LA Joséphine

SITBON Benjamin

Read Me

I)Analyse du problème.

L'objectif de ce projet est de résoudre des instances du problème du voyageur de commerce, et de trouver un plus court chemin qui visite chaque ville exactement une fois et qui se termine dans la ville initiale. Nous avons modélisé le problème sous forme de graphe non orienté, où les arrêtes indiquent un lien et la distance entre deux villes.

Pour la mise en place du graphe, nous avons créé deux classes.

La première classe est la classe Vertex, possédant toutes les méthodes permettant de caractériser le point d'un graphe.

La deuxième classe est la classe Graphe. Nous y avons défini différentes méthodes nous permettant de rajouter des sommets dans notre graphe, et des liens qui peuvent relier les villes les unes aux autres. Afin de représenter notre graphe, nous avons choisi d'utiliser une matrice symétrique. Soit A notre matrice, alors Aij représente la distance qu'il y a à parcourir entre la ville i et la ville j. Du fait que le chemin est le même que l'on parte de i vers j ou de j vers i, Aij = Aji. Cette matrice nous a été très utilise car elle permettait de récupérer facilement le poids d'une arête.

Les états possibles sont les villes i avec i pour i allant de 0 à n-1, avec n le nombre de villes.

ACTIONS(i) = {Go j} pour j allant de 0 à n tel que i ≠ j if and only if j is not visited

RESULTS(i, Go j) = {j}

Nous avons considéré qu'un état initial est un chemin que l'on a pris de manière aléatoire.

Le résultat final est un chemin dont le coût est la valeur minimale.

Nous avons utilisé les bibliothèques suivantes : random, cys et time.

II)Recherche informée.

A) Code

Subgraph() : cette fonction, prenant en argument un graphe G et une liste de sommet L, permet de créer un nouveau graphe G'. Si certains sommets de G sont présents dans la liste L, cette fonction renverra un nouveau graphe qui ne contiendra que les sommets non présents dans la liste L. Ainsi, nous pourrons utiliser l'heuristique recommandé c'est-à-dire que l'on fera le Minimum Spanning Tree des villes non visitées et de la ville associée au nœud n.

mst() : Cet algorithme est un algorithme glouton qui calcule un arbre couvrant minimal dans un graphe connexe valué et non orienté. Initialement un arbre avec un seul sommet, l'arbre s'agrandira à chaque itération en prenant l'arête de coût minimum reliant un sommet de l'arbre à un sommet hors de l'arbre.

Pour implémenter cette fonction, il fallait d'abord créer un tableau permettant d'indiquer si un sommet a déjà été visité ou non. Pour chaque sommet visité, on étend une recherche à tous ses sommets voisins et on stock dans une variable cette valeur minimale. Une fois que tous les sommets sont visités, et en sommant les arêtes de cet arbre, on obtient un arbre couvrant minimal.

Add_frontier(): Cette fonction nous permettra d'ajouter un élément à la frontière en le triant. Si la somme f = g + h (g représente le coût et h l'heuristique) est plus petite alors elle se plaçera avant. Si deux sommes sont égales, alors comme le cours le précise, nous visiterons la ville avec l'heuristique la moins élevée.

Astar(): Cette algorithme permet de faire tourner l'algorithme Astar que nous avons étudié. Elle part du nœud demandé et calcule son f. On ajoutera le 'f' de chaque sommet et on l'ajoutera à la frontière. Celui qui a la valeur la moins élevée sera le sommet que l'on souhaite développer sera le premier élément de notre frontière grâce à la fonction **add_frontier**().

Tant que le premier élément de notre frontière n'est pas une chemin complet, c'est-à-dire qu'il visite tous les sommets exactement une fois et qu'il revient a la ville de départ, nous continuons de développer les chemins.

B) Heuristique

L'heuristique proposé dans l'énoncé du sujet est admissible et cohérente. En effet, elle est admissible car elle ne surestimera jamais le coût. Supposons que nous soyons dans la ville A, il nous reste les villes B,C,D,E à visiter. Nous obtenons Minimum Spanning Tree suivant :

Certains Minimum Spanning Tree de donneront pas de chemin, nous pouvons relaxer l'énoncé en ne comptant pas le chemin retour si nous arrivons au bout d'un chemin. Ainsi en sommant les arcs nous obtenons une heuristique admissible.

