Université Paris Dauphine

Intelligence Artificielle

Voyageur de commerce

Etudiant
Alexy Abi Hanna Daher

Professeur
M. Paolo Viappiani

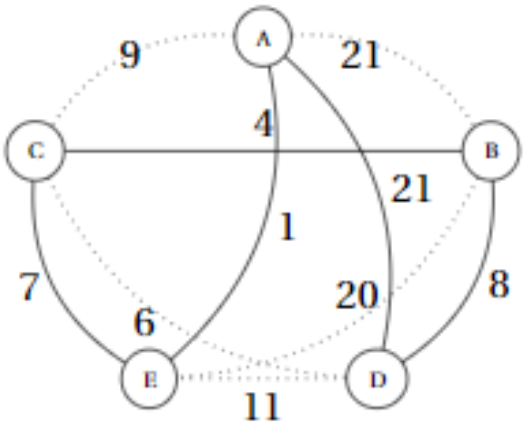


Table des matières

1 Introduction **2**

2 Utilisation **2**

3 Graphe **2**
3.1 Produire un graphe 2
3.2 Représentation d'un graphe 2

4 Recherche informée **3**
4.1 Minimum Spanning Tree 3
4.2 Etat 3
4.3 A* 3

5 Recherche locale **3**
5.1 Hill Climb 3

6 Quelques réflexions **3**

7 Quelques manipulations **4**
7.1 Exemple du sujet 4
7.2 Graphe aleatoire 4

1 Introduction

On se propose dans ce projet de résoudre des instances du problème du voyageur de commerce. Dans ce problème, étant donnés un ensemble de villes et des distances entretoutes les paires de villes (on peut se représenter un graphe complet non-orientéet avec des poids sur les arêtes), on cherche à trouver un plus court chemin qui visite chaque ville exactement une fois et qui se termine dans la ville initiale.On parlera decircuit Hamiltonien. Les villes sont représentées par des points dans un espace à deux dimensions.

2 Utilisation

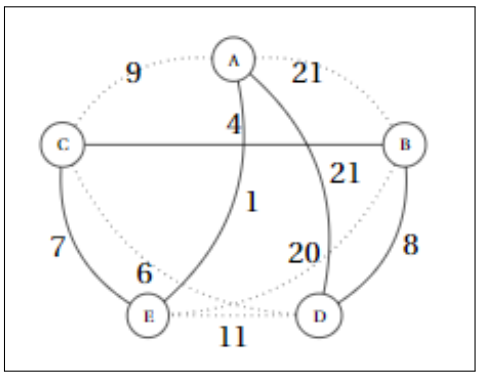
Après avoir compiler et éxecuter le programme, taper 1 pour éxecuter les algorithmes sur l'exemple du sujet du projet ou taper 0 pour générer aléatoirement un graphe et aussi éxecuter les algorithmes sur ce nouveau graphe.

3 Graphe

3.1 Produire un graphe

Dans la classe PathGenerator, on génère un graphe aléatoire par rapport au nombre de villes mis en argument. Puis on ajoute des arêtes de poids aléatoires dans le graphe.

3.2 Représentation d'un graphe




```
Graph :
   A   B   C   D   E
A 000 021 009 021 001
B 021 000 004 008 020
C 009 004 000 006 007
D 021 008 006 000 011
E 001 020 007 011 000
```

4 Recherche informée

4.1 Minimum Spanning Tree

Pour calculer l'heuristique, on utilise la valeur de l'arbre couvrant minimal.

```
MST : [(A, E, 1), (E, C, 7), (C, B, 4), (C, D, 6)]
MST weigth : 18
```

4.2 Etat

Un état (ou State) contient la ville courante, les villes explorées et l'état précédent. Pour trouver le chemin, il suffira de revenir en arrière dans les états précédents (comme une liste chainée). On utilisera cet objet dans l'algorithme A*.

