UFR Sciences et Techniques
Master informatique parcours "optimisation en recherche opérationnelle (ORO)"
Année académique 2018-2019

Rapport de TER

Implémentation Julia de l'algorithme simplexe multi-objectif avec la forme révisé du simplexe et la décomposition LU

Stagiaire : Arthur Gontier
Encadrants : Anthony Przybylski[1], Xavier Gandibleux[2]

5 juillet 2019

Résumé

On fais une abstract ?

Table des matières

1 Introduction

A REFAIRE

Nous savons aujourd'hui résoudre beaucoup de problèmes mono-objectifs et il existe de plus en plus d'algorithmes pour les problèmes bi-objectifs. Mais les solutions apportées aux problèmes à plus de trois objectifs sont rares du fait non pas de leur complexité mais par un manque de propriétés à exploiter. Si les algorithmes bi-objectifs peuvent parcourir les solutions paréto-optimales de manière linéaire, cela devient plus difficile pour plus de trois objectifs même si la résolution du problème reste une somme pondérée. On se retrouve alors obligé d'énumérer en évitant les cyclages toutes les bases qui donnent une solution non-dominée de manière un peut brut.
L'objet de ce travail encadré de recherche est donc d'abord la compréhension de l'algorithme du simplexe multicritère puis son implémentations avec la forme révisé du simplexe et la décomposition LU.
Ce projet est donc parti du Chapitre 7 : "multicriteria simplexe" du livre de Matthias Ehrgott "Multicriteria optimization" [1]

2 Définitions et notations

2.1 Programme linéaire

Un programme linéaire est la formulation d'un problème avec une fonction objectif $x^T c$ et des contraintes $Ax \leq b$ sous forme de fonctions linéaires. La solution optimale x doit maximiser ou minimiser la valeur de la fonction objectif $x^T c$ tout en respectant les contraintes $Ax \leq b$. On remarque que les contraintes sont des demi-espaces. Ensemble, elles délimitent une enveloppe convexe, un polytope dans lequel les solution sont acceptés. Il se peut que l'intersection des demi-espaces donne un polytope vide. On dit alors que le problème n'a pas de solution. s'il n'est pas vide, les solutions qui sont dans le polytope sont appelés solutions réalisables. Comme la fonction objectif est une fonction linéaire, elle traverse le polytope et donc la meilleure solution du problème est sur le bord du polytope dans la direction de la fonction objectif. Elle est appelée solution optimale.

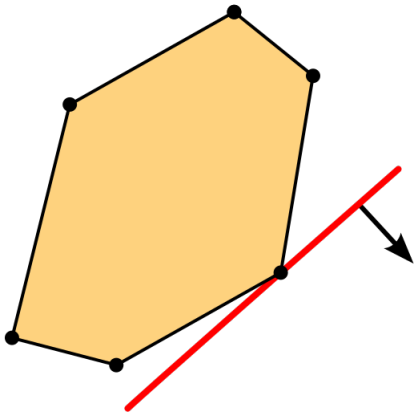

FIGURE 1 – Exemple de polytope, de fonction objectif et de solution optimale

Pour les notation, ^{T} est la transposé d'un vecteur, x est une solution, c le vecteur des coûts de la fonction objectif, A est la matrice des coûts des contraintes et b le vecteur des valeurs que ces contraintes ne doivent pas dépasser. On peut alors écrire le programme linéaire comme suit :

$$min\{c^T x : Ax \geq b, x \geq 0\}$$

Si on a n le nombre de variables et m le nombre de contraintes, on a :

$$x \in \mathbb{R}^n; c \in \mathbb{R}^n; A \in \mathbb{R}^{n \times m}; b \in \mathbb{R}^m$$

2.2 Algorithme du simplexe

L'algorithme du simplexe permet de résoudre des programmes linéaires de manière très efficace en pratique malgré sa complexité exponentielle. Il se déroule en deux phases distinctes :

— La **phase une** consiste à vérifier que les contraintes ne sont pas impossibles à satisfaire et à trouver un point dans la région admissibles du problème. Elle est identique pour le simplexe multi-objectif.

