Couvert Nicolas
Torres Andy
Groupe 685

Projet de Recherche Opérationnelle

Descriptif et justification des structures de données choisies

Dans ce projet, nous avons principalement utilisé les structures de données «Matrix» pour gérer les tableaux de données des instances plats et reliefs données ainsi que des «Vector» afin de manipuler et stocker les cycles dans le cadre de la méthode de résolution Dantzig-Fulkerson-Johnson (DFJ) notamment. Cela nous permet d'avoir un accès en temps constant aux cases du tableau et de pouvoir rajouter aisément un élément à la fin de celui-ci en utilisant la fonction *push!*. En effet, nous utilisons la structure «Vector{Tuple{Int64,Int64}}» pour stocker les cycles sous forme de paires d'éléments. Par ailleurs, la structure «Vector{Vector{Int64}}» est aussi très utile dans les fonctions *Exist* et *PlusPetit* pour savoir si l'élément existe et trouver le plus petit élément, respectivement, dans le «Vector{Vector{Int64}}», avec une complexité quadratique dans le pire cas.

Preuve de l'affirmation

L'affirmation de la partie 3.2 est vraie car si on obtient une solution composée d'un seul cycle alors nous n'avons pas de contraintes supplémentaires à ajouter puisqu'il n'y a plus de cycles à casser. Ainsi, la région admissible trouvée ne pouvant pas être davantage réduite par de nouvelles contraintes, nous obtenons donc la solution optimale du problème.

Pseudo-Codes des fonctions demandées

Voici le pseudo-code des fonctions *Exist*, *Cycles* et *PlusPetit* nous permettant d'établir la solution en permutation puis en produit de cycles disjoints :

```
Fonction Exist(val::Entier sur 64 bits, tableau::Vecteur de Vecteur d'entiers sur 64 bits)
   n::Entier sur 64 bits <- taille(tableau)
   m::Entier sur 64 bits <- 0

   i::Entier sur 64 bits <- 1

   Tant que i ≤ n    #boucle servant au parcours du tableau sur une dimension
      m <- taille(tableau[i])
      j::Entier sur 64 bits<- 1
      Tant que j ≤ m    #parcours sur la deuxieme, on parcourt tout le tableau
         Si (tableau[i][j] = val)
            retourner vrai     #si on trouve la valeur choisie, on renvoie true et on stoppe la fonction
         Fin Si
         j <- j + 1
      Fin Tant que
      i <- i + 1
   Fin Tant que
   retourner faux        #si on n'a pas stoppé, on n'a pas trouvé la valeur, donc on retourne faux
Fin
```

Fonction prenant en entrée un vecteur de tuples d'entier, et renvoyant un vecteur de vecteur, contenant les cycles induit par le vecteur en entrée

```
Fonction Cycles(X::Vecteur de paires d’entiers sur 64 bits)
   n::Entier sur 64 bits <- taille(X)
   i::Entier sur 64 bits <- 0
   courrant::paire d’entiers sur 64 bits <- (0, 0)
   tab::Vecteur d’entiers sur 64 bits <- Vector{Int64}(undef, n)
  #tab sera un vecteur ayant à chaque positon i, la valeur j de l'endroit lié, exemple avec un tuple (5,4), on aura en position 5, la valeur 4


   Tant que i < n     #boucle servant à remplir tab
      i <- i + 1
      courrant <- X[i]
      tab[courrant[1]] <- courrant[2]
   Fin Tant que

   tableau::Vecteur de vecteur d’entiers sur 64 bits <- []      #tableau sera le vecteur de vecteur contenant les cycles
   nbCycles::Entier sur 64 bits <- 0
   i <- 1


   #on parcourt tab, on prend les indices et les valeurs contenues et on les ajoute dans un vecteur de "tableau"
  #quand on tombe sur un élément déjà présent dans tableau, on a fini un cycle et on recommence, en ajoutant dans un nouveau vecteur

   Tant que (i ≤ taille(tab) - 1)
      nbCycles <- nbCycles + 1
      push!(tableau, [])

      Tant que (Exist(tab[i], tableau) et i < taille(tab))
         i <- i + 1
      Fin tant que

      valeur::Entier sur 64 bis <- tab[i]
      Si !(Exist(valeur, tableau))
         push!(tableau[nbCycles], valeur)
      Fin Si
      Tant que !(Exist(tab[valeur], tableau))
         valeur <- tab[valeur]
         Si !(Exist(valeur, tableau))       #ce Si sert à ne pas avoir de doublon, sinon on aurait deux fois la 1ere valeur qu'on trouve aussi à la fin
            push!(tableau[nbCycles], valeur)
         Fin Si
      Fin Tant que
   Fin Tant que
   pop!(tableau)      #sans cette ligne, on a aussi un vecteur vide en trop
   retourner tableau
Fin
```

Cette fonction prend en entrée un vecteur de vecteur, et retourne le vecteur avec la taille la plus petite

