Nantes Université

PROJET

RECHERCHÉ OPÉRATIONNELLE

2022-2023

LICENCE 3 - INFORMATIQUE

ATALLA SALIM - 684J
ABDULLA RAHAF - 685K

Résumé du problème

Nous abordons un problème complexe de planification de mission pour un voyage sur Mars. Ce voyage nécessite la collecte de ressources locales pour assurer la survie et le retour sur Terre. Les ressources sont limitées et des drones doivent être utilisés pour collecter ces ressources. Pour optimiser le temps et la consommation d'énergie, nous devons planifier les trajectoires des drones de manière efficace.

Dans cette contexte, le problème de planification de trajectoire des drones pour collecter des ressources locales sur Mars peut être considéré comme une variation du problème de "voyageur de commerce". Dans ce cas, les "villes" seraient les points d'intérêt où les drones doivent collecter des ressources, et le chemin à trouver serait la trajectoire optimale pour chaque drone afin de minimiser le temps total et la consommation d'énergie.

Dans ce rapport, nous explorons les solutions possibles pour planifier les trajectoires des drones et minimiser le temps total et la consommation d'énergie tout en explorant les points d'intérêt affectés à chaque drone.

Problème de "voyageur de commerce"

Le problème de "voyageur de commerce" est un problème bien connu dans le domaine de la recherche opérationnelle. Il s'agit d'un problème d'optimisation combinatoire qui cherche à trouver le chemin le plus court reliant un ensemble de villes tout en passant par chacune d'entre elles une seule fois. Le problème est NP-complet, ce qui signifie qu'il n'existe pas d'algorithme connu pour résoudre le problème en un temps raisonnable pour des instances de grande taille.

Modélisation du problème

Il existe plusieurs approches de modélisation pour résoudre le problème du voyageur de commerce. Cependant, nous allons nous concentrer sur deux modélisations qui utilisent la méthode de programmation linéaire.
Le nombre de lieux à visiter, comprenant la base et les points d'intérêt, est représenté par la variable n. Le temps nécessaire pour aller du lieu i au lieu j est noté c_{ij}, où $i, j \in \{1,...,n\}$. Les variables de décision sont définies comme suit :

$$x_{ij} = \begin{cases} 1 & Si\ le\ drone\ se\ rend\ directement\ du\ lieu\ i\ au\ lieu\ j \\ 0 & Sinon \end{cases} \qquad i, j \in \{1,...,n\}$$

Méthodes de résolution

Dans le cadre de ce projet, nous allons examiner deux méthodes de résolution exacte basées sur les deux modèles présentés dans la section précédente. Nous étudierons également une méthode de résolution approximative qui se base sur un algorithme glouton.

1] Résolution exacte en utilisant le modèle Miller-Tucker-Zemlin

Variables de décision:

$$x_{ij} = \begin{cases} 1 & Si\ le\ drone\ se\ rend\ directement\ du\ lieu\ i\ au\ lieu\ j \\ 0 & Sinon \end{cases} \qquad i, j \in \{1,...,n\}$$

t_j : Date à laquelle le lieu j est visité.

La fonction objectif:

$$Min\ z = \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}$$

$$s.c. \sum_{j\in\{1,...,n\}\setminus\{i\}} x_{ij} = 1 \quad \forall i \in \{1,...,n\}$$

$$\sum_{i\in\{1,...,n\}\setminus\{j\}} x_{ij} = 1 \quad \forall j \in \{1,...,n\}$$

$$t_i - t_j + n.x_{ij} \leq n - 1 \quad \forall i, j \in \{2,...,n\}$$

$$x_{ij} \in \{0,1\} \quad \forall i, j \in \{1,...,n\}$$

$$0 \leq t_j \leq n \quad \forall i, j \in \{2,...,n\}$$

La méthode MTZ est une méthode exacte de résolution pour le problème du "voyageur de commerce". Toutefois, étant donné que ce problème relève de l'optimisation combinatoire NP-difficile, il n'existe pas d'algorithme polynomial capable de résoudre efficacement ce problème pour des instances de grande taille.

COMMENT IMPLÉMENTER CE PROBLÈME ?