— Si le problème n'est pas impossible, on entame la **phase deux**. L'idée est de se déplacer sur les arrêtes du polytope jusqu'à trouver la solution optimale. Pour cela, on ajoute au problème une variable d'écart pour chaque contrainte. Cette variable modélise l'écart entre la solution courante x et la contrainte. Quand cette variable est égale à zéro, cela signifie que la solution est au bord de la contrainte. On dit que la contrainte est saturée et que les variables d'écart sont en base. L'algorithme consiste donc à chercher la combinaison de variables en bases qui donnent la meilleure solution. Pour ceci, il fait des pivot de gauss pour se déplacer de base en base.

2.2.1 pivot de gauss

Un pivot de gauss, dans le cas du simplexe consiste à faire entrer une variables X_j en base et à choisir une variable sortante X_s. ces choix se font toujours en fonction le l'objectif du problème. On ajoute la variable qui augmentera le plus notre solution et on retire la variable d'écart qui saturera le plus tard possible pour faire le plus grand déplacement possible vers la solution optimale. Par exemple si on prend le tableau simplexe suivant :

$$\left[\begin{array}{c|ccccc|c} & x_1 & x_2 & x_3 & x_4 & x_5 & b \\ \hline z & 3 & 2 & 0 & 0 & 0 & 0 \\ \hline x_3 & 2 & 1 & 1 & 0 & 0 & 6 \\ x_4 & 1 & 1 & 0 & 1 & 0 & 4 \\ x_5 & 0 & 1 & 0 & 0 & 1 & 3 \end{array}\right] \tag{1}$$

On souhaite faire entrer La variable X_1 dans la base car elle a la plus grande valeur dans la foncions objectif (z).

On cherche donc la variable en base qui permettra d'augmenter le plus possible la valeur de la fonction objectif avant de saturer. On calcul la division du vecteur b par la colonne de X_1 pour trouver le meilleur ratio. Dans ce cas on a $6/2$ et $4/1$ donc la variable sortante est X_3

On effectue le pivot de gauss et on obtient le tableau suivant :

$$\left[\begin{array}{c|ccccc|c} & x_1 & x_2 & x_3 & x_4 & x_5 & b \\ \hline z & 0 & 1/2 & -3/2 & 0 & 0 & -9 \\ \hline x_1 & 1 & 1/2 & 1/2 & 0 & 0 & 3 \\ x_4 & 0 & 1/2 & -1/2 & 1 & 0 & 1 \\ x_5 & 0 & 1 & 0 & 0 & 1 & 3 \end{array}\right] \tag{2}$$

On répète l'algorithme jusqu'à ce qu'il n'y ai plus de valeurs positives dans la fonction objectif z.

2.3 Forme révisée du simplexe

L'algorithme du simplexe sous sa forme classique demande de calculer une suite de tableaux simplexe. En théorie cela ne pose aucun problèmes mais dans la pratique ces calculs en génèrent des imprécisions numériques qui se propagent de tableau en tableau. Et si le problème est trop long à résoudre, on peut finir avec des tableaux complètement erronés.

C'est pourquoi nous nous intéresseront à la forme révisée du simplexe pour calculer ces tableaux

La forme révisée du simplexe permet de calculer un tableau simplexe dans n'importe quelle base à partir des données d'origine du problème. Par exemple si on reprend la formulation du problème :

$$min\{c^T x : Ax \geq b, x \geq 0\}$$

Et qu'on note B la base dans laquelle on veut trouver le prochain pivot.

On recalcule les coûts réduits, cet à dire les valeurs de la fonction objectif avec la formule :

$$c^T - c_B^T A_B^{-1} A$$

La colonne de la variable X_j est donnée par la formule :

$$A_B^{-1} A$$

Et le vecteur b par la formule

$$A_B^{-1} b$$

Ces informations suffisent pour détecter la variable entrante et trouver la variable sortante du prochain pivot de gauss.

