Recherche Opérationnelle
Life on Mars ?

LATIF Mehdi

7 avril 2018

1 Preuve

Dans ce sujet, il nous est demander de prouver les deux affirmations suivantes :

— Nous obtiendrons une permutation qui se décompose en un unique cycle, soit une solution admissible pour le problème de voyageur de commerce.

— Nous avons donc ici obtenu une solution admissible et optimale pour cette instance du problème de voyageur de commerce en ne considérant que deux contraintes de l'ensemble (3').

La méthode proposée consiste à résoudre à l'optimalité de manière répétée des sous-problèmes comportant un sous-ensemble de contraintes du problème initialement posé.
On peut supposer que la région admissible du problème du voyageur de commerce est incluse dans celle des sous problèmes. Ainsi, si une solution est admissible et optimale pour le problème du voyageur, elle l'est également pour celle du sous problème considéré.
Par ajout successif de contraintes, on tend à raffiner notre région admissible tout en continuant à résoudre les sous problèmes à l'optimalité. Nous trouvons donc itérativement par résolution des sous problèmes avec de nouvelles contraintes, des bornes supérieures pour la valeur optimale du problème du voyageur de commerce.
Une fois que l'on a obtenu une solution composé d'un seul cycle, on peut en déduire que les valeurs de la solution admissible pour le sous problème considéré et celui du voyageur coincident.

2 Résolution par méthode exacte

Nous allons dans cette partie, présenter la méthode et les différents algorithmes mis en place pour la traduction d'une solution obtenue en permutation puis en produit de cycle disjoints. Pour ce faire, nous prendrons pour exemple l'instance *plat/exemple.dat*

Traduction de la solution obtenue en permutations

Après avoir résolu une première fois le problème proposé avec le solver GLPK (et obtenu une solution optimale partielle à $z = 2220.0$), le programme nous retourne une matrice des contenant l'état des variables de décisions x_{ij} sous la forme :

$$x_{ij} = \begin{cases} 1 & \text{Si l'on va de la ville } i \text{ à la ville } j \\ 0 & \text{Sinon} \end{cases}$$

Dès lors et à l'aide de la fonction *getvalue(m[:x])*, nous récupérons une matrice contenant les valeurs de la variable de décision x_{ij} obtenue après résolution. (type retourné : 7×7 Array{ Float64,2})

```
julia> getvalue(m[:x])
```

$$\begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 \end{pmatrix}$$

Nous pouvons alors appliquer l'algorithme de recherche de permutation dont le code est présenté ci-dessous

```
function permutation(X::Array{Float64,2})
    nbPoint = size(X,1);V=[0 for i in 1:nbPoint]
        for i in 1:nbPoint
            for j in 1:nbPoint
                if (X[i,j]==1) V[i]=j end
            end
        end
    return V
end
```

Nous déclarons tout d'abord un vecteur V ne contenant que des 0 et dont la taille correspond au nombre de points d'intérêt de l'instance considérée. Ce vecteur, une fois retourné, sera la matrice des successeurs des sommets de cette instance.
Pour remplir ce vecteur de successeurs, nous parcourons successivement les lignes et les colonnes de la matrice contenant les valeurs des variables de decisions x_{ij}. Si pour une ligne donnée i, nous trouvons la valeur 1 à la position j, cela signifie que le sommet j sera visité après i ($V[i] = j$).
Nous répetons ce parcours de matrice autant de fois qu'il y a de sommets dans notre graphe à l'aide d'une boucle *pour*. Une amélioration possible serait d'utiliser, pour le parcours des colonnes j, une boucle *tant que* permettant d'arreter le parcours une fois le successeur du sommet i trouvé.
Le résultat à la fin de l'execution de cet algorithme est le suivant :

```
  P = permutation(getvalue(m[:x]))
     Permutations :
  (1 -> 7) (2 -> 6) (3 -> 4) (4 -> 3) (5 -> 1) (6 -> 2) (7 -> 5)
```

Le résultat obtenu est bien celui proposé dans le sujet.

Traduction des permutations obtenues en produit de cycles disjoints

Pour répondre à ce problème, nous avons décidé de mettre en place un parcours en profondeur d'un graphe à l'aide de l'algorithme *DFS*. Le principe de cet algorithme est le suivant :

C'est un algorithme de recherche qui progresse à partir d'un sommet S en s'appelant récursivement pour chaque sommet voisin de S.
Le nom d'algorithme en profondeur est dû au fait qu'il [...] il explore en fait « à fond » les chemins un par un : pour chaque sommet, il marque le sommet actuel, et il prend le premier sommet voisin jusqu'à ce qu'un sommet n'ait plus de voisins (ou que tous ses voisins soient marqués), et revient alors au sommet père.

