



## **PROJET INTELLIGENCE ARTIFICIELLE M1 MIAGE**

*SEBBAN Samuel  
SARI Mohand ou Kaci*

# **Introduction**

## Processus de réflexion:

Le problème principal de l'algorithme  $A^*$  est de trouver une heuristique consistante qui fait converger l'algorithme.

La consistance de l'heuristique  $h$  assure la convergence de  $A^*$ , tandis que l'admissibilité ne l'assure pas .

En effet on peut aisément trouver un exemple de non-convergence avec l'heuristique nulle  $h=0$ .

L'objectif est donc de trouver une borne inférieure.

On a d'abord, comme suggéré dans le projet, utilisé le Minimum-Spanning-Tree (MST) des villes non visitées qui a la bonne propriété de ne pas surestimer le coût réel mais dans ce cas on n'a pas de convergence.

On a donc décidé de connecter le graphe des villes non visitées avec la ville du nœud  $n$  en question et aussi à la ville initiale comme suggéré dans le projet et on a obtenu la consistance.

Pour la recherche locale, nous avons opté pour Hill-climbing, codant une fonction qui échange deux sommets dans le graphe comme suggéré dans l'énoncé.

Le nombre d'itérations est quant à lui laissé en paramètre.

## PARTIE I: RECHERCHE INFORME

### Algorithme de l'arbre couvrant minimal (PRIM):

**Nom de la fonction:** MST(numpy array, numpy array, dataframe)

**INPUT:** Liste de noeuds dans l'arbre (numpy array), Liste de noeuds en dehors de l'arbre (numpy array) et Matrice des distances (dataframe)

**OUTPUT:** poids de l'arbre couvrant minimal (integer)

- A partir d'une liste de noeuds déjà dans l'arbre on déroule l'algorithme PRIM avec les noeuds en dehors de l'arbre:  
Pour chaque sommet en dehors de l'arbre, on prend l'arête de coût minimale reliant aux sommets déjà dans l'arbre.

exemple: avec l'arbre ABCDE de l'énoncé,

MST(ABC, DE) = 6 + 1 = 7 car A-E (1) et C-D (6) sont minimales dans ABC

### Algorithme A\* :

**Nom de la fonction :** a\_star(integer, dataframe)

**INPUT:** ville initiale (numéro de 0 à n-1 si n est le nombre de villes), matrice des distances

**OUTPUT:** retourne le circuit de plus court chemin et le coût associé

- A partir d'un noeud initial et d'une matrice de distances représentant le graphe on déroule l'algorithme A\*
- A partir d'un nœud appelé n représentant une ville, **ACTIONS(n)** désigne toutes les possibilités de navigation dans l'arbre parmi les villes possibles, c'est donc un chemin parmi les villes non visitées.  
**ACTIONS(n)** : Aller de n à n', pour n' appartenant à la liste ouverte de villes non visitées
- **RESULT(n, action(n))** désigne une ville parmi celles non visitées.  
**RESULT(n, action(n,n'))** : n'
- On note que la ville initiale est une ville non visitée car on doit terminer le chemin par la ville initiale mais ne peut être généré qu'à la fin.
- **Déroulement de A\***: On commence au noeud initial et on va générer ses voisins, c'est-à-dire les villes non encore visitées, on calcule  $f=g+h$  pour chaque voisin et on choisit le nœud qui minimise f parmi TOUS les noeuds qui n'ont pas été visités

On réitère l'opération avec le noeud choisi et on avance dans l'arbre jusqu'à ce qu'il reste qu'un noeud car lorsqu'il reste qu'un noeud on n'a pas le choix le coût est alors le chemin vers ce noeud à l'origine.

### Heuristique:

A chaque étape de l'algorithme étant donné un sommet S, on déploie ses voisins et on doit calculer pour chaque nœud n enfants de S,  **$g(n)$**  et  **$h(n)$** .

**$g(n)$**  : est donné par la matrice de distance, c'est le coût du chemin parcouru dans l'algorithme A\* jusqu'au noeud n.