On remplace donc une suite de tableaux simplexe par un calcul de la matrice inverse A_B^{-1} pour avoir directement les valeurs qui nous intéressent.

2.4 Décomposition LU

La forme révisé remplace la suite de tableau par le calcul d'une matrice inverse A_B^{-1} mais si nos problèmes se modélisent souvent par des matrice creuses les matrices inverses de ces dernières ne le sont généralement pas.

La décomposition LU permet de simplifier grandement les calculs matriciels de la forme révisé du simplexe. En effet plutôt que de calculer une matrice inverse, on calcule la décomposition LU de la matrice A_B. Cette méthode décompose la matrice A_B en deux matrices L et U qui sont respectivement des matrices triangulaires infèrieure et triangulaires supèrieure.

Une fois la décomposition LU calculée, on en déduit les coûts réduits, la colonne de la variable X_j, et le vecteur b avec des systèmes d'équation. Ces systèmes d'équations sont très simples grâce à la nature triangulaire des matrices L et U.

Ces deux méthodes permettent d'avoir une implémentation d'algorithme moins encline à générer des imprécisions numériques. Elles sont très utilisées dans les simplexe des solveurs.

2.5 Optimisation multi-objectif

Il arrive bien souvent que l'on souhaite optimiser un problème en prenant en compte un ou plusieurs autres objectifs. Par exemple on pourrais vouloir acheter un bonbon par rapport à sa taille et son prix. On voudrais prendre en compte les deux en même temps et cela malgré le fait que ces objectifs sont contradictoires. Pour résoudre ce problème, on maximiserai la taille du bonbon tout en minimisant son prix.

On indice des vecteurs de ces objectifs $c_1 ... c_n$ et on les met tous dans une matrice C. On peut alors noter le MOLP comme suit :

$$min\{Cx : Ax \geq b, x \geq 0\}$$

2.5.1 Optimalité et Dominance

À REFAIRE On se retrouve avec des solution que l'on ne peut pas comparer [1]. Comment décider entre un petit bonbon pas cher et un gros bonbon très cher. Il existe plusieurs définition de dominances mais ici on ne s'intéressera qu'aux solution dont il n'existe pas d'autre solution admissibles qui leur soit aussi bonne en les dépassant strictement sur au moins objectif. Ces solutions sont appelés solutions non-dominées ou solutions paréto-optimales.

On peut alors représenter ces bonbon non-dominés sur un graphique dans l'espace des objectifs :

1. Un peu comme dans un treillis

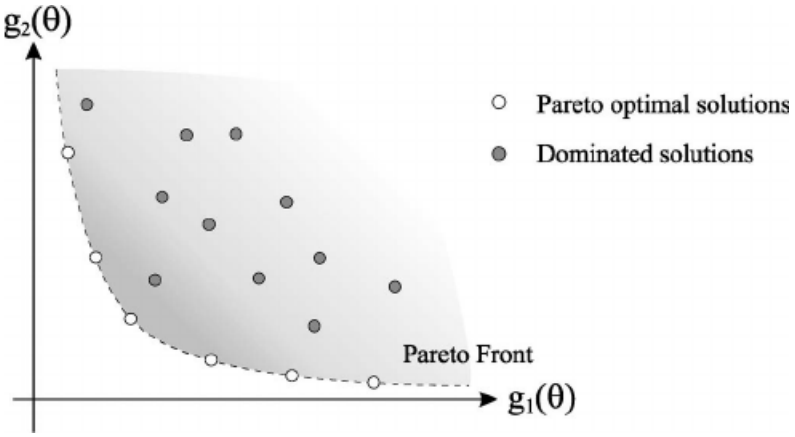

FIGURE 2 – Exemple de solutions paréto-optimales dans un espace à deux objectifs QUELQU'UN N'AURAI PAS UNE MEILLEURE IMAGE SVP ?