Wikipédia - Algorithme de parcours en profondeur

Avant d'entamer l'exploration du graphe, nous déclarons deux vecteurs de la taille du vecteur des permutations retourné précédemment.

— *etat* : un vecteur permettant de savoir si le sommet à été visité (0 : Non, 1 : Oui)
— *pere* : un vecteur d'entiers dans lequel, à l'indice i, sera stocké le *sommet père* (ou prédécesseur) du sommet i.

```
nbPoint = size(P,1)
etat = zeros(Int64,nbPoint)
pere = zeros(Int64,nbPoint)
```

Vous trouverez ci dessous l'implémentation de l'algorithme de parcours en profondeur. Son principe est le suivant :
Pour un noeud n qui lui est passé en paramètre, l'algorithme ajoute dans un vecteur (noté sous_cycle), ce noeud, modifie son état pour indiquer que ce noeud à été visité et récupère le sommet suivant de n grâce à la matrice des permutations G que l'on a définie auparavant dans le programme.
Par la suite, l'algorithme tente de savoir si le sommet suivant de n a déjà été visité :
— Si oui , l'algorithme s'arrête et on retourne le vecteur contenant le sous_cycle que l'on vient d'identifier.
— Si non, alors :
1. On définit dans le vecteur *père* que le sommet suivant de n, $G(n)$ a pour prédécesseur n (logique)
2. On applique récursivement l'algorithme de recherche en profondeur sur le sommet suivant $G(n)$ afin de trouver d'autres sommets successeurs appartenant à un sous_cycle démarrant par n.

A la fin de l'execution de cet algorithme, on récupère un vecteur contenant le sous_cycle dont le sommet de départ est n.
Note : Après relecture du code, il s'avère que le vecteur *père* n'est pas utilisé dans le reste de l'algorithme. Il s'agit cependant d'un bon détrompeur afin de vérifier la cohérence des résultats.

```
function DFS(G::Array{Int64,1},n::Int64,ss_cycle::Array{Int64,1},etat::Array{Int64,1},pere::Array{Int64,1})
    ss_cycle = push!(ss_cycle,n)
    etat[n] = 1
    v = G[n]
    if (etat[v]==0)
        pere[v] = n
        DFS(P,v,ss_cycle,etat,pere)
    end
    return ss_cycle
end
```

La méthode *DFS* que l'on vient de détailler est appelé par une méthode d'exploration. Tout d'abord dans cette méthode, nous définissons un vecteur de vecteur de sommets (appelé cycle) qui va recevoir des vecteurs contenant les sous_cycles. Le principe d'exploration est alors le suivant :
A l'aide d'une boucle *pour*, nous parcourons l'ensemble du vecteur d'état afin de trouver un sommet qui n'a pas été encore visité.
Le comportement de l'algorithme est le suivant :
— Si il trouve un sommet qui a déjà été visité, il l'ignore et passe au suivant.
— Si il trouve un voisin qui n'a pas encore été visité, signifiant qu'il s'agit du point de départ d'un nouveau sous_cycle, l'algorithme modifie la valeur du père de ce sommet (mis à 0) puis appelle la fonction *DFS* sur ce même sommet permettant ainsi de rechercher et de retourner le sous_cycle issu de ce sommet.

Lorsque l'algorithme à trouvé un sommet qui n'était pas visité et qu'il a retourné le sous_cycle démarrant par ce voisin, il ajoute ce sous_cycle dans le vecteur cycle.
La fonction d'exploration continue à étudier les sommets tant qu'il reste des parmi ces dernier, certains qui n'ont pas encore été visités.