Commençons par définir les variables de décision. Nous avons tout d'abord la variable binaire x_{ij}, qui représente le déplacement direct du drone du lieu i au lieu j. Ensuite, nous avons besoin d'une matrice c_{ij} qui représente la distance entre les points d'intérêts. Cette matrice peut être symétrique ou asymétrique et a une taille nxn. Nous récupérons cette matrice à partir d'un fichier .dat en utilisant la fonction de parseTSP.
De plus, nous ajoutons une variable supplémentaire t_j qui représente la date à laquelle le lieu j est visité.
Enfin, la fonction objectif vise à minimiser la distance totale entre les points d'intérêts. Comme dans un problème classique de "voyageur de commerce" résolu par un PL régulier, la solution optimale est obtenue par la fonction objectif suivante : $z = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$

Nous nous intéressons maintenant aux contraintes. Nous avons deux contraintes essentielles pour ce type de problème, appelées "contraintes de regroupement". Ces contraintes garantissent que nous ne passons qu'une seule fois d'un lieu i à un autre lieu j sans retourner au lieu i. Ainsi, la matrice x_{ij} ne contient qu'un seul "1" par ligne et par colonne.
En ce qui concerne la troisième contrainte, elle implique une notion de temps. Elle exige que la date ti à laquelle le lieu i a été visité soit antérieure à la date t_j à laquelle le lieu j a été visité.

OBSERVATION:

Après avoir mené des observations sur notre problème, nous avons constaté que le solveur a réussi à résoudre des problèmes ayant des instances symétriques de taille n=40 en un temps dépassant les 94 secondes, ce qui correspond à ses limites. Cependant, le solveur a réussi à continuer à résoudre des problèmes avec des instances asymétriques de taille n=150, bien que cela ait nécessité beaucoup de temps et de mémoire. En effet, pour résoudre un problème avec des instances asymétriques de taille n=140, le solveur a mis 73 secondes, tandis que pour résoudre des problèmes de taille n=150, il a fallu 94 secondes. Il est donc clair que cette méthode de résolution n'est pas efficace, voire impossible, pour les problèmes de grandes tailles, car le temps et la consommation de mémoire augmentent très rapidement.

2] Résolution exacte en utilisant le modèle Dantzig-Fulkerson-Johnson

Suite au modèle de résolution précédent, il est apparu nécessaire d'adopter un modèle plus performant pour résoudre rapidement des problèmes plus complexes. C'est pourquoi nous allons maintenant présenter le modèle DFJ. Nous conserverons dans ce nouveau modèle la fonction objectif, les contraintes de regroupement (1 et 2), les variables de décision x_{ij} ainsi que la matrice de distances c_{ij} où $i, j \in \{1,...,n\}$.

Qu-est ce que la permutation et les cycles ?

Avant tout, définissons deux notions : la permutation et les cycles. Nous pouvons décrire la permutation comme la transformation d'une matrice de distances bidimensionnelle en un vecteur unidimensionnel, où chaque indice du vecteur correspond au numéro d'un lieu et sa valeur représente le lieu le plus proche. Les cycles, quant à eux, représentent les chemins possibles à partir d'un lieu donné, qui peuvent inclure des arrêts intermédiaires avant de revenir au point de départ. Il peut y avoir plusieurs cycles pour inclure tous les lieux.

L'algorithme consiste à casser le cycle le plus petit en ajoutant une contrainte au modèle à chaque résolution de PL jusqu'à obtenir un seul cycle. Lorsque le PL est résolu, une matrice binaire x_{ij} est obtenue, qui permet de construire un nouveau vecteur en utilisant la méthode de permutation. Pour cela, nous vérifions les cases égales à 1 de la matrice (puisqu'il n'y a qu'un seul 1 par ligne/colonne) : le numéro de colonne j correspond au lieu courant, et la ligne i où se trouve le 1 correspond au numéro du lieu suivant dans le cycle.

Structures de données utilisées & pseudo-codes:

La matrice de distances c_{ij} : C::Matrix{Int64}

La matrice contient les informations relatives au problème posé. Les indices de ligne correspondent à des numéros de lieux, tandis que les valeurs associées à chaque indice représentent les distances qui séparent ce lieu des autres lieux, de manière symétrique par rapport aux indices de colonnes.