2.5.2 Somme pondéré

Pour se ramener à un problème plus simple, on peut faire une somme des fonction objectif pour se retrouve avec une seule fonction objectif. Et ainsi on peut résoudre le programme linéaire avec un simplexe de manière classique. Cependant, pour pouvoir trouver toutes les solution non dominées, il faut faire cette somme mais en favorisant tour à tour les d'objectifs. Et si on résout toutes les sommes pondérées objectifs avec toutes les combinaisons de poids possibles, on aura alors trouvé toutes les solution paréto-optimales. Seulement les combinaisons de poids sont très nombreuses.

2.5.3 Cas bi-objectif

À REFAIRE Dans le cas d'un problème bi-objectif, l'espace des objectifs est un plan et les solutions non dominées forment une ligne que l'on peut explorer d'un bout à l'autre.

2.5.4 Cas multi-objectif

À REFAIRE Mais dans le cas multi-objectif, on à trois ou plus objectifs. Donc les solutions non-dominées sont sur un polytope de dimension supérieure à trois. N'ayant plus la possibilité de les explorer d'un bout à l'autre, on les énumères toutes en faisant attention de ne pas recalculer deux fois la même solution.

2.6 Algorithme de Benson

À ENLEVER ?

3 Algorithme du simplexe multi-objectif

TROP CONCI

3.1 idée générale

Le simplexe multicritère est un algorithme qui énumère toutes les bases non-dominées d'un MOLP (Multi Objective Linear Problem). Comme on est dans le cas d'un problème linéaire, on sais que toutes les bases non-dominées sont au bord d'un polytope et qu'on peut passer d'une base à une autre par un pivot dit "efficace".
Un pivot est efficace si le problème auxiliaire construit à partir des fonction objectifs de la base que l'on cherche à atteindre est non borné. L'algorithme consiste donc à explorer tous l'ensemble des solutions à partir d'une

seule base (**phase 3**).
La première base non-dominée est calculée par la **phase 2** Pour cela, on fait une résolution de la somme pondérée du problème. Les poids de cette somme sont calculés lors de la résolution de la variante duale de l'algorithme de Benson.
Avant de commencer toute résolution, il faut vérifier que le problème n'est pas impossible, pour cela, on fait une **phase 1** de simplexe classique.

Construction du problème auxiliaire :
Pour savoir si il serait intéressant selon les objectifs de faire entrer une variable Xj dans la base, on construit un problème auxiliaire avec les fonction objectifs de la base courante. On met les colonnes des variables hors base dans une matrice R. On utilise cette matrice dans le PL : la matrice des contraintes est la concaténation de cette matrice R et de moins la colonne j de R,(la colonne de la variable que l'on pense faire entrer dans la base). On construit la fonction objectif de ce problème avec la somme de ses contraintes. Si la résolution de ce problème est bornée, alors introduire la variables j dans la base mènera vers une base non-dominée.

3.2 Algorithme du simplexe multi-objectif

Algorithme 1 : Multicriteria simplexe Algorithm From Matthias Ehrgott[1]

```
1  Phase 1 : phase 1 simplexe classique
2  La phase 1 du simplexe nous donne une solution admissible X_0
3  if il n'y a pas de solution admissible then
4  |  le problème est impossible
5  else
6  |  Phase 2 : Dual de Benson et première base
7  |  On construit un problème dual de Benson du MOLP
8  |  if le dual est impossible then
9  |  |  le MOLP est non-borné.
10 |  else
11 |  |  On utilise les poids trouvés par le dual de Benson et on résout la somme pondérée
12 |  |  On obtiens une première base non-dominée et on la met dans la liste L
13 |  |  Phase 3 : Énumération des bases non-dominées
14 |  |  On se place sur la première base trouvée à la phase 2
15 |  |  while il reste des bases à explorer dans la liste do
16 |  |  |  On utilise la forme révisée du simplexe pour calculer les fonctions objectifs dans la base courante
17 |  |  |  for Chaque variable X_j qui n'est pas en base do
18 |  |  |  |  On résout problème auxiliaire avec le simplexe
19 |  |  |  |  if le problème auxiliaire est non borné then
20 |  |  |  |  |  La variable est efficace,
21 |  |  |  |  |  On calcul alors la variable sortante grâce à la forme révisée et la décomposition LU
22 |  |  |  |  |  if une variable peut sortir then
23 |  |  |  |  |  |  On construit une nouvelle base en interchangeant la variable entrante et la variable
   |  |  |  |  |  |   sortante
24 |  |  |  |  |  |  if La nouvelle base n'est pas déjà dans la liste L then
25 |  |  |  |  |  |  |  On l'ajoute au bout de la liste
26 |  |  |  |  |  |  end
27 |  |  |  |  |  end
28 |  |  |  |  end
29 |  |  |  |  On se place sur la base suivante de la liste
30 |  |  |  end
31 |  |  |  retourner Liste L
32 |  |  end
33 |  end
34 end
```