Une fois que le parcours complet des sommets à été réalisé, l'algorithme s'arrête et retourne le vecteur cycle contenant le ou les sous_cycle(s) qu'il a détecté à partir de la matrice des permutations.

```
function explorer(G::Array{Int64,1},etat::Array{Int64,1},pere::Array{Int64,1})
    cycle = [[0]]
    for i in 1:nbPoint
        if (etat[i] ==0)
            pere[i] = 0
            ss_cycle = DFS(G,i,[0],etat,pere)
            shift!(ss_cycle);push!(cycle,ss_cycle)
        end
    end
    shift!(cycle)
    return cycle
end
```

Cet algorithme bien que fonctionnel nécessite des améliorations. Tout d'abord, il serait intéressant d'apprendre à bien déclarer les vecteurs vides en Julia, ce qui nous permettrait de ne pas avoir à initialiser ces derniers avec un élément dedans puis devoir utiliser par la suite la fonction *shift !()* permettant de supprimer le premier élément contenu.
Dans un second temps, il serait également intéressant d'utiliser des booléens plutôt que d'entiers (0 ou 1) afin d'étudier les états des sommets. Cette approche a été mise en place pour des raisons pratiques ; En effet, lors des affichages, il était plus simple d'identifier les sommets visités par des 0 et 1 plutôt que des true et false (afin de ne pas surcharger les informations affichées).

Présentation d'un exemple

Vous trouverez ci dessous les résultats obtenue sur l'instance d'exemple :

```
julia> include("Projet_LATIF.jl")
Résolution exacte pour plat/exemple.dat points à visiter :
Résolution d'initiale :
> temps total = 2220.0
   Permutations :
(1 -> 7) (2 -> 6) (3 -> 4) (4 -> 3) (5 -> 1) (6 -> 2) (7 -> 5)
> Cycle(s) trouvé(s) : Array{Int64,1}[[1, 7, 5], [2, 6], [3, 4]]
> Nombre de cycle(s) trouvé(s) : 3

Itération n° 1 Cassage de contrainte
> Cycle à casser : [2, 6]
> Taille du cycle à casser : 2
> Nouvelle contrainte : x[2,6] + x[6,2] ? 1

> Nouvelle résolution après ajout de la nouvelle contrainte !
> temps total = 2335.0
   Permutations :
(1 -> 7) (2 -> 5) (3 -> 4) (4 -> 3) (5 -> 6) (6 -> 2) (7 -> 1)
> Cycle(s) trouvé(s) : Array{Int64,1}[[1, 7], [2, 5, 6], [3, 4]]
> Nombre de cycle(s) trouvé(s) : 3

Itération n° 2 Cassage de contrainte
> Cycle à casser : [1, 7]
> Taille du cycle à casser : 2
> Nouvelle contrainte : x[1,7] + x[7,1] ? 1

> Nouvelle résolution après ajout de la nouvelle contrainte !
> temps total = 2575.0
   Permutations :
(1 -> 5) (2 -> 3) (3 -> 4) (4 -> 7) (5 -> 6) (6 -> 2) (7 -> 1)
> Cycle(s) trouvé(s) : Array{Int64,1}[[1, 5, 6, 2, 3, 4, 7]]
> Nombre de cycle(s) trouvé(s) : 1

FIN - Problème résolu :
> temps total = 2575.0
> Nombre d'itération nécéssaires : 3
> Nombre de contraintes ajoutées : 2
> Ordre de parcours des drônes :
1 -> 5 -> 6 -> 2 -> 3 -> 4 -> 7 -> 1.
  0.666449 seconds (247.14 k allocations: 12.325 MiB)
```

Nous pouvons constater que les résultats obtenus sont conformes à ceux proposés par le sujets.

2.1 Analyse expérimentale

	Temps total de parcours	Nb Contraintes ajoutées	Temps machine (secondes)	CPU
Plat 10	170.0	2	0.137014	5.89 k alloc : 393.406 KiB
Plat 20	200	11	0.165501	26.36 k alloc : 2.154 MiB
Plat 30	148.0	14	0.206548	57.40 k alloc : 5.059 MiB
Plat 40	192	20	1.062222	123.52 k alloc : 11.635 MiB
Plat 50	207	23	0.492398	195.61 k alloc : 19.725 MiB
Plat 60	134.0	27	3.375432	311.61 k alloc : 31.073 MiB
Plat 70	160.0	49	17.763255	667.01 k alloc : 72.797 MiB
Plat 80	183.0	38	6.156659	685.97 k alloc : 73.051 MiB
Plat 90	157.0	41	7.125538	911.33 k alloc : 96.495 MiB
Plat 100	173.0	54	41.788991	1.41 M alloc : 157.975 MiB
Plat 110	152.0	44	12.828082	1.42 M alloc : 155.523 MiB
Plat 120	138.0	59	35.971777	2.14 M alloc : 236.806 MiB
Plat 130	112.0	66	45.006563	2.75 M alloc : 323.459 MiB
Plat 140	141.0	69	74.752989	3.30 M alloc : 383.730 MiB
Plat 150	146.0	71	105.200705	3.88 M alloc : 445.682 MiB