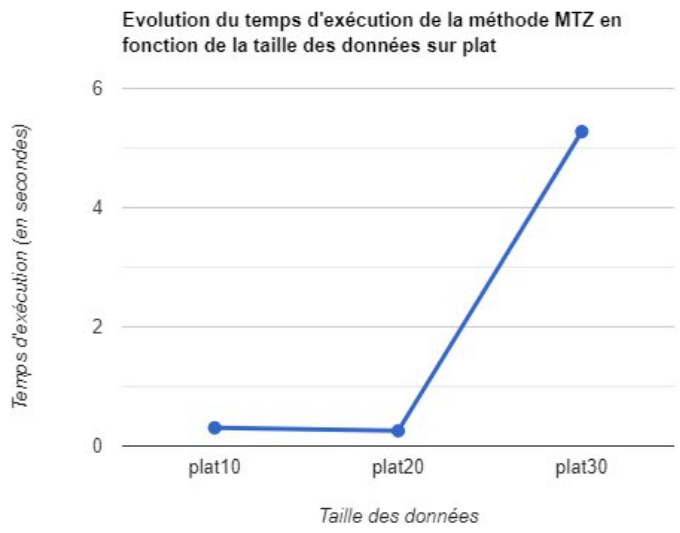
```
Fonction PlusPetit(X: Vecteur de vecteur d'entiers sur 64 bits)
   n:: Vecteur d'entiers sur 64 bits <- []
   min:: Entier sur 64 bits <- {valeur maximale possible sur les entiers}

   Pour x:: Vecteur d'entiers sur 64 bits dans X
      Si taille(x) ≤ min
         min <- taille(x)
         n <- x
      Fin Si
   Fin Pour

   retourner n
Fin
```

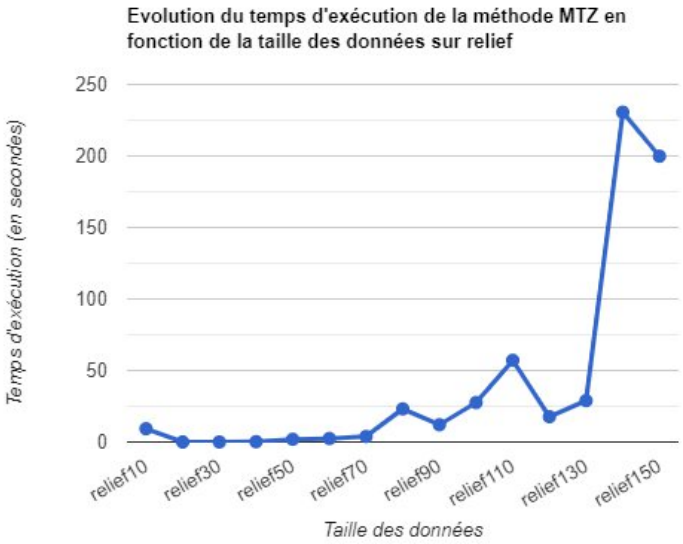
Analyse expérimentale des méthodes de résolution exactes

Méthode MTZ sur les fichiers de type «plats» :

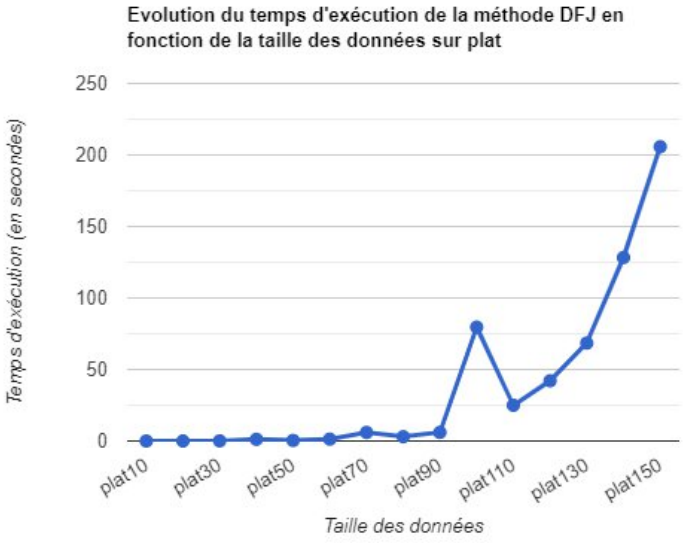


Les fichiers de la catégorie plat allant de plat40 à plat150 ont un temps d'exécution beaucoup trop long, nous avons donc décidé de ne montrer que les premiers résultats qui montrent néanmoins une forte croissance du temps de calcul à partir de plat30. On a un temps d'exécution de plat40 à 150 supérieur à 5 minutes.