4 Implémentation de l'algorithme

L'implémentation de l'algorithme précédemment décrit a été réalisé en Julia, version 1.02. Nous allons ici décrire les choix d'implémentations qui on été faits leurs justifications et leurs conséquences.

4.1 Structure de donnée

PAS CLAIR ? Pour coder les données du problème, on utilise simplement les matrices et les vecteurs de Julia. Ces objets sont indicés à partir de 1 en Julia.

Pour résoudre la phase un du simplexe, le dual de benson et la première somme pondérée, on utilise la modélisation JuMP et un solveur, par exemple GLPK.

Si les choix d'implémentations ci-dessus paraissent évident, il n'est pas de même pour le choix concernant l'encodage des Bases et des listes de Bases. En effet, L'algorithme tel que décrit dans [1] propose deux listes L1 et L2. Les bases que l'on a trouvées mais pas encore explorées sont dans la liste L2 et quand on a détecté toutes les variables efficaces de cette base, on la retire de L2 pour la mettre dans L1. Lors de l'algorithme, quand on trouve une nouvelle base, avant de l'ajouter à la liste L2, on doit vérifier qu'elle n'appartient pas déjà à aucune des listes en prenant en compte que la base peut être dans un autre ordre. On a donc ici une représentation ensembliste des bases. Il existe en julia une telle classe d'objet, les set qui on été testés dans les premières implémentations. On avais alors des bases qui étaient des ensembles de variables et des liste qui étaient des ensembles de bases.

Cependant, l'implémentation s'est dirigée vers une structure plus simple les tableaux. Cette implémentation a deux avantages par rapport aux set, on connais sa complexité et elle permet de ne manipuler qu'une seule liste. En effet comme un tableau est indicé, cela nous permet de passer simplement sur la base suivante du tableau et d'ajouter les nouvelles bases au bout de celui-ci. Pour savoir si une base appartient à la liste, on est obligé de regarder une à une les variables de chaque bases de la liste. On a donc une complexité de $\mathcal{O}(\#(B)^2 \times \#(L))$. Une deuxième idée pour gagner en complexité serai de trier les bases. De cette manière, on peut savoir plus simplement si elles appartiennent à la liste de bases. Cette structure a théoriquement une complexité de $\mathcal{O}(\#(B \times log(B)) + \#(B) \times \#(L))$ Même si dans la pratiques les deux approches sont équivalentes en terme de temps d'exécution car la principale complexité de l'algorithme est dans la résolution des problèmes auxiliaires et des systèmes d'équations de la décomposition LU.

4.2 Comparaison des implémentations et observations

Lors de l'implémentation, une erreur dans la décomposition LU a forcé la réécriture de la phase trois avec la matrice inverse de la forme révisé du simplexe pour vérifier les calculs. Comme dit précédemment, la décomposition LU permet de simplifier les calculs et d'accumuler moins d'imprécisions numériques. Cette observation est flagrante lors des tests même si sur les grandes instances, la décomposition LU peut générer quelques erreurs d'imprécisions numériques. Pour les tests, une matrice arrondissant la matrice du problème auxiliaire à été écrite. Elle est lourde en complexité temporelle mais elle permet d'éviter quelques erreurs.