La Permutation : x::Matrix{Float64} -> Vector{Int64}

La fonction prend en entrée la matrice x_{ij} obtenue lors de la résolution du PL, et génère un vecteur d'entiers dont les indices représentent les numéros des lieux et les valeurs indiquent les lieux suivants dans le cycle.

```
fonction permutation (x : Matrice de réels) : Vecteur d'entiers

Variables:    n : Entier, j : Entier, res : Vecteur d'entiers

Début:

   n <- taille(x, 1) # Taille de x sur une seule dimension
   res <- allocation(n) # allouer le vecteur par n cases

   Pour i de 1 à n faire
      j <- 1
      Tantque x[i, j] != 1 faire
        j <- j + 1
      Fin Tantque
      res[i] <- j
   Fin Pour

   Retourner res

Fin:
```

Les Cycles : Vector{Int64} -> Vector{Vector{Int64}}

Les cycles sont générés en utilisant le vecteur obtenu par la permutation. Ils se présentent sous la forme d'un vecteur de vecteurs d'entiers, où chaque vecteur correspond aux différents lieux traversés par les drones lors de chaque cycle.

```
fonction creer_cycles (V : Vecteur d'entiers) : Vecteur de Vecteurs d'entiers

Variables:    n : Entier, i : Entier, comp : Entier,
              ajoutee : Vecteur de booléens,
              cycles : Vecteur de Vecteurs d'entiers

Début:

   n <- longueur(V)
   cycles <- allocation() # Initialiser le vecteur (vide au départ)
   ajoutee <- allocation(n, faux) # Allouer le vecteur n cases,
                                  # et le remplir avec des faux
   Pour lieu de 1 à n faire
      Si !(ajoutee[lieu]) alors

         Ajouter(cycles, [lieu]) # Ajouter le premier lieu dans le cycle
         ajouter[lieu] <- vrai
         comp <- V[lieu]

         i <- 1
         Tantque (comp != lieu) et (i <= n) faire
            i <- i + 1
            # Insérer comp dans le dernier vecteur dans cycles
            Ajouter(cycles[longueur(cycles)], comp)
            ajouter[comp] <- vrai
            comp <- V[comp]
         Fin Tantque
      Fin Si
   Fin Pour

   Retourner cycles

Fin:
```

Le Cycle à casser : Vecteur{Vector{Int64}} -> Vector{Int64}

Après avoir obtenu les cycles, nous sélectionnons le plus court d'entre eux et le représentons sous forme d'un vecteur d'entiers contenant les lieux du cycle. Nous utiliserons ensuite ce vecteur pour formuler la contrainte qui sera ajoutée avant la prochaine résolution du PL.

```
fonction cycle_a_casser (cycles : Vecteur de Vecteurs d'entiers) : Vecteur d'entiers

Variables:    min_len : Entier, min_vec : Vecteur d'entiers

Début:

   min_len <- Infinité
   min_vec <- cycles[1]

   Pour cycle dans cycles faire # Pour chaque cycle
     Si (longueur(cycle) < min_len) alors
         min_len <- longueur(cycle)
         min_vec <- cycle
      Fin Si
   Fin Pour

   Retourner min_vec

Fin:
```

Les contraintes de cassage : aff::AffExpr

En partant d'un modèle, une matrice de variables de référence et un vecteur représentant les lieux des cycles à casser (obtenu lors de l'étape précédente), il est possible de générer une expression affine. Cette expression sera ensuite ajoutée à la contrainte pour casser le cycle.

```
fonction generer_aff (m : Model, x : Tableau de variables de référence,
                          vec : Vecteur d'entiers) : expression affine

Variables:    n : Entier, aff : expression affine

Début:

   n <- taille(x, 1) # Taille de x sur une seule dimension
   aff <- expression(m, 0.0) # Initialiser l'expression avec une valeur nulle

   Pour i de 1 à n faire
      Pour j de 1 à n faire
         Si (i != j) et (i dans vec) et (j dans vec)
            ajouter_a_expression(aff, 1.0, x[i, j])
         Fin Si
      Fin Pour
   Fin Pour

   Retourner aff

Fin:
```

La preuve de l'affirmation (dans la section 3.2):

Pour prouver l'affirmation, nous devons démontrer deux choses :

1. La solution obtenue est admissible, ce qui signifie qu'elle respecte toutes les contraintes du problème.
2. La solution obtenue est optimale, ce qui signifie qu'elle a la valeur la plus petite possible parmi toutes les solutions admissibles.

