POLYTECHNIQUE
MONTRÉAL

INF8215 : Intelligence artificielle : méthodes et algorithmes

---

LABORATOIRE 1 : Problème du voyageur de commerce

Erwan Marchand - 1659273

Ludovic Font - 1758936

Louis Tan-Therrien - 1634883

Lundi 18 septembre 2017

3. À vos ordinateurs

Modifications apportées au fichier de départ

La version du code que nous avons remise comporte des changements dans les classes *Solution* et *A_star*. Voici un aperçu de ces modifications.

Solution

Nous avons modifié la classe *Solution* en initialisant correctement les valeurs de tous les attributs qu'elle contient. La fonction *add_edge* qui ajoute une ville à la solution courante ajoute le coût du nouveau lien à la solution. Elle enlève la ville ajoutée à la liste des villes non-visitées et l'ajoute à la liste des villes visitées. Enfin, nous avons renommé la fonction *print* pour *printGraph* puisque nous n'arrivons pas à surcharger la fonction sans que cela ne fasse planter l'IDE.

A_star

Nous avons supprimé l'import *from heapqueue.binary_heap import BinaryHeap* puisqu'il faisait planter le code et n'avait aucune utilité pour ce travail. Nous avons complété l'initialisation des attributs de la classe et nous avons ajouté une variable globale qui nous indique combien de noeuds ont été créés ainsi que l'attribut *heuristic_cost* servant au calcul heuristique. L'attribut *h* est relié à la fonction isN2betterThanN1.

Dans la fonction *explore_node* une variable globale tient le compte des noeuds visités. Cette fonction itère sur la liste ordonnée des noeuds non-visités et ajoute les villes dans une queue de priorité si cet ajout ne crée pas de cycle. La fonction isN2betterThanN1 se sert d'une variable globale pour compter le nombre de comparaisons effectuées. L'attribut h sert à éviter de recalculer inutilement l'amélioration heuristique proposée en 4.2.1. Nous avons noté une nette amélioration temporelle en procédant de cette façon.

Les fonctions *closest_unexplored_city_from_current* et *closest_unexplored_city_to_source* Seront traitées dans la partie 4.2 de ce travail afin d'éviter les répétitions.

Exécution : Lancez maintenant l’algorithme A* et reportez vos résultats

Le tableau contenant tous les résultats de l’algorithme se trouvent dans l’annexe 1 de ce rapport. Sans heuristique, on peut constater que nous pouvons obtenir les résultat optimal, cependant, le temps d'exécution augmente rapidement, nous empêchant d'exécuter l’algorithme en un temps raisonnable pour plus de 12 villes.

4. Heuristique A*

4.1 Arbre couvrant minimum (MST)

Avec l’utilisation de l’algorithme de Kruskal et du MST pour calculer une heuristique, on peut s'apercevoir que nous pouvons exécuter notre algorithme pour 15 villes. L’exemple des 15 villes est d’ailleurs le plus rapide à exécuter, on peut donc supposer que les valeurs de cet exemple se pretent tres bien à notre nouvel algorithme.

4.2 Affiner la borne inférieure

4.2.1 La distance de la ville actuelle à v, la ville la plus proche

La fonction *closest_unexplored_city_from_current* vérifie s’il reste des villes non visitées et retourne le coût de la ville la plus proche du dernier noeud de la solution courante. Sinon elle retourne zéro. On vérifie les cas où la ville courante est la source ou la destination de la ville la plus proche. On ajoute ainsi ce coût a l’heuristique MST précédemment calculée à l’aide de Kruskal.

Le résultat de cette modification, présent également dans le tableau de l’annexe, est décevant. En effet, pour un cas (10 villes), le coût optimal est dépassé. Cela vient du fait qu'ajouter à Kruskal une valeur de parcours pouvant déjà être inclus dans le MST a pour résultat que, la somme de l’heuristique et de la solution courante peut dépasser la solution optimale, entraînant ainsi un mauvais résultat. Cette heuristique n’est donc pas bonne à utiliser.

4.2.2 La plus petite distance entre une ville non visitée et le point de départ

La fonction *closest_unexplored_city_to_source* procède de la même façon mais s’affaire à déterminer le coût de la ville la plus proche de la ville de départ.

Le résultat de cette modification est très positif. En effet, contrairement à l'utilisation de la distance précédente, nous conservons la valeur optimale du meilleur chemin entre les villes. En effet, cette heuristique reste inférieur ou égale à la solution optimal après ajout du coût de la solution actuelle.

5. Bonus

Sachant que certaines solutions vues en classe ne garantissent pas nécessairement un résultat optimal:

5.1 Pouvez-vous approcher la solution optimale du problème N17.data avec d'autres techniques de recherches vues en classe?

Il est possible d'utiliser une approche par voisinage afin d'approcher la solution optimale du problème N17. Cette méthode consiste en, prendre une solution aléatoire, interchanger aléatoirement 2 routes, si le résultat obtenu est meilleur, on garde la solution, sinon, on garde la solution avec une probabilité exponentiellement inversement proportionnelle au coût de la nouvelle solution soustrait par le coût de l'ancienne. On diminue progressivement la température (qui qualifie également la probabilité de changement en cas de solution moins bonne) et on arrête l'algorithme une fois que cette température est trop faible pour induire le moindre changement.

Le code de cet algorithme se trouve dans un fichier à part entière du reste du code ("TPun-neighborhood-search") car il nécessitait un changement important de la structure.