TABLE 1 – Résultats de l'analyse expérimentale sur les instances de type plat

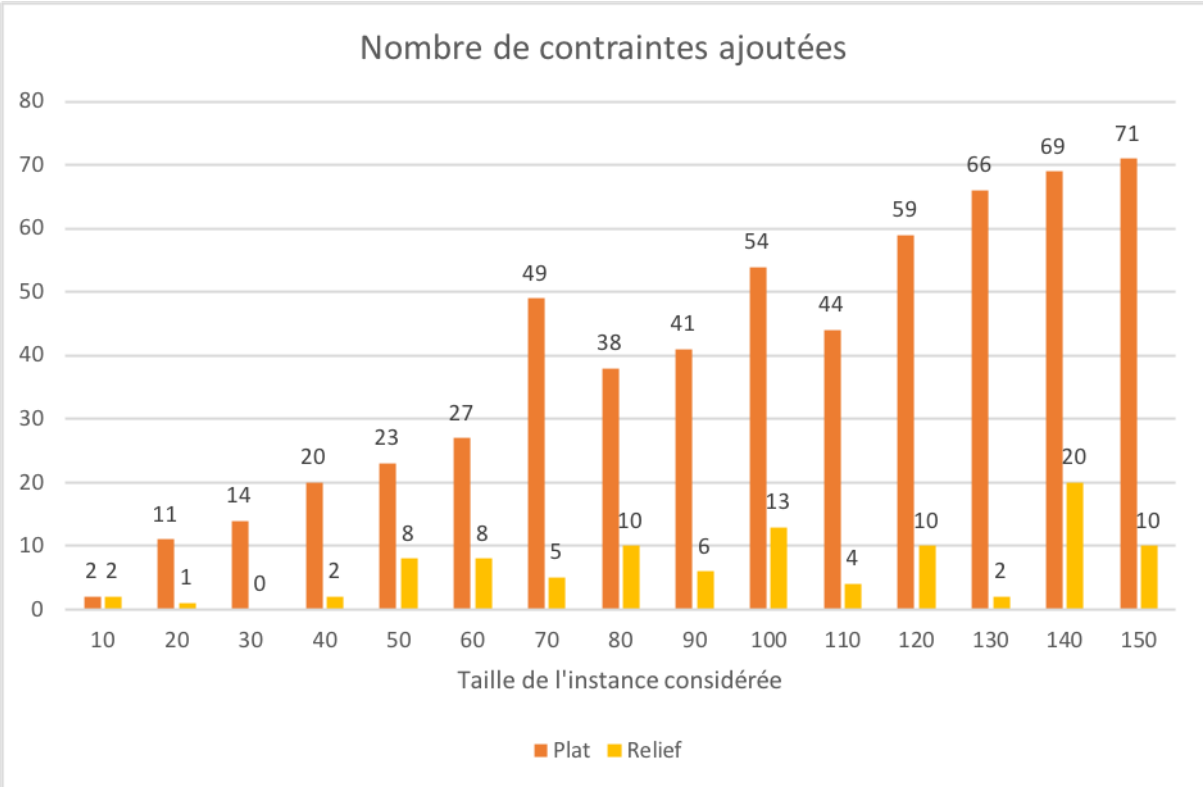
	Temps total de parcours	Nb Contraintes ajoutées	Temps machine (secondes)	CPU
Relief 10	198.0	2	0.130585	5.89 k alloc : 394.094 KiB
Relief 20	147.0	1	0.145352	13.47 k alloc : 937.688 KiB
Relief 30	116.0	0	0.131483	6.80 k alloc : 856.938 KiB
Relief 40	105.0	2	0.155768	49.13 k alloc : 3.593 MiB
Relief 50	155.0	8	0.480462	110.15 k alloc : 9.687 MiB
Relief 60	136.0	8	0.478024	154.46 k alloc : 13.213 MiB
Relief 70	115.0	5	0.384103	175.00 k alloc : 14.856 MiB
Relief 80	99.0	10	1.156912	296.54 k alloc : 26.947 MiB
Relief 90	118.0	6	0.976702	302.92 k alloc : 25.066 MiB
Relief 100	103.0	13	3.671390	521.89 k alloc : 50.510 MiB
Relief 110	113.0	4	1.016204	399.29 k alloc : 31.878 MiB
Relief 120	103.0	10	2.732261	655.98 k alloc : 58.963 MiB
Relief 130	107.0	2	0.819629	487.95 k alloc : 38.457 MiB
Relief 140	111.0	20	9.981334	1.30 M alloc : 134.869 MiB
Relief 150	100.0	10	3.296910	1.02 M alloc : 94.883 MiB