La solution est admissible :

Pour prouver que la solution est admissible, nous devons montrer qu'elle satisfait toutes les contraintes du problème de voyageur de commerce. Les contraintes (1') et (2') sont respectées par construction de la solution, car elles ont été utilisées dans le processus de résolution. La contrainte (3') est ajoutée uniquement si la solution n'est pas un cycle unique. Cela signifie que si la solution obtenue ne nécessitait pas l'utilisation de la contrainte (3), cela implique que la solution est composée d'un seul cycle et qu'elle respecte donc toutes les contraintes du problème. Ainsi, la solution obtenue est admissible.

La solution est optimale :

Pour prouver que la solution est optimale, nous devons montrer que sa valeur est la plus petite possible parmi toutes les solutions admissibles. Comme nous avons obtenu la solution en utilisant une méthode heuristique, nous ne pouvons pas être certains qu'elle est la solution optimale. Cependant, si nous avons appliqué la méthode jusqu'à ce qu'il ne reste qu'un seul cycle, cela signifie que toutes les contraintes du problème ont été satisfaites et que la solution obtenue est la plus petite possible parmi toutes les solutions admissibles qui respectent les contraintes (1') et (2'). Ainsi, la solution obtenue est optimale.

En conclusion, si nous résolvons un problème ne contenant qu'un sous-ensemble des contraintes (3') et que nous obtenons une solution composée d'un seul cycle, cette solution est admissible et optimale pour cette instance du problème.

Exemple :

Prenons l'exemple suivant avec 5 lieux (numérotées de 1 à 5) et la matrice des distances suivante :

	1	2	3	4	5
1	0	5	20	15	10
2	5	0	10	30	25
3	20	10	0	3	35
4	15	30	3	0	8
5	10	25	35	8	0

Commençons par appliquer les contraintes (1) et (2) pour obtenir une solution de base admissible :

La solution obtenue avec seulement les contraintes (1) et (2) est :

1 -> 2 2 -> 3 3 -> 4 4 -> 5 5 -> 1

Cycle -> (12345)

Avec une solution optimale z = 36

	1	2	3	4	5
1	0	0	0	0	1
2	1	0	0	0	0
3	0	1	0	0	0
4	0	0	1	0	0
5	0	0	0	1	0

Maintenant, supposons que nous ajoutions la contrainte :

$$x_{23} + x_{32} \leq 1$$

qui casse le sous-tour {2, 3}.

Si nous résolvons à nouveau le problème, nous obtenons la solution suivant :

1 -> 5 2 -> 1 3 -> 2 4 -> 3 5 -> 4

Cycle -> (15432)