Avec cette approche, on peut trouver des solutions proches de la solution optimale (des optimums locaux) mais également parfois la solution optimales. Ci dessous, on a un cas où l'algorithme a atteint la solution optimale :

0
3
12
6
7
5
16
13
14
2
10
9
1

4
8
11
15
0
COST - 2085.0
number of better solutions found : 1796
number of random changes : 758
duration : 31.121000051498413 seconds

5.2 Pouvez vous améliorer l'heuristique proposée dans le TP?

Une heuristique à laquelle nous avons pensé serait une heuristique calculant la somme des coût de routes entre les villes non visitées et d'autres villes non visitées. Par exemple, si les villes 4, 5, 6 et 0 ne sont pas encore visitées (0 sera la 1ere visitées mais reste dans la liste des villes non visitées), l'heuristique vaudrait la somme des routes de la ville 4 vers la ville 5, 6 ou 0 (la plus proche), de la ville 5 vers la ville 4, 6 ou 0 (la plus proche) et de la ville 6 vers la ville 4, 5 ou 0 (la plus proche). On ne considère pas de route partant de 0 sachant qu'elle doit déjà avoir étée ajoutée au chemin au tout début de l'algorithme.

Les résultats de l'utilisation de cette heuristique seule se trouvent dans la tableau suivant :

	10 villes	**12 villes**	**15 villes**
Ordre parcours	0 - 7 - 8 - 1 - 9 - 2 - 5 - 3 - 6 - 4 - 0	0 - 9 - 2 - 4 - 10 - 6 - 8 - 1 - 3 - 11 - 7 - 5 - 0	0 - 12 - 1 - 14 - 8 - 4 - 6 - 2 - 11 - 13 - 9 - 7 - 5 - 3 - 10 - 6
Coût	135	1733	291
Nombre de noeuds créés	21047	67988	58554
Nombre de noeuds visités	6295	15142	8150
Temps d'execution (secondes)	32.3	115.0	111.3

Tableau 5 : Résultats avec l'algorithme A* sans heuristique

Nous retrouvons toujours bien la solution optimale, cependant, les temps d'exécutions sont plus élevés qu'avec l'utilisation des autres heuristiques.

ANNEXE I

Voici un tableau qui résume les différentes heuristiques utilisées dans le cadre de travail pratique:

(1) : L'heuristique A* en utilisant l'algorithme du MST avec Kruskal
(2) : La distance de la ville actuelle à la ville la plus proche
(3) : La plus petite distance entre une ville non visitée et le point de départ

		Sans heuristique	**Avec heuristique (1)**	**Avec heuristiques (1) et (2)**	**Avec heuristiques (1) et (3)**	**Avec heuristiques (1), (2) et (3)**
10 villes	**Ordre parcours**	0 - 4 - 6 - 3 - 5 - 2 - 9 - 1 - 8 - 7 - 0	0 - 4 - 9 - 2 - 5 - 3 - 6 - 1 - 8 - 7 - 0	0 - 7 - 6 - 3 - 5 - 2 - 9 - 1 - 8 - 4 - 0	0 - 7 - 8 - 1 - 9 - 2 - 5 - 3 - 6 - 4 - 0	0 - 7 - 6 - 3 - 5 - 2 - 9 - 1 - 8 - 4 - 0
	Coût	135	135	136	135	136
	Nombre de noeuds créés	116620	7736	9055	5346	5868
	Nombre de noeuds visités	41621	2178	22409	1333	1432
	Temps d'execution (secondes)	32.4	7.9	11.0	6.6	7.7
12 villes	**Ordre parcours**	0 - 9 - 2 - 4 - 10 - 6 - 8 - 1 - 2 - 11 - 7 - 5 - 0	0 - 5 - 7 - 11 - 3 - 1 - 8 - 6 - 10 - 4 - 2 - 9 - 0	0 - 5 - 7 - 11 - 3 - 1 - 8 - 6 - 10 - 4 - 2 - 9 - 0	0 - 5 - 7 - 11 - 3 - 1 - 8 - 6 - 10 - 4 - 2 - 9 - 0	0 - 5 - 7 - 11 - 3 - 1 - 8 - 6 - 10 - 4 - 2 - 9 - 0

	Cout	1733	1733	1733	1733	1733
	Nombre de noeuds créés	4134561	5106	15482	2336	6862
	Nombre de noeuds visités	1164095	902	2926	371	1165
	Temps d'execution (secondes)	1186.1	7.8	26.2	4.6	13.0
15 villes	**Ordre parcours**		0 - 12 - 1 - 14 - 8 - 4 - 6 - 2 - 11 - 13 - 9 - 7 - 5 - 3 - 10 - 0	0 - 10 - 3 - 5 - 7 - 9 - 13 - 11 - 2 - 6 - 4 - 8 - 14 - 1 - 12 - 0	0 - 10 - 3 - 5 - 7 - 9 - 13 - 11 - 2 - 6 - 4 - 8 - 14 - 1 - 12 - 0	0 - 10 - 3 - 5 - 7 - 9 - 13 - 11 - 2 - 6 - 4 - 8 - 14 - 1 - 12 - 0
	Cout		291	291	291	291
	Nombre de noeuds créés		263	430	145	231
	Nombre de noeuds visités		33	56	19	29
	Temps d'execution (secondes)		0.8	1.5	0.6	0.9