4.3 Instances

4.3.1 Générateur d'instances

La génération de problèmes aléatoires est dirigée par plusieurs paramètres qui peuvent êtres modifiés.

Notation :
— $[a : b]$ signifie toutes les valeurs entières entre a et b inclus.
Pour les paramètres :
— $[3 : 9]$ objectifs,
— $[2 : 20]$ variables,
— $[1 : 30]$ contraintes,
— $[-9 : 9]$ pour les valeurs dans A et C avec une forte probabilité pour la valeur 0
— $[1 : 20]$ pour les valeurs de b

— une chance sur 6 d'avoir une contrainte d'égalité et le reste des chances réparties entre les inégalités

4.3.2 Observations

On remarque que 70% des instances générés sont impossibles, 25% des restantes ne passent pas la phase 2 et quand le problème n'est pas dégénéré, il est rare de ne pas avoir plusieurs rayons infini lors de l'exploration des bases. On en déduit qu'il est très difficile de générer des instances possibles et intéressantes. Quelques exemples générés intéressants sont dans le fichier resultats.jl

4.3.3 Parser d'instances

Pour tester d'autres instances, un parser pour les instances .mop a été fais.
Exemple d'instance qui peut être parsé : "data/molp_10_779_10174_entropy.mop"

5 Pistes d'améliorations

5.1 Récupération de la première base dans JuMP ou GLPK

Pour l'instant, l'algorithme fais une hypothèse de non-dégénérescence pour déterminer la première base à la fin de la phase 2. Cette hypothèse est très dérangeante car la majorité des instances existantes sont dégénérées.

5.2 Interface avec voptgeneric

L'interface avec vopt reste à faire mais la mise a jour en cour de JuMP vers sa version 19 rend ceci compliqué.

5.3 Optimisation

On pourrait peut-être gagner un peu temps en reprogrammant la résolution des systèmes triangulaires (sup et inf) si elle n'est pas traitée spécifiquement dans Julia. Mais sinon, l'implémentation tableau semble être la plus optimisée pour la phase 3, à voir si on peut faire des choses combinées avec vOpt pour la phase une et deux.
Peut-être que d'autres structures qu'un tableau pourraient être utilisées mais elles ne paraissent pas très convaincantes lors des premières implémentations.

5.4 Interprétation graphique

L'algo nous donne les bases et les solution donc on peu recalculer les coûts mais il faudrait un petit traitement pour reconnaître les bases qui forment des faces non dominées pour apporter une réponse graphique.

6 Instances

Les instances utilisés dans ce projet sont tirées des livres suivants :[1], [2], [3]