**$h(n)$**  l'heuristique, en reprenant les notations du MST plus haut, l'heuristique est calculée de la façon suivante:

- **$h(n) = MST(E) + MST(E, 0) + MST(E, n)$**   
où  
**E: ensemble des villes non encore visitées en excluant la ville n du noeud en question déroulé**  
**0: ville initiale du parcours**  
**n: ville associée au noeud n**

**L'Heuristique ainsi définie est consistante (et donc aussi cohérente).**

### Preuve :

on montre que  **$h(n) \leq c(n, a, n') + h(n')$**  pour  $n'$  successeur de n  
 **$c(n, a, n')$**  est le coût de n à  $n'$  dans le graph non orienté.

pour  $n'$  successeur de n:

$$h(n) = MST(E) + MST(E, 0) + MST(E, n)$$

$$h(n') = MST(E/\{n'\}) + MST(E/\{n'\}, 0) + MST(E/\{n'\}, n')$$

$$\text{or } MST(E, 0) \leq MST(E/\{n'\}, 0)$$

et remarquons que  $MST(E, n) \leq c(n, a, n')$  car  $n'$  appartient à E

donc

$$h(n) - h(n') \leq MST(E) - MST(E/\{n'\}) - MST(E/\{n'\}, n') + c(n, a, n')$$

mais  $MST(E/\{n'\}) + MST(E/\{n'\}, n')$  est toujours plus grand que  $MST(E)$  car  $MST(E)$  est le poids de l'arbre minimal de E mais  $MST(E/\{n'\}) + MST(E/\{n'\}, n')$  est une manière 'forcée' de calculer l'arbre minimal de E non de manière non optimal.

$$\text{Ainsi } MST(E) \leq MST(E/\{n'\}) + MST(E/\{n'\}, n')$$

$$\text{et donc } h(n) - h(n') \leq c(n, a, n')$$

Ce qui termine la preuve.

**Exemple:** ville initiale A, voisins de A = B,C,D,E,  
noeud n=B

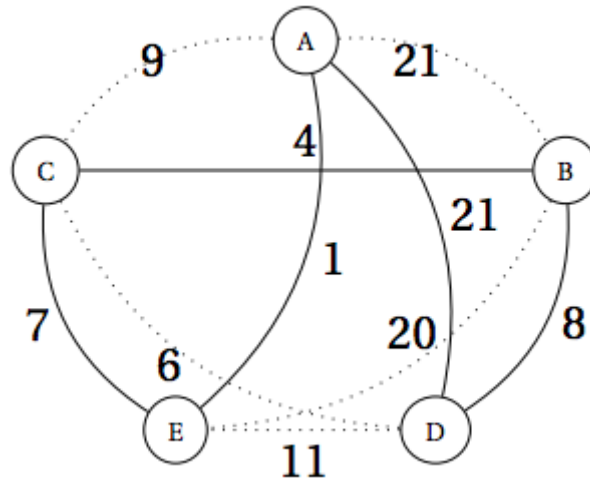
$$h(n) = \text{MST}(\{C,D,E\}) + \text{MST}(\{C,D,E\}, \{A\}) + \text{MST}(\{C,D,E\}, \{B\})$$

Par définition, l'heuristique MST donne le coût minimum du graphe qu'il traite, même si ce coût n'est pas réalisable en réalité, donc ce coût est toujours plus petit que le coût réel le plus bas, il ne surestime pas le coût.

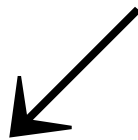
De plus, on sait que d'après la définition du cours une heuristique consistante est admissible.

**Donc l'heuristique MST est admissible.**

# EXEMPLE DE DÉROULEMENT DE A\* SUR UNE ITÉRATION AVEC LE GRAPH DE L'ÉNONCÉ



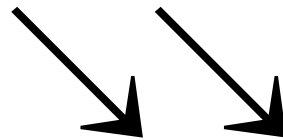
A g=0 h=0



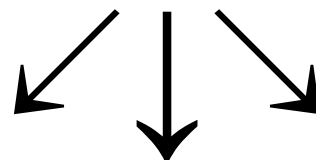
B g=21 h=18  
f=39



C g=9 h=24  
f=33



D g=21 h=18 f=39    E g=1, h=26  
f=27



B g=21 h=19  
f=40

C g=8 h=33  
f=41

D g=12 h=19  
f=31

Calcul de l'heuristique:

**On a  $h(n) = MST(E) + MST(E, 0) + MST(E, n)$**

**E: ensemble des villes non encore visitées en excluant la ville n du nœud en question déroulé.**

**0: ville initiale du parcours**

**n: ville associée au nœud n**

**1ère étape: On déploie tous les nœuds en partant de la ville A.**

$$\begin{aligned}h(\mathbf{B}) &= MST(CDE) + MST(CDE, \mathbf{B}) + MST(CDE, A) \\&= 13 + 4 + 1 \\&= 18\end{aligned}$$

$$\begin{aligned}h(\mathbf{C}) &= MST(BDE) + MST(BDE, \mathbf{C}) + MST(CDE, A) \\&= 19 + 4 + 1 \\&= 24\end{aligned}$$

Ici E est choisi.

**Etape 2: On choisit alors E car son f est le minimum et on déploie ensuite les sommets B,C,D.**

$$\begin{aligned}h(\mathbf{B}) &= MST(CD) + MST(CD, \mathbf{B}) + MST(CD, A) \\&= 6 + 4 + 9 \\&= 19\end{aligned}$$

On fait pareil pour les sommets C et D.

On choisit alors de déployer D (car  $31 < 33 < 39 < 40 < 41$ ).

Et ainsi de suite ...Jusqu'à que l'on arrive à l'état but où notre f est minimum.

## PARTIE II : RECHERCHE LOCALE

### Algorithme de recherche locale utilisé: Hill climbing

#### Fonctions auxiliaires de hill climbing:

##### **path\_cost:**

INPUT: circuit Hamiltonien

OUTPUT: coût du circuit

- l'algorithme utilise la matrice des distances pour calculer le coût

##### **2\_opt:**

INPUT: circuit Hamiltonien

OUTPUT: circuit Hamiltonien dans lequel 2 sommets sont échangés aléatoirement

- l'algorithme sélectionne un élément  $x_i$  dans le circuit
- l'algorithme sélectionne un élément  $x_j$  dans le circuit différent de  $x_{i-1}$ ,  $x_i$  et  $x_{i+1}$
- ensuite on change le noeud  $x_{i+1}$  par le noeud  $x_j$  et inversement

##### **hill climbing:**

INPUT: ville initiale, matrice de distances, nb\_itération

OUTPUT: circuit, cout, taux d'amélioration

- l'algorithme génère aléatoirement un chemin initial
- ensuite à chaque itération, on utilise l'algorithme **2\_opt** pour générer un nouveau chemin
- on compare le coût de ce nouveau chemin au chemin initial avec **path\_cost** et on le nouveau nouveau est gardé jusqu'à avoir atteint nb\_itération

#### Explication de non-convergence et de convergence locale dans Hill climbing:

Dans Hill climbing, on génère un circuit aléatoirement et à chaque étape on change 2 villes deux à deux de places dans le circuit, on regarde si le coût de ce nouveau circuit est plus faible que l'ancien, si c'est le cas le circuit est changé, sinon on recommence jusqu'à un nombre d'itérations souhaités.

Ainsi, il est possible que Hill climbing "bloque" sur une solution non optimale et n'arrive jamais à la solution du problème.

En effet, si le changement de deux villes mène toujours à un circuit de coût plus élevé que le circuit courant on n'aura jamais de changement donc jamais le circuit de coût optimal.

C'est donc un minimum local.



## Génération de graphes:

### **generate\_graph:**

INPUT: nb\_cities (integer), max\_distance(integer)

OUTPUT: pandas dataframe de taille nb\_citiesxnb\_cities

L'algorithme pour générer le graph est **generate\_graph**.

Tout d'abord on génère des graphes de la manière suivante:

On choisit un nombre de villes N et une distance maximum entre les villes.

Ensuite on génère NxN nombre aléatoires qui représentent les distances et on les met sous forme de matrice (matrice NxN).

On rend cette matrice triangulaire inférieure et on l'additionne avec la transposée de son inverse.

Enfin on ajoute des zéros le long de la diagonale.

Les villes sont représentées par des nombres de 0 à N-1 et le croisement dans la matrice la distance entre elles.