C) Résultats

- 5 villes

Nous retournons le chemin à emprunter, son coût, ainsi que la taille maximale de notre frontière. Pour l'exemple de l'énoncé, nous avons affiché la frontière pour montrer le détail de notre algorithme. Pour des soucis d'affichage, nous vous avons copié/collé la frontière.

```
astar runs in :  0.0 s
Le chemin le plus court est : [0, 4, 3, 1, 2, 0]
Le cout du chemin est : 33
La taille maximale de la frontière est de : 10
```

Nous sommes à la 0 ème itération(s) voici la frontière :

[[0, [], 18, 0]]

Nous sommes à la 1 ème itération(s) voici la frontière :

[[4, [0], 18, 1], [2, [0], 18, 9], [3, [0], 18, 21], [1, [0], 18, 21]]

Nous sommes à la 2 ème itération(s) voici la frontière :

[[2, [0, 4], 17, 8], [2, [0], 18, 9], [3, [0, 4], 17, 12], [1, [0, 4], 17, 21], [3, [0], 18, 21], [1, [0], 18, 21]]

Nous sommes à la 3 ème itération(s) voici la frontière :

[[1, [0, 4, 2], 10, 12], [3, [0, 4, 2], 10, 14], [2, [0], 18, 9], [3, [0, 4], 17, 12], [1, [0, 4], 17, 21], [3, [0], 18, 21], [1, [0], 18, 21]]

Nous sommes à la 4 ème itération(s) voici la frontière :

[[3, [0, 4, 2], 10, 14], [2, [0], 18, 9], [3, [0, 4, 2, 1], 8, 20], [3, [0, 4], 17, 12], [1, [0, 4], 17, 21], [3, [0], 18, 21], [1, [0], 18, 21]]

Nous sommes à la 5 ème itération(s) voici la frontière :

[[2, [0], 18, 9], [3, [0, 4, 2, 1], 8, 20], [3, [0, 4], 17, 12], [1, [0, 4, 2, 3], 8, 22], [1, [0, 4], 17, 21], [3, [0], 18, 21], [1, [0], 18, 21]]

Nous sommes à la 6 ème itération(s) voici la frontière :

[[1, [0, 2], 4, 13], [3, [0, 4, 2, 1], 8, 20], [3, [0, 4], 17, 12], [1, [0, 4, 2, 3], 8, 22], [3, [0, 2], 17, 15], [4, [0, 2], 17, 16], [1, [0, 4], 17, 21], [3, [0], 18, 21], [1, [0], 18, 21]]

Nous sommes à la 7 ème itération(s) voici la frontière :

[[3, [0, 4, 2, 1], 8, 20], [3, [0, 4], 17, 12], [1, [0, 4, 2, 3], 8, 22], [3, [0, 2], 17, 15], [4, [0, 2], 17, 16], [1, [0, 4], 17, 21], [3, [0], 18, 21], [1, [0], 18, 21], [3, [0, 2, 1], 19, 21], [4, [0, 2, 1], 19, 33]]

Nous sommes à la 8 ème itération(s) voici la frontière :

[[3, [0, 4], 17, 12], [1, [0, 4, 2, 3], 8, 22], [3, [0, 2], 17, 15], [4, [0, 2], 17, 16], [1, [0, 4], 17, 21], [3, [0], 18, 21], [1, [0], 18, 21], [3, [0, 2, 1], 19, 21], [4, [0, 2, 1], 19, 33]]

Nous sommes à la 9 ème itération(s) voici la frontière :

[[2, [0, 4, 3], 10, 18], [1, [0, 4, 2, 3], 8, 22], [1, [0, 4, 3], 10, 20], [3, [0, 2], 17, 15], [4, [0, 2], 17, 16], [1, [0, 4], 17, 21], [3, [0], 18, 21], [1, [0], 18, 21], [3, [0, 2, 1], 19, 21], [4, [0, 2, 1], 19, 33]]