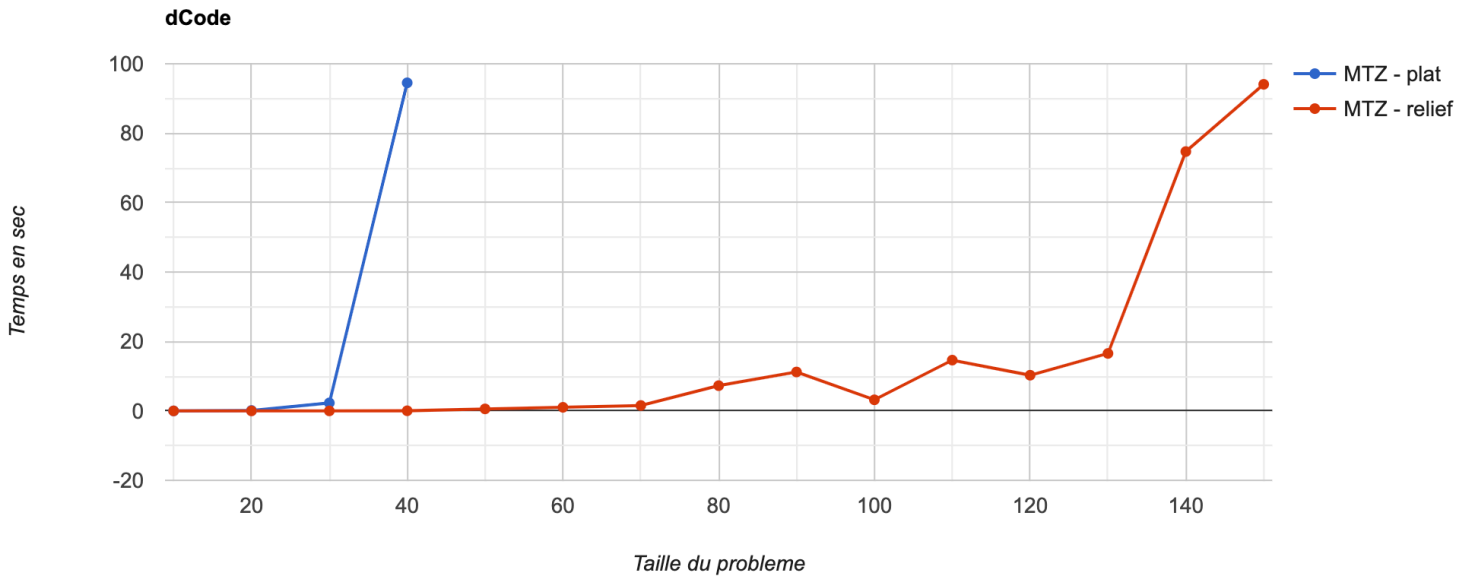
Avec une solution optimale z = 36

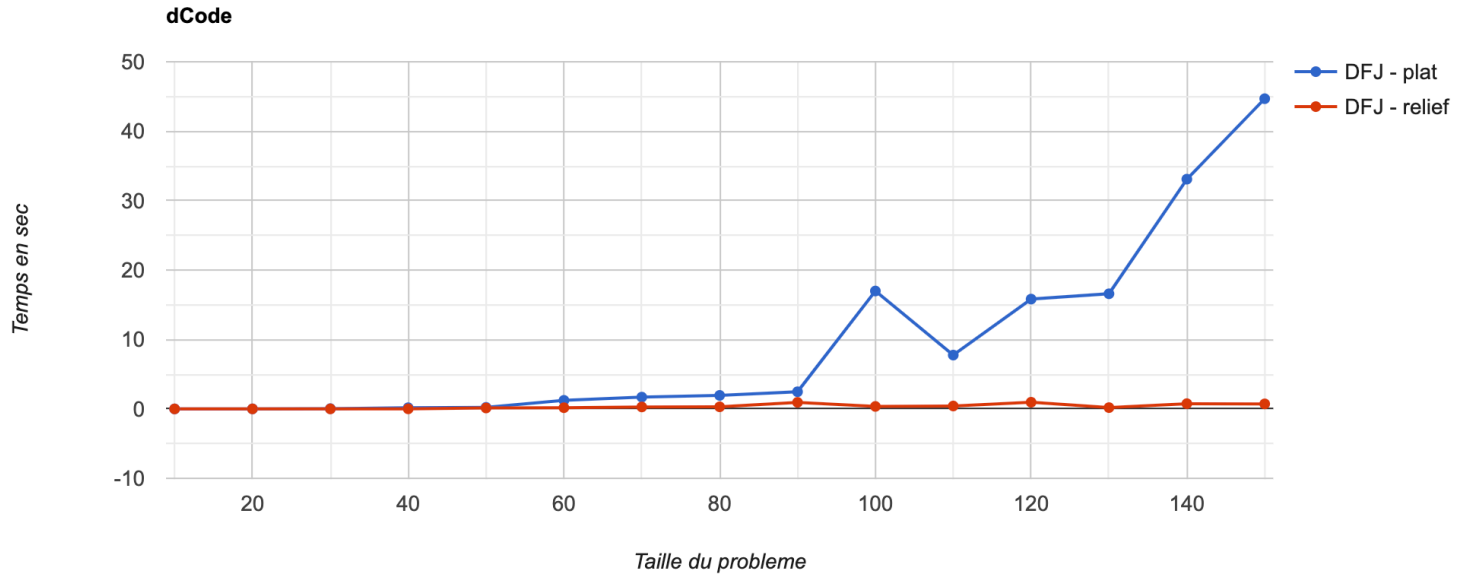
	1	2	3	4	5
1	0	1	0	0	0
2	0	0	1	0	0
3	0	0	0	1	0
4	0	0	0	0	1
5	1	0	0	0	0

Cela confirme que la solution optimale obtenue avec seulement les contraintes (1) et (2) reste optimale même si nous ajoutons une contrainte de type (3') qui casse un sous-tour.