**\*Au lieu de générer des points aléatoires puis de calculer leur distance euclidienne, nous avons décidé de générer directement une matrice qui donne les distances aléatoires entre chaque ville.**

### Exécution des algorithmes sous python (best practices):

#### algorithme A\*:

```
matrix = generate_graph(nb_cities=10, max_distance=25)
```

```
init_city=0
```

```
ans = a_star(init_city=0, matrix=matrix)
```

```
ans[0] est le circuit
```

```
ans[1] est le coût associé
```

#### hill climbing:

```
matrix = generate_graph(nb_cities=10, max_distance=25)
```

```
init_city=0
```

```
it=1000
```

```
ans = hill_climbing(init_city=0, matrix=matrix, it=1000)
```

```
ans[0] est le circuit
```

```
ans[1] est le coût associé
```

```
ans[2] est le taux d'amélioration
```

Il faut compiler le code petit à petit (en utilisant la commande Run Selection or Current line), les indications et les différentes étapes de compilation sont données dans le fichier Python .

Voici un lien où il y a un record des étapes à suivre pour compiler :

<https://screencast-o-matic.com/watch/c3VDYGVogqE>

## RÉSULTATS :

Avec le graphe de l'énoncé du projet (5 villes), avec A\* on obtient le chemin:

**chemin trouvé :** A E D B C

**coût associé:** 33

Sur cet exemple: (sommet: A,B,C,D; arêtes: AB 1 AC 3 AD 3 BD 2 BC 3 CD 50)

**chemin trouvé:** A C B D

**coût associé:** 13

### Exécution de A\* avec le graphe de l'énoncé (5 villes)

```
A STAR ALGORITHM *
Node:  1 f:  39
Node:  2 f:  33
Node:  3 f:  39
Node:  4 f:  27
frontier:  [[1, 39], [2, 33], [3, 39], [4, 27]]
Explored cities:  [0 4]
Node:  1 f:  40
Node:  2 f:  41
Node:  3 f:  31
frontier:  [[1, 39], [2, 33], [3, 39], [1, 40], [2, 41], [3, 31]]
Explored cities:  [0 4 3]
Node:  1 f:  33
Node:  2 f:  43
frontier:  [[1, 39], [2, 33], [3, 39], [1, 40], [2, 41], [1, 33], [2, 43]]
Explored cities:  [0 4 3 1]
TIME FOR A* STAR --- 0.021250009536743164 seconds ---
**** RESULT FOR A* ALGORITHM ****
Optimal path:  [0 4 3 1 2 0]
Optimal cost is:  33
```

### Exécution de Hill Climbing avec le graphe de l'énoncé (5 villes) et 100 itérations

```
>>> ans = hill_climbing(init_city=0, matrix=dd, it=100)
[0 2 1 3 4 0]
Amelioration: 11
[0 2 3 1 4 0]
No Amelioration
[0 2 3 1 4 0]
No Amelioration
[0 2 3 1 4 0]
No Amelioration
[0 2 3 1 4 0]
No Amelioration
[0 2 4 3 1 0]
No Amelioration
[0 2 1 4 3 0]
No Amelioration
[0 2 3 1 4 0]
```

On affiche ici le nouveau chemin trouvé, si il est meilleur que le précédent on affiche l'amélioration de coût obtenu sinon on affiche "No amelioration".

Ici l'algorithme a trouvé un chemin optimal A B D E.

### Comparaison entre A\* et Hill climbing:

```
TIME FOR HILL CLIMBING --- 0.6690030097961426 seconds ---
**** RESULT FOR HILL CLIMBING ****
Final path: [0 2 1 3 4 0]
Cost is: 33
Amelioration % from initial circuit: 50 %
Distance to optimum: 0
```

On compare comme demandé le distance à l'optimum: c'est le coût du chemin trouvé dans hill climbing moins le coût optimale trouvé avec A\*.

