

TP3 Métaheuristiques

M1 informatique s1 — Algorithmique Avancée

Université Paul Sabatier



Table de contenus

Exercice 2: Voyageur de commerce (noté)	2
Algorithmes implémentés	2
Récupération des données	2
random_sol(n)	2
distance(ville1, ville2)	2
f(X)	2
steepest_hill_climbing(X)	2
steepest_hill_climbing_redemarrage(X)	2
recherche_tabou(X, taille)	2
Voisinages implémentés	3
meilleur_voisin(X)	3
meilleur_voisin_non_tabou(X)	3
Configuration et Résultats	4
Conclusions	5

Exercice 2: Voyageur de commerce (noté)

Algorithmes implémentés

Récupération des données

[question 2.0 - ligne 16 - exo2.py]

On lit le fichier et on génère une liste de strings correspondant chaque case à chaque ligne. On récupère la première ligne (le nombre n), on fait le "cast" en "int" et on l'enlève de la liste. Après, on transforme chaque string en une liste (donc on obtient une liste de listes, une matrice), on enlève l'identifiant de chaque ville, car on peut utiliser les indices du tableau +1 pour le trouver, et on fait le "cast" de tous les nombres à des entiers. De façon qu'on obtient une liste de villes où chaque ville a la forme $ville[x, y]$.

random_sol(n)

[question 2.1 - ligne 34 - exo2.py]

On utilise la méthode itérative basée sur l'échange aléatoire, comme décrit dans le sujet: on génère la liste ordonnée des villes et, pour chaque ville, on sélectionne une ville aléatoirement et on les permute.

distance(ville1, ville2)

[question 2.2 - ligne 45]

On applique la fonction de distance euclidienne, tel que demandé dans le sujet, avec la formule: $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Correspondant $ville1$ et $ville2$ à x et y respectivement.

f(X)

[question 2.2 - ligne 48]

On calcule la distance entre la localisation initiale (0,0) et la première ligne, puis pour chaque ville, on calcule la distance avec la suivante et à la fin, on calcule la distance entre la dernière ligne et la localisation initiale (0,0).

steepest_hill_climbing(X)

[question 2.4 - ligne 72]

C'est l'algorithme expliqué en pseudo-code dans le sujet, "traduit" à python.

steepest_hill_climbing_redemarrage(X)

[question 2.4 - ligne 85]

C'est une boucle jusqu'à MAX_ESSAIS où à chaque tour, on compare le résultat de `steepest_hill_climbing(X)` avec le dernier meilleur voisin, puis on garde le meilleur.

recherche_tabou(X, taille)

[question 2.5 - ligne 121]

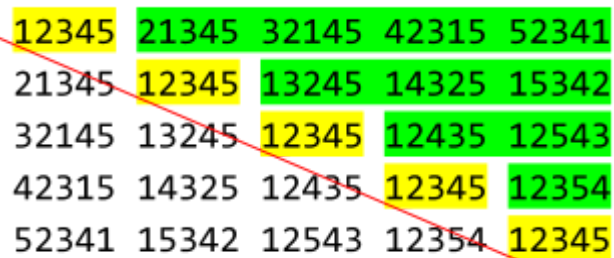
C'est l'algorithme expliqué en pseudo-code dans le sujet, "traduit" à python.

Voisinages implémentés

`meilleur_voisin(X)`

L'algorithme de voisinage n'est pas décrit dans le sujet. Donc, on va détailler le processus suivi pour arriver à l'algorithme trouvé. D'abord, on a pris un exemple de solution: [1, 2, 3, 4, 5] qu'on va noter 12345. On a cherché "à la main" tous ces voisins de façon ordonnée en obtenant une matrice de voisins, où chaque ligne d'indice i contient les voisins issus de toutes les permutations possibles entre la ville d'indice i et les autres, comme on peut voir dans le schéma à côté.

On peut observer que cette matrice est symétrique et que le grand diagonal (marquée en jaune) a toujours la solution initiale. À droite de cette diagonale, on obtient des nouveaux voisins (marquées en vert)



12345	21345	32145	42315	52341
21345	12345	13245	14325	15342
32145	13245	12345	12435	12543
42315	14325	12435	12345	12354
52341	15342	12543	12354	12345

et à gauche, on obtient des voisins répétés. En conséquence, on va s'intéresser à générer les solutions qui sont à droite de la diagonale. Vu que les autres solutions, même si elles sont répétées, sont toujours valides, on pourrait tout parcourir en gardant les solutions dans un set par exemple et éviter les répétitions, mais c'est plus optimal de chercher exactement ce dont on a besoin.

Après avoir observé les relations entre les solutions d'une même ligne, on peut trouver un algorithme qui puisse générer tous ces voisins: On va parcourir ligne par ligne de haut en bas, et après de gauche à droite, en partant chaque fois de la solution de départ (la diagonale). Pour trouver les solutions d'une ligne i , on va permuter la ville d'indice aussi i , avec toutes les autres villes, sauf avec les villes qui ont un indice inférieur à i , qui seront "ignorées". Autrement dit, on va faire les permutations qu'avec les villes qui suivent la ville d'indice i uniquement, en "ignorant" à chaque ligne plus de villes.