TABLE 2 – Résultats de l'analyse expérimentale sur les instances de type relief

TABLE 3 – Longueurs et nombres de contraintes cassées

	Taille contraintes à casser : nb*(taille)		**Taille contraintes à casser : nb*(taille)**
Plat 10	2x(2)	**Relief 10**	2x(2)
Plat 20	6x(2),2x(8),2x(7), 1x(3)	**Relief 20**	1x(3)
Plat 30	14x(2)	**Relief 30**	0
Plat 40	13x(2), 2x(8),2x(16),2x(3),1x(18)	**Relief 40**	1x(3),1x(13)
Plat 50	23x(2)	**Relief 50**	1x(2),1x(4),1x(16),1x(15),1x(6),1x(8),1x(23),1x(5)
Plat 60	22x(2),2x(19),2x(11),1x(15)	**Relief 60**	2x(3),1x(8),1x(2),1x(17),1x(12),1x(7),1x(13)
Plat 70	36x(2),4x(4),3x(13),2x(35),2x(10),2x(3)	**Relief 70**	2x(2),1x(12),1x(5),1x(8)
Plat 80	33x(2),2x(33),2x(6),1x(8)	**Relief 80**	2x(6),1x(18),1x(33),1x(20),1x(31),1x(7),1x(10),1x(34),1x(3)
Plat 90	41x(2)	**Relief 90**	1x(2),1x(11),1x(22),1x(23),1x(8),1x(10)
Plat 100	46x(2),2x(4),2x(21),2x(5),1x(10),1x(14)	**Relief 100**	3x(11),2x(4),2x(15),1x(7),1x(27),1x(29),1x(17),1x(37),1x(13)
Plat 110	44x(2)	**Relief 110**	2x(5),1x(7),1x(2)
Plat 120	58x(2),1x(12)	**Relief 120**	2x(6),2x(10),1x(3),1x(8),2x(31),1x(11),1(49)
Plat 130	66x(2)	**Relief 130**	1x(6),1x(4)
Plat 140	65x(2),3x(20),1x(8)	**Relief 140**	1x(38),1x(12),2x(11),2x(65),3x(2),2x(4),1x(24),1x(3),1x(5),1x(32),1x(52),1x(29),1x(6),1x(8),1x(41)
Plat 150	69x(2),1x(49),1x(38)	**Relief 150**	2x(2),2x(3),2x(4),1x(5),1x(8),1x(24),1x(34)

Analyse des résultats



L'analyse des résultats concernant le nombre de contraintes cassantes à ajouter pour une taille d'instance donnée nous montre que celle de type plat nécessitent beaucoup plus d'ajout avant d'arriver à l'optimum.
Cela peut s'expliquer par le fait que dans les instances de type plat, les distanciers sont symétriques ; il y a donc beaucoup plus de sous tours de taille 2 à supprimer. Cette hypothèse est confirmée par l'étude du troisième tableau dans lequel nous pouvons voir le nombre de contraintes par taille à casser pour chaque instance du problème considéré.
L'importance du nombre de sous tours de taille 2 explique également l'importante croissance des temps d'exécution en fonction du nombre de sommets dans les instances considérés.
Moralité, l'étudiant du master VICO doit améliorer les données fournies pour que l'on puisse effectuer nos mesures sur les distanciers de type relief.