### TABLE POUR L'EXECUTION DE A\*

	Nb de ville	Coût	Frontier max	Temps	Nb Iteration
0	6	48	10	0	4
1	7	41	15	0	5
2	8	36	21	0	6
3	9	37	28	0	7
4	10	34	36	0	8
5	11	37	45	0	9
6	12	37	55	0	10
7	13	44	66	0	11
8	14	44	78	0	12
9	15	39	91	0	13
10	16	41	105	0	14
11	17	44	120	0	15
12	18	36	136	1	16
13	19	37	153	1	17
14	20	39	171	1	18
15	21	36	190	2	19
16	22	31	210	3	20
17	23	41	231	3	21
18	24	42	253	4	22

Chaque ligne représente les résultats de A\* en fonction d'une génération d'un graph comportant le nombre de ville passé en paramètre.

### TABLE POUR L'EXÉCUTION DE HILL CLIMBING:

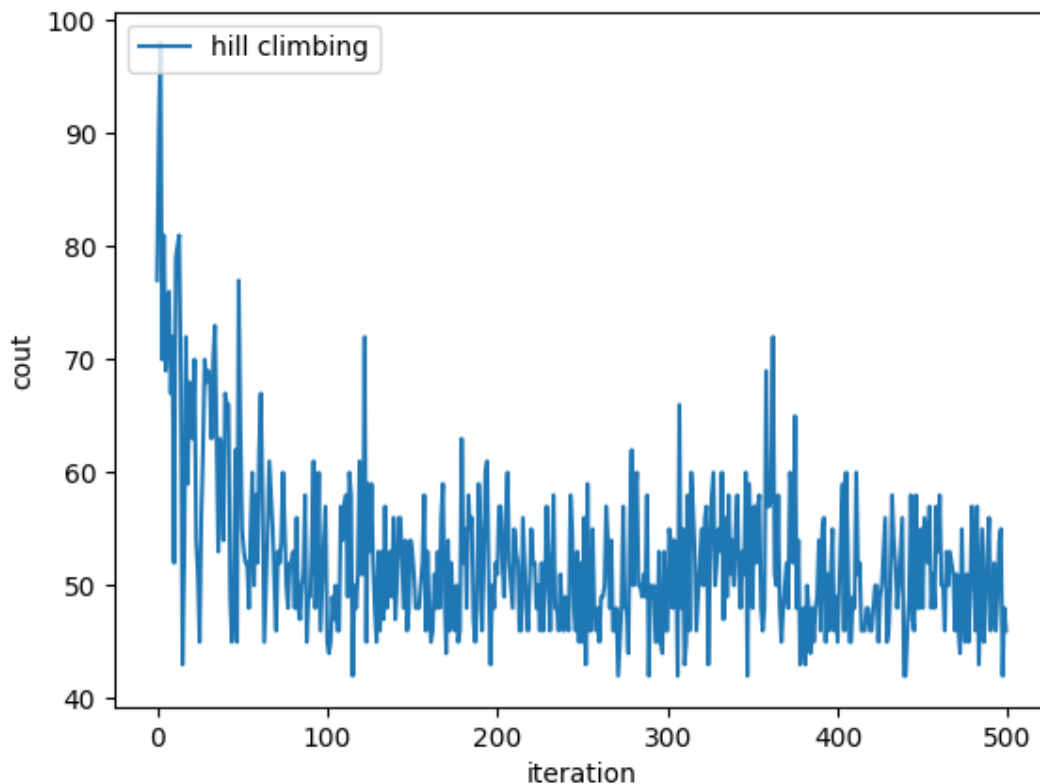
	Nb de ville	Coût	Amélioration %	Temps	Nb Iteration
0	6	26	100	0	100
1	7	24	32	0	100
2	8	46	83	0	100
3	9	31	45	0	100
4	10	46	41	0	100
5	11	42	36	0	100
6	12	36	33	0	100
7	13	31	21	0	100
8	14	44	35	0	100
9	15	57	37	0	100
10	16	55	33	0	100
11	17	81	66	0	100
12	18	80	48	0	100
13	19	59	36	0	100
14	20	106	66	0	100
15	21	89	43	0	100
16	22	78	48	0	100
17	23	87	48	0	100
18	24	120	51	0	100

### Taille du graphe que l'on peut gérer :

- L'algorithme A\* peut gérer un graphe d'au moins 2 villes car si on choisit 1 ville alors l'open list contenant les villes non encore visitées est vide et on ne peut pas revenir à la ville initiale, nous avons choisi de commencer à 2 villes car un graphe avec 1 seule ville est trivial.
- l'algorithme hill-climbing peut gérer au moins 5 villes car le processus d'échanges 2-opt nécessite au moins deux arêtes à échanger, sur 4 villes [0 1 2 0] on ne peut pas appliquer l'algorithme.