Analyse expérimentale pour les méthodes de résolution exacte:

Pour cette analyse, nous supposons que n représente le nombre de lieux dans les problèmes "plat" et "relief". Les tableaux et courbes présentés ci-dessous indiquent la taille de chaque problème ainsi que le temps nécessaire pour le résoudre par chaque algorithme.





Remarque 1:

Il convient de noter que la présence d'un tiret "-" dans les tableaux indique que le problème correspondant n'a pas été résolu ou le temps de résolution nécessaire a été très long.

Remarque 2:

La colonne 'Contr.' Signifie le nombre de contraintes à ajouter pour casser les sous-tours et construire un seul cycle.

Observations:

Les tableaux et les courbes présentés ci-dessus/ci-contre illustrent clairement que les problèmes asymétriques sont moins complexes à résoudre que les problèmes symétriques. De plus, il est évident que le temps de résolution du premier algorithme augmente beaucoup plus rapidement que celui du deuxième algorithme. Par conséquent, les limites du premier algorithme sont clairement visibles lors de la résolution de problèmes symétriques dont la taille est supérieure à 40.

En outre, nous constatons que la construction du cycle final (pour l'algorithme DFJ) nécessite un nombre important de contraintes pour les instances "plat", afin de casser les sous-tours par rapport aux instances "relief".

En somme, ces résultats mettent en lumière les avantages et les limites des différentes approches de résolution de problèmes d'optimisation, en fonction des caractéristiques des instances considérées.

MTZ - plat

n	Temps (s)	Z
10	0.004470	170
20	0.126152	200
30	2.306254	148
40	94.511424	192
50	-	-
60	-	-
70	-	-
80	-	-
90	-	-
100	-	-
110	-	-
120	-	-
130	-	-
140	-	-
150	-	-

DFJ - plat

n	Temps (s)	Contr.	Z
10	0.001303	2	170
20	0.008380	9	200
30	0.039149	14	148
40	0.176004	16	192
50	0.226997	24	207
60	1.234479	28	134
70	1.698799	40	160
80	1.947858	37	183
90	2.473421	42	157
100	16.983544	57	173
110	7.752113	46	152
120	15.817111	61	138
130	16.586123	63	112
140	33.089629	73	141
150	44.679639	70	146

MTZ - relief

n	Temps (s)	Z
10	0.001914	198
20	0.007562	147
30	0.010161	116
40	0.040516	105
50	0.583671	155
60	1.039511	136
70	1.564216	115
80	7.277698	99
90	11.216648	118
100	3.203872	103
110	14.630800	113
120	10.286870	103
130	16.558137	107
140	74.710785	111
150	94.078099	100

DFJ - relief

n	Temps (s)	Contr.	Z
10	0.001140	2	198
20	0.002428	1	147
30	0.003772	0	116
40	0.009822	2	105
50	0.134669	8	155
60	0.177467	10	136
70	0.286205	11	115
80	0.304049	7	99
90	0.946204	11	118
100	0.360463	8	103
110	0.421670	5	113
120	0.964486	11	103
130	0.190101	1	107
140	0.758651	12	111
150	0.728179	7	100

Conclusion :

Les instances symétriques ont pris plus de temps à résoudre que les instances asymétriques car elles ont plus de contraintes et sont donc plus complexes à résoudre. En effet, les contraintes de la symétrie des distances dans les instances symétriques réduisent le nombre de possibilités de solutions et peuvent causer des blocages dans la recherche de solutions optimales. Cependant, les instances asymétriques n'ont pas de telles contraintes et donc la recherche de solutions peut être plus rapide.