Nous sommes à la 10 ème itération(s) voici la frontière :

[[1, [0, 4, 3, 2], 4, 22], [1, [0, 4, 2, 3], 8, 22], [1, [0, 4, 3], 10, 20], [3, [0, 2], 17, 15], [4, [0, 2], 17, 16], [1, [0, 4], 17, 21], [3, [0], 18, 21], [1, [0], 18, 21], [3, [0, 2, 1], 19, 21], [4, [0, 2, 1], 19, 33]]

Nous sommes à la 11 ème itération(s) voici la frontière :

[[1, [0, 4, 2, 3], 8, 22], [1, [0, 4, 3], 10, 20], [3, [0, 2], 17, 15], [4, [0, 2], 17, 16], [1, [0, 4], 17, 21], [3, [0], 18, 21], [1, [0], 18, 21], [3, [0, 2, 1], 19, 21], [1, [0, 4, 3, 2, 1, 0], 0, 43], [4, [0, 2, 1], 19, 33]]

Nous sommes à la 12 ème itération(s) voici la frontière :

[[1, [0, 4, 3], 10, 20], [3, [0, 2], 17, 15], [4, [0, 2], 17, 16], [1, [0, 4], 17, 21], [3, [0], 18, 21], [1, [0], 18, 21], [3, [0, 2, 1], 19, 21], [1, [0, 4, 3, 2, 1, 0], 0, 43], [4, [0, 2, 1], 19, 33]]

Nous sommes à la 13 ème itération(s) voici la frontière :

[[2, [0, 4, 3, 1], 4, 24], [3, [0, 2], 17, 15], [4, [0, 2], 17, 16], [1, [0, 4], 17, 21], [3, [0], 18, 21], [1, [0], 18, 21], [3, [0, 2, 1], 19, 21], [1, [0, 4, 3, 2, 1, 0], 0, 43], [4, [0, 2, 1], 19, 33]]

Nous sommes à la 14 ème itération(s) voici la frontière :

[[3, [0, 2], 17, 15], [2, [0, 4, 3, 1, 2, 0], 0, 33], [4, [0, 2], 17, 16], [1, [0, 4], 17, 21], [3, [0], 18, 21], [1, [0], 18, 21], [3, [0, 2, 1], 19, 21], [1, [0, 4, 3, 2, 1, 0], 0, 43], [4, [0, 2, 1], 19, 33]]

- 10 villes

```
Voici la matrice présentant les coûts entre chaque ville
row 0 : [0, 20, 9, 12, 5, 8, 10, 13, 17, 12]
row 1 : [20, 0, 20, 20, 1, 8, 17, 7, 16, 8]
row 2 : [9, 20, 0, 6, 8, 16, 13, 16, 1, 3]
row 3 : [12, 20, 6, 0, 20, 12, 6, 16, 20, 16]
row 4 : [5, 1, 8, 20, 0, 20, 9, 6, 13, 12]
row 5 : [8, 8, 16, 12, 20, 0, 16, 19, 3, 3]
row 6 : [10, 17, 13, 6, 9, 16, 0, 6, 16, 11]
row 7 : [13, 7, 16, 16, 6, 19, 6, 0, 18, 16]
row 8 : [17, 16, 1, 20, 13, 3, 16, 18, 0, 5]
row 9 : [12, 8, 3, 16, 12, 3, 11, 16, 5, 0]
```

```
astar runs in :  0.05 s
Le chemin le plus court est : [0, 4, 1, 7, 6, 3, 2, 8, 9, 5, 0]
Le cout du chemin est : 48
La taille maximale de la frontière est de : 238
```