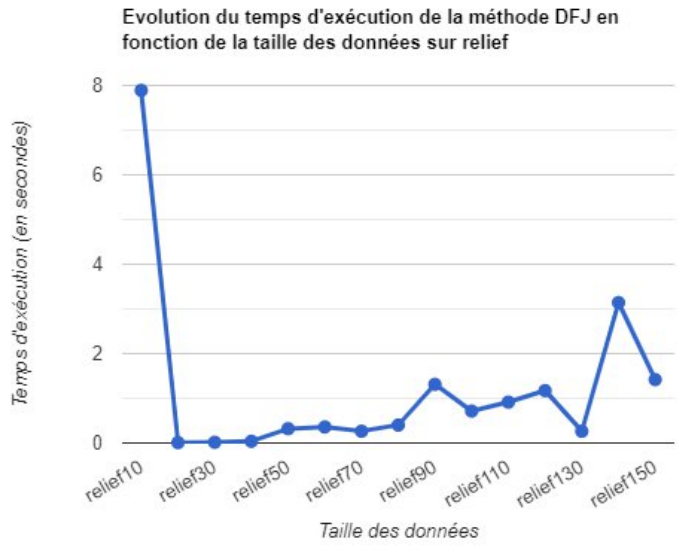
Méthode MTZ sur les fichiers de type «relief» :



Méthode DFJ sur les fichiers de type «plat» :

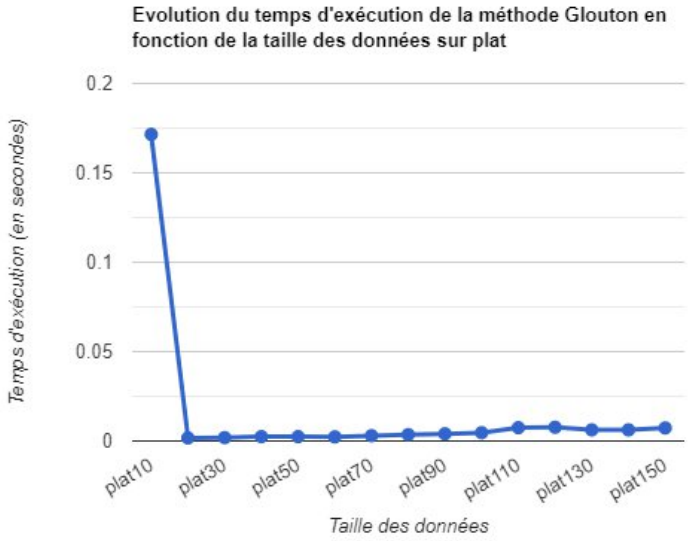


Méthode DFJ sur les fichiers de type «relief» :

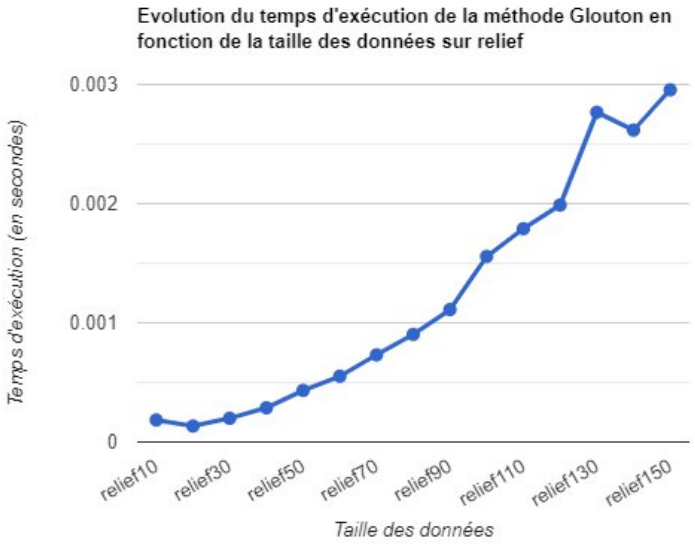


Lorsque l'on analyse les différents graphes des deux méthodes de résolution, nous observons que le temps de calcul pour la catégorie relief est toujours plus rapide. Cela s'explique par la symétrie du distancier des instances de plats. En effet, lorsque l'algorithme trouve une solution, le chemin inverse fonctionne également puisqu'il y a la même distance. Il y aura donc plus de solutions à traiter à chaque fois, ce qui n'est pas le cas pour les instances de relief dans lesquelles le distancier est asymétrique.

Méthode Glouton sur les fichiers de type «plat» :



Méthode Glouton sur les fichiers de type «relief» :



Nous remarquons que la méthode Glouton est bien plus rapide que les autres méthodes à la fois sur les instances de plat et relief (allant jusqu'à plusieurs puissances de 10) mais fournit des résultats approximatifs (la distance obtenue est en moyenne 2 à 3 fois plus élevée qu'avec les autres méthodes). Elle est donc utile dans les situations où l'on souhaite obtenir un résultat rapidement au détriment de la précision.