[question 2.3 - ligne 60 - exo2.py]

Dans l'algorithme de `meilleur_voisin(X)`, on applique l'algorithme de voisinage expliqué en haut, mais au lieu de sauvegarder chaque voisin dans, par exemple, une liste, on va comparer sa valeur de distance avec celle du meilleur courant (au début, X) afin de garder que le meilleur parmi les deux. On applique la fonction $f(X)$ dans les deux solutions et on garde la solution dont la valeur est la plus basse.

`meilleur_voisin_non_tabou(X)`

[question 2.5 - ligne 98] `trouver_voisins(X)`

[question 2.5 - ligne 107] `trouver_voisins_non_tabous(X)`

[question 2.5 - ligne 114] `meilleur_voisin_non_tabou(X)`

Ces fonctions sont des modifications de l'algorithme `meilleur_voisin(X)` afin de pouvoir appliquer l'algorithme Tabou. Au lieu de garder que le meilleur, on les garde tous, on les filtre avec la liste des voisins tabous, et finalement, on sélectionne le meilleur parmi les filtrés.

Configuration et Résultats

Par rapport à la configuration, j'ai trouvé que les algorithmes sont beaucoup plus lents que dans l'exercice 1. À cause de ça, j'ai dû modifier la configuration par défaut et réduire tous les deux MAX_DEPL et MAX_ESSAIS à 10. Même si pour instances petites (comme tsp5.txt) ça ne va pas affecter, pour des instances volumineuses (comme tsp101.txt) il y aura sans doute un impact dans les résultats.

À cause de la longueur des solutions de cet exercice et du fait que pour les comparer, on n'a besoin que de ses valeurs, les solutions ne sont pas montrées dans le rapport, mais sont affichés pendant l'exécution du programme.

Résultats SHC (avec options par défaut) pour le fichier **tsp5.txt**

Sans redémarrage: 2 solutions différentes de $f(X)$ (existence de minimum local)			
Essai	Solution X	$f(X)$	Nb déplacements
1	[3, 5, 4, 2, 1]	194.04052963659356	2
2	[1, 3, 2, 5, 4]	196.12466980422548	2
3	[1, 2, 4, 5, 3]	194.04052963659356	3

Avec redémarrage: 1 seule solution de $f(X)$			
Essai	Solution X	$f(X)$	Nb déplacements
1	[3, 5, 4, 2, 1]	194.04052963659356	31
2	[3, 5, 4, 2, 1]	194.04052963659356	27
3	[1, 2, 4, 5, 3]	194.04052963659356	24

Résultats Tabou (avec options par défaut) pour le fichier **tsp5.txt**

Les mêmes qu'avec SHC sans redémarrage: parfois il s'arrête au minimum local de $f(X)$ = 196.12466980422548, même avec le tableau Tabou de longueur 1000. Dans les deux cas (longueur 10 et 1000) et dans les 3 essais, l'algorithme Tabou a donné exactement les mêmes résultats avec toujours le maximum de déplacements atteint (10).

Résultats pour le fichier **tsp101.txt**

Les algorithmes SHC et Tabou donnent des résultats similaires et sont très lents (plusieurs minutes au total autour des 5 minutes), notamment le SHC avec redémarrage. Cependant, ce dernier donne des résultats légèrement meilleurs que les autres (par questions d'espace on ne montre pas les solutions obtenues):

Essai	SHC sans redémarrage et Tabous		SHC avec redémarrage	
	$f(X)$	Nb dépl.	$f(X)$	Nb dépl.
1	2892.447295885964	10 (max)	2883.3699039291787	100 (max)
2	3067.928397431669	10 (max)	2757.0572944098662	100 (max)
3	3054.524168123682	10 (max)	2863.254613000916	100 (max)

Conclusions

On peut observer que l'algorithme SHC avec redémarrage est le plus lent, mais aussi le plus précis (les minimums trouvés sont toujours inférieurs ou égales aux minimums trouvés avec les autres algorithmes). En même temps, on peut observer que l'algorithme Tabou obtient toujours les mêmes résultats que le SHC sans redémarrage, mais en utilisant le maximum des déplacements possibles. La chose la plus surprenante, pourrait être qu'il n'y a pas de différence observable en fonction de la longueur du tableau de Tabou, même avec une relation de 1:100. C'est possible que ça soit à cause d'une configuration trop restrictive, en ayant un très bas nombre maximal de déplacements par rapport à la grandeur de l'instance.

Pour le fichier `tsp101.txt`, aucun des algorithmes a trouvé une même réponse tout le temps, même pas le SHC avec redémarrage. Ça veut dire que cette instance est pleine de minimums locales et que la configuration est trop restrictive pour pouvoir trouver des résultats plus précis. Probablement avec des ordinateurs plus puissants et une configuration plus exigeante, on pourrait trouver des minimums encore inférieurs.