4.3 A*

On a developpé l'algorithme A* dans la fonction récursive aStar :
— Si la liste 'explored' contient toutes les villes alors on renvoie le dernier état qu'on a trouvé
— Sinon, on choisi le plus petit élement de la 'frontier' (la ville qui a la plus petite heuristique)
— On ajoute la ville dans la liste 'explored'
— On ajoute tous les voisins de la ville dans la 'frontier' avec leur heuristique calculé à l'aide du MST
— On appelle de nouveau la fonction avec la nouvelle 'frontier'
Enfin, lors de l'exéution, on affiche la 'frontier' et la liste des villes 'explored'

5 Recherche locale

5.1 Hill Climb

On a developpé l'algorithme Hill Climb dans la fonction recursive hillClimb :
— Si on trouve la ville "goal' ou il ne reste plus de ville à parcourir, on renvoie la liste des vecteur 'vectors'
— On parcours les villes qui restent à parcourir et on choisi la ville qui a la plus petite heuristique.
— On ajoute le vecteur entre le 'current' et la ville qu'on a trouvé dans la liste 'vectors'
— On enleve la nouvelle ville de 'remainingCities'
— On appelle de nouveau la fonction hillClimb avec nos nouvelles informations

6 Quelques réflexions

Après plusieurs essaies, on trouve que l'algorithme A* est beaucoup plus efficace que l'algorithme Hill Climb mais il prend plus de temps pour trouver la solution.

7 Quelques manipulations

7.1 Exemple du sujet

A* :

```
Astar :
explored : [A]
frontier : []
explored : [A, E]
frontier : [(C, 46), (B, 58), (D, 58)]
explored : [A, E, C]
frontier : [(D, 29), (B, 38), (D, 58), (B, 58), (C, 46)]
explored : [A, E, C, B]
frontier : [(D, 22), (B, 38), (D, 29), (B, 58), (C, 46), (D, 58)]
explored : [A, E, C, B, D]
frontier : [(D, 22), (B, 38), (D, 29), (B, 58), (C, 46), (D, 58)]
vectors  : [(A, E, 1), (E, C, 7), (C, B, 4), (B, D, 8), (D, A, 21)]
path  : A->E->C->B->D->A
Weight : 41
Duration : 8378 microseconds
```

Hill Climb :

```
Hill Climb :
current vectors : []
current vectors : [(A, D, 21)]
current vectors : [(A, D, 21), (D, E, 11)]
current vectors : [(A, D, 21), (D, E, 11), (E, B, 20)]
current vectors : [(A, D, 21), (D, E, 11), (E, B, 20), (B, C, 4)]
vectors  : [(A, D, 21), (D, E, 11), (E, B, 20), (B, C, 4), (C, A, 9)]
path  : A->D->E->B->C->A
Weight : 65
Duration : 477 microseconds
```

7.2 Graphe aleatoire

Graphe :

```
Graph :
    A   B   C   D   E
A 000 009 004 002 006
B 009 000 004 003 004
C 004 004 000 009 002
D 002 003 009 000 004
E 006 004 002 004 000
```

A* :

```
Astar :
explored : [A]
frontier : []
explored : [A, D]
frontier : [(C, 17), (E, 19), (B, 22)]
explored : [A, D, B]
frontier : [(E, 14), (C, 17), (B, 22), (E, 19), (C, 19)]
explored : [A, D, B, C]
frontier : [(E, 8), (C, 17), (E, 14), (E, 19), (C, 19), (B, 22)]
explored : [A, D, B, C, E]
frontier : [(E, 8), (C, 17), (E, 14), (E, 19), (C, 19), (B, 22)]
vectors  : [(A, D, 2), (D, B, 3), (B, C, 4), (C, E, 2), (E, A, 6)]
path  : A->D->B->C->E->A
Weight : 17
Duration : 8303 microseconds
```

Hill Climb :

```
Hill Climb :
current vectors : []
current vectors : [(A, E, 6)]
current vectors : [(A, E, 6), (E, D, 4)]
current vectors : [(A, E, 6), (E, D, 4), (D, C, 9)]
current vectors : [(A, E, 6), (E, D, 4), (D, C, 9), (C, B, 4)]
vectors  : [(A, E, 6), (E, D, 4), (D, C, 9), (C, B, 4), (B, A, 9)]
path  : A->E->D->C->B->A
Weight : 32
Duration : 414 microseconds
```