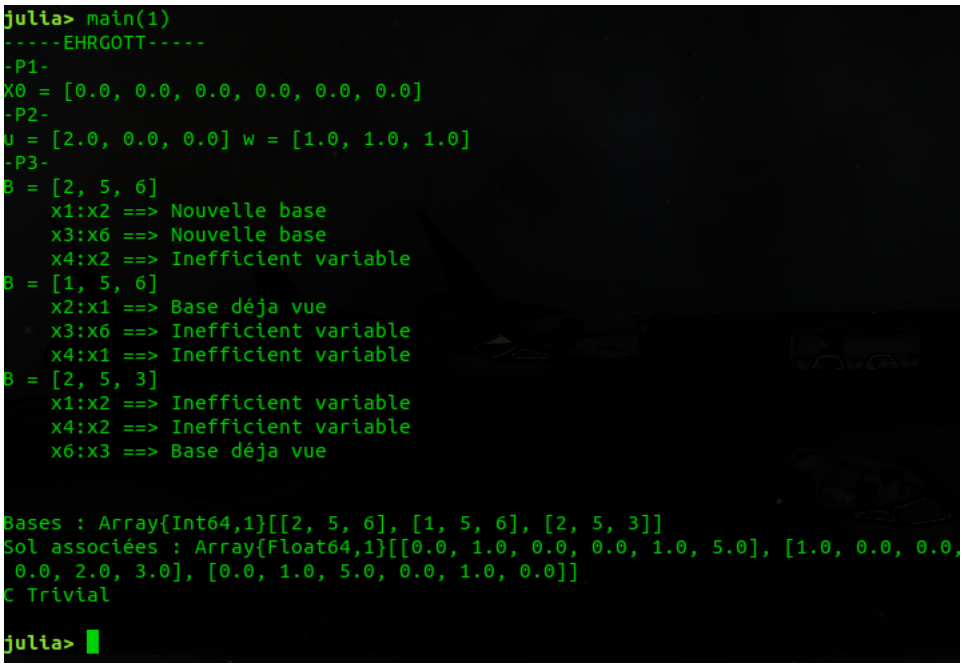

FIGURE 3 – Exemple d'exécution sur l'instance du livre de Matthias Ehrgott[1]

7 Multicriteria simplexe Algorithm : Pseudocode formel

Cette section propose une écriture plus formelle de l'algorithme. Comme Julia est un langage de programmation de haut niveau, l'implémentation est très proche de cette écriture.

Nomenclature

- MOLP : $min\{Cx : Ax + Iz = b, x, z \geq 0\}$
- C : matrice des fonctions objectifs
- A : matrice des contraintes
- b : vecteur des membres de droite des contraintes
- x : variables
- z : variables d'écarts

Phase 1

- e : vecteur null
- x^0 : solution de la phase 1

Phase 2

- Dual du MOLP : $min\{u'b + \lambda' Cx^0 : u'A + \lambda' C \geq 0, \lambda \geq e\}$
- u : variables duales associées au vecteur b
- λ : variables duales associées au fonction objectifs
- e : vecteur de 1
- λ_1 : poids trouvés après la résolution du dual. Ils sonts utilisés dans la première somme pondérée
- $B1$: première base
- $x1$: première solution
- LB : liste des bases
- Lx : liste des solutions

Phase 3

- cb : indice de la base courante
- B base courante
- ω : ensemble des variables
- N : variables hors bases
- A_B : Matrice A avec seulement les colonnes des variables en bases
- L,U : décomposition en une matrice triangulaire inférieure et une matrice triangulaire supérieure de la matrice A_B

— k : indice de la fonction objectif
— $C_B[k]$: Coefficients de la fonction objectif k dans la base B
— A_N : Matrice A avec seulement les colonnes des variables hors bases
— $C_N[k]$: Coefficients de la fonction objectif k des variables hors bases
— R : matrice des coefficients des fonctions objectifs obtenus par la forme révisée et la décomposition LU
— PL auxiliaire : $min\{e'(y+\delta) : Ry - r^j\delta + Iv = 0, y, \delta, v \leq 0\}$ PL qui donne l'efficacité d'une variable
— r^j : colonne j de la matrice R
— e : vecteur de la somme des colonnes de la matrice $concat(R, -r^j)$
— x_b : solution associée à la base B
— x^s : variable sortante
— Bj : nouvelle base
— x_j : nouvelle solution associée à la base Bj

Algorithme

Algorithme 2 : Multicriteria simplexe Algorithm From Matthias Ehrgott[1]