### Calibrage des algorithmes étudiés:

#### Itération de Hill Climbing en fonction du coût minimum trouvé:



Ici on peut voir que l'algorithme ne s'améliore pas après 20 itérations.

On peut donc calibrer le nombre d'itérations optimales à l'aide de ce graphe pour chaque nombre de villes choisies.

Pour A\*, aucun calibrage n'est requis car le seul paramètre de l'algorithme est le calcul de l'heuristique.

### CONCLUSION:

L'algorithme A\* a l'avantage de donner toujours la solution optimale mais lorsque le nombre de ville grandit, il devient lent car il doit calculer g et h pour tous les nœuds restants à chaque étape.

C'est alors ici que Hill-Climbing intervient: bien qu'il ne converge pas tout le temps vers une solution optimale, on peut en un temps raisonnable obtenir une solution convenable, c'est-à-dire un coût de trajet relativement bas par rapport à une solution donnée au hasard.

Le choix entre les deux algorithmes se fait en fonction des problématiques de l'utilisateur (nombre de villes, temps d'attente, optimalité de la solution).

## RÉFÉRENCES :

[https://classes.engr.oregonstate.edu/mime/fall2017/rob537/hw\\_samples/hw2\\_sample2.pdf](https://classes.engr.oregonstate.edu/mime/fall2017/rob537/hw_samples/hw2_sample2.pdf)

<https://fr.wikipedia.org/wiki/2-opt>

## ANNEXE :

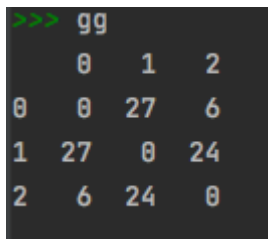
On déroule A\* sur un petit exemple (n=3 villes) :

### **Exemple avec 3 villes:**

code:

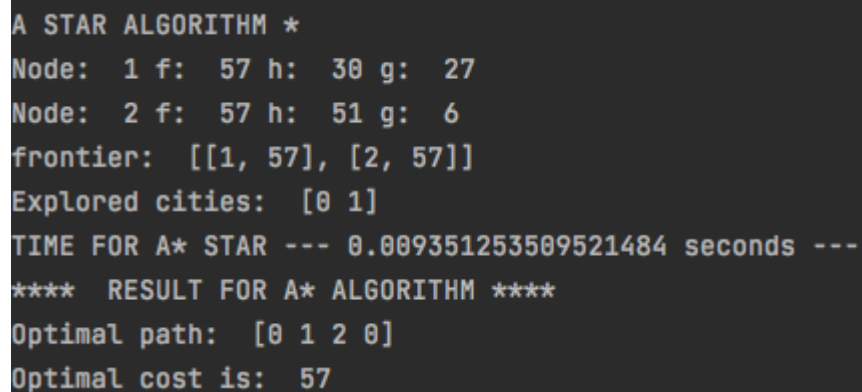
```
# graph generation
gg = generate_graph(nb_cities=3, max_distance=30)
# ##### A STAR #####
ans_a = a_star(init_city=0, matrix=gg)
opt_path = ans_a[0]
opt_cost = ans_a[1]
print("**** RESULT FOR A* ALGORITHM ****")
print("Optimal path: ", opt_path)
print("Optimal cost is: ", opt_cost)
```

Graphe représentant les villes:



	0	1	2
0	0	27	6
1	27	0	24
2	6	24	0

Déroulement de A\*:



```
A STAR ALGORITHM *
Node:  1 f:  57 h:  30 g:  27
Node:  2 f:  57 h:  51 g:   6
frontier:  [[1, 57], [2, 57]]
Explored cities:  [0 1]
TIME FOR A* STAR --- 0.009351253509521484 seconds ---
**** RESULT FOR A* ALGORITHM ****
Optimal path:  [0 1 2 0]
Optimal cost is:  57
```