3 Résolution par méthode approchée

Vous trouverez ci dessous les résultats obtenue sur l'instance d'exemple :

```
julia> include("Projet_LATIF.jl")
Résolution approchée pour plat/exemple.dat points à visiter :
Proches voisins
P : [1, 7, 4, 3, 6, 2, 5, 1]
Parcours de base :
[7, 5, 6, 3, 1, 2, 4]
Coût de base :
2586
################################################################################
??? < 0 trouvé :
          ?((i = 3, j = 6);(i' = 2, j' = 5) = -11
```

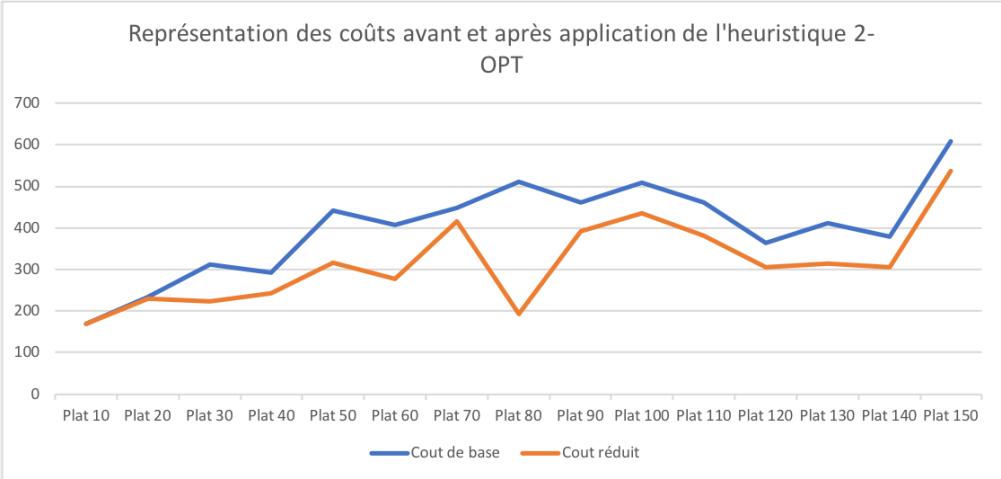
```
          Parcours initial : [7, 5, 6, 3, 1, 2, 4]
          Parcours modifié : [7, 6, 2, 3, 1, 5, 4]
          Coût init 2586 - Coût modif 2575
          Variation de coût : -11
          > Coût inférieur obtenu avec ce nouveau parcours -> Solution améliorante.
          Parcours retourné : [7, 6, 2, 3, 1, 5, 4]
################################################################################
Nombre de solutions potentiellement améliorantes détectées : 1
  0.104846 seconds (18.30 k allocations: 952.895 KiB)
Cout AV = 2586
Cout AP = 2575
Delta coût : -11
Parcours final : [7, 6, 2, 3, 1, 5, 4]
```

3.1 Analyse expérimentale

OPT	Coût init	Cout réduit	Delta Coût	Nb Solutions potentiellement améliorantes	Nb Total de Comparaisons	Temps machine (secondes)	CPU
Plat 10	170	170	0	0	73	0.057533	7.06 k alloc 383.305 KiB
Plat 20	234	230	4	2	323	0.050759	7.49 k alloc : 402.070 KiB
Plat 30	311	223	88	8	243	0.051375	9.21 k alloc : 477.180 KiB
Plat 40	293	243	50	6	777	0.047224	9.06 k alloc 472.305 KiB
Plat 50	442	317	125	4	94	0.047263	8.68 k alloc : 456.367 KiB
Plat 60	407	277	130	19	1026	0.055708	15.66 k alloc : 772.211 KiB
Plat 70	449	415	34	8	1047	0.049780	11.23 k alloc : 573.617 KiB
Plat 80	511	193	318	10	1463	0.054752	12.87 k alloc : 650.055 KiB
Plat 90	461	391	70	14	1653	0.045220	16.01 k alloc : 795.805 KiB
Plat 100	508	435	73	13	1552	0.050341	16.18 k alloc : 806.852 KiB
Plat 110	462	382	80	21	2247	0.054896	22.97 k alloc : 1.096 MiB
Plat 120	364	305	59	9	2574	0.052555	14.54 k alloc : 731.992 KiB
Plat 130	411	315	96	12	2032	0.051334	17.71 k alloc : 880.539 KiB
Plat 140	379	305	74	4	3562	0.050174	11.02 k alloc : 569.492 KiB
Plat 150	609	537	72	30	4704	0.055015	37.10 k alloc : 1.745 MiB

TABLE 4 – Résultats de l'analyse expérimentale sur les instances de type plat

Analyse des résultats



Les améliorations constatées par applications de l'heuristique 2-OPT varient en fonction des instances du problème considéré. Cependant, il ne faut pas oublier que le principe d'une heuristique est de fournir un rapidement un optimum local et non global pour des problèmes difficiles (TSP étant de classe $\mathcal{NP}$ ~Complet. Il est donc, à mon sens, normal de trouver de tels résultats.