Phase 1 : phase 1 simplexe classique /* mono-objective simplexe Phase 1 */
Solve $min\{e'z : Ax + Iz = b, x, z \geq 0\}$
if $z == 0$ **then**
  x^0 is a basic solution of the MOLP
else
  $X = \oslash$ /* the MOLP is infeasible */
end
Phase 2 : Benson Dual and first basis
Solve $min\{u'b + \lambda' Cx^0 : u'A + \lambda' C \geq 0, \lambda \geq e\}$ /* dual simplexe gives us the first wheights:λ_1 */
if *:Infeasable* **then**
  $X_E = \oslash$ /* The MOLP has no efficients solutions */
else
  Solve $min\{\lambda_1' Cx : Ax = b, x \geq 0\}$ /* the first weighted sum gives us the first non-dominated Basis */
  Find basis $B1$ and solution $x1$
  $LB := B1$ and $Lx := x1$ /* we put them in a list of basis to explore, and a list of solution */
end
Phase 3 : Basis enumaration
cb = 1 /* index of the curent basis */
while $cb \leq length(LB)$ **do**
  $B = LB[cb]$ and $x_b = Lx[cb]$ /* while we have basis to explore in the list */
  $N = \Omega - B$ /* N:non-basis variables (Ω:all variables) */
  L,U = LU décomposition of A_B /* LU decomposition of matrix A on basis B */
  for $k \leftarrow 1$ **to** *number of objectives* **do**
    Solve $t'U = C_B[k]$ /* calcul des objectifs par la forme révisée du simplexe et la décomp LU */
    Solve $y'L = t'$
    $R[k] = (C_N[k])' - y'A_N$ /* R is the matrix of the non-basic vars of the objectives fonctions */
  end
  for $j \leftarrow 0$ **to** *length(N)* **do**
    /* for all variable in N, we test if there are efficient with the following PL */
    Solve $min\{e'(y + \delta) : Ry - r^j\delta + Iv = 0, y, \delta, v \leq 0\}$
    if *:Unbounded* **then**
      "Inefficient variable"
    else
      Solve $Ly = x_b$ /* revised simplexe + LU decomp */
      Solve $Ud = y$
      s = find positive min $\frac{x_b}{d}$ /* x^s:variable out of the pivot */
      if *s doesn't exist* **then**
        "Impossible Pivot or infinite ray"
      else
        "Pivot Xj :Xs"
        Bj new basis and x_j new solution
        if Bj *is already in* L **then**
          "already known Basis" /* with 3 or more objectives, we can return on old basis */
        else
          $LB := Bj$ and $Lx := x_j$ /* we add the new basis and the new solution to the lists */
        end
      end
    end
  end
  cb = cb + 1
end
retourner *Liste L*

8 Conclusion

Nous avons tout d'abord cherché à implémenter l'algorithme tel que décrit dans [1]. Nous avons remarqué que celui-ci pouvait être légèrement optimisé en modifiant les structures de données qu'il utilise.

Puis nous avons amélioré l'algorithme avec la forme révisée du simplexe et la décomposition LU. Et nous avons obtenu une réduction conséquente des imprécisions numériques.

Nous avons observé qu'un générateur d'instance peine à créer des problèmes intéressants. Il nous est aussi difficile de parser des instance à cause de l'hypothèse de non-dégénérescence de la phase 2. L'idéal aurai été de la récupérer directement de la modélisation de JuMP mais cette opération restera très incertaine tant que sa version ne sera pas stable.

C'est pour cette même raison que l'intégration de l'algorithme à vOptSolver n'a pas été faite.

L'interface graphique pourrait être intéressante mais elle ne parais pas vitale pour cet algorithme car il ne sera pas utilisé en tant que tel. En effet, il existe peu de problèmes molti-objectif linéaires et continus. L'utilisation de cet algorithme parais plus utile dans la relaxation d'un problème multi-objectif combinatoire. De ce fait, l'algorithme se doit d'être le plus optimisé possible. Et pour ceci, les deux pistes que nous avons sont la reprogrammations de la décomposition LU d'une matrice et de la résolution des systèmes d'équations triangulaires. Car on ne sais pas si ces méthodes implémentées en Julia correspondent exactement à notre problème.

Références

[1] Matthias Ehrgott. *Multicriteria optimization*, volume 491. Springer Science & Business Media, 2005.

[2] Ralph E Steuer. Multiple criteria optimization. *Theory, Computation and Applications*, 1986.

[3] Milan Zeleny and James L Cochrane. *Multiple criteria decision making*. University of South Carolina Press, 1973.