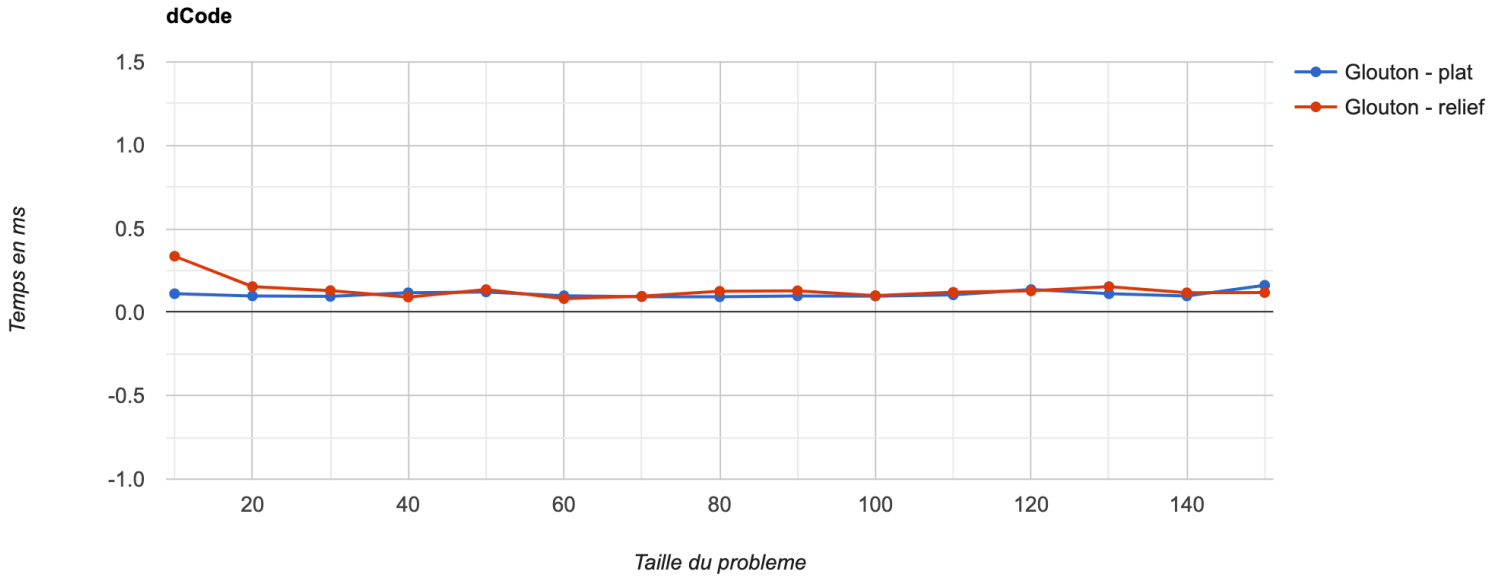
3] Résolution approchée par un algorithme glouton

L'algorithme glouton est une méthode heuristique de résolution de problèmes d'optimisation qui consiste à prendre les décisions localement optimales à chaque étape dans l'espoir d'atteindre une solution globale optimale. Cette algorithme ne garantit pas de trouver la solution optimale, mais peut fournir une bonne approximation en un temps raisonnable.

Voici les étapes de l'algorithme pour résoudre le problème :

1. Choisir un point de départ aléatoire.
2. Visiter le point le plus proche du point actuel qui n'a pas encore été visité.
3. Répéter l'étape 2 jusqu'à ce que tous les points aient été visités.
4. Ajouter l'arête finale pour revenir au point de départ et construire un cycle.

Analyse expérimentale pour la méthode de résolution approchée:



Glouton - plat

n	Temps (ms)	Z
10	0.111	170
20	0.098	234
30	0.095	311
40	0.117	293
50	0.122	442
60	0.099	407
70	0.093	449
80	0.093	511
90	0.098	461
100	0.097	508
110	0.104	462
120	0.136	364
130	0.111	411
140	0.098	379
150	0.162	609

Glouton - relief

n	Temps (ms)	Z
10	0.336	296
20	0.154	187
30	0.13	331
40	0.091	334
50	0.136	334
60	0.082	404
70	0.096	309
80	0.126	390
90	0.129	430
100	0.1	483
110	0.12	406
120	0.128	300
130	0.154	491
140	0.117	445
150	0.118	382

La différence entre l'algorithme glouton (résolution approchée) et les algorithmes de résolution exacte est clairement observable. En effet, le temps de résolution de l'algorithme glouton est très rapide, presque instantané. Toutefois, il convient de souligner que cette méthode fournit une résolution approchée. Dans certains cas, les solutions peuvent être très proches des solutions optimales du problème, tandis que dans d'autres cas, elles peuvent être éloignées de l'optimum, en fonction des valeurs dans la matrice de distances initiale.