- 15 villes

Plus le nombre de ville est élevé, plus il est recommandé d'utiliser des distances plus précises et donc plus élevées pour éviter les cas d'égalité d'heuristique et de coût.

```
Voici la matrice présentant les coûts entre chaque ville
row 0 : [0, 83, 74, 6, 97, 51, 37, 63, 63, 29, 24, 63, 43, 11, 73]
row 1 : [83, 0, 97, 99, 53, 37, 98, 44, 13, 37, 75, 17, 88, 8, 17]
row 2 : [74, 97, 0, 87, 85, 89, 31, 97, 57, 87, 34, 15, 77, 60, 19]
row 3 : [6, 99, 87, 0, 39, 75, 13, 54, 53, 28, 37, 76, 62, 89, 4]
row 4 : [97, 53, 85, 39, 0, 59, 1, 29, 29, 19, 68, 29, 88, 66, 32]
row 5 : [51, 37, 89, 75, 59, 0, 66, 6, 9, 68, 57, 35, 100, 13, 33]
row 6 : [37, 98, 31, 13, 1, 66, 0, 44, 68, 43, 51, 42, 19, 70, 96]
row 7 : [63, 44, 97, 54, 29, 6, 44, 0, 53, 20, 9, 71, 43, 88, 30]
row 8 : [63, 13, 57, 53, 29, 9, 68, 53, 0, 39, 71, 19, 77, 92, 77]
row 9 : [29, 37, 87, 28, 19, 68, 43, 20, 39, 0, 75, 43, 10, 51, 65]
row 10 : [24, 75, 34, 37, 68, 57, 51, 9, 71, 75, 0, 36, 62, 12, 6]
row 11 : [63, 17, 15, 76, 29, 35, 42, 71, 19, 43, 36, 0, 19, 94, 49]
row 12 : [43, 88, 77, 62, 88, 100, 19, 43, 77, 10, 62, 19, 0, 10, 69]
row 13 : [11, 8, 60, 89, 66, 13, 70, 88, 92, 51, 12, 94, 10, 0, 99]
row 14 : [73, 17, 19, 4, 32, 33, 96, 30, 77, 65, 6, 49, 69, 99, 0]
```

```
astar runs in :  0.98 s
Le chemin le plus court est : [0, 3, 6, 4, 9, 12, 11, 2, 14, 10, 7, 5, 8, 1, 13, 0]
Le cout du chemin est : 164
La taille maximale de la frontière est de : 3080
```

- 17 villes

```
Voici la matrice présentant les coûts entre chaque ville
row 0 : [0, 8, 11, 97, 6, 83, 79, 53, 77, 9, 49, 1, 11, 69, 64, 89, 22]
row 1 : [8, 0, 1, 26, 53, 83, 65, 30, 52, 32, 57, 91, 77, 38, 90, 38, 38]
row 2 : [11, 1, 0, 5, 14, 22, 91, 45, 91, 86, 33, 48, 66, 74, 24, 79, 97]
row 3 : [97, 26, 5, 0, 27, 59, 91, 88, 92, 75, 49, 34, 23, 4, 34, 62, 2]
row 4 : [6, 53, 14, 27, 0, 58, 34, 74, 95, 91, 99, 36, 75, 94, 58, 26, 62]
row 5 : [83, 83, 22, 59, 58, 0, 30, 55, 50, 54, 66, 80, 80, 68, 75, 52, 56]
row 6 : [79, 65, 91, 91, 34, 30, 0, 69, 99, 65, 9, 93, 87, 32, 37, 36, 100]
row 7 : [53, 30, 45, 88, 74, 55, 69, 0, 93, 11, 64, 91, 15, 41, 33, 44, 89]
row 8 : [77, 52, 91, 92, 95, 50, 99, 93, 0, 90, 92, 92, 14, 33, 21, 53, 19]
row 9 : [9, 32, 86, 75, 91, 54, 65, 11, 90, 0, 71, 9, 16, 11, 43, 91, 62]
row 10 : [49, 57, 33, 49, 99, 66, 9, 64, 92, 71, 0, 21, 56, 83, 42, 98, 100]
row 11 : [1, 91, 48, 34, 36, 80, 93, 91, 92, 9, 21, 0, 18, 46, 60, 37, 86]
row 12 : [11, 77, 66, 23, 75, 80, 87, 15, 14, 16, 56, 18, 0, 6, 13, 38, 92]
row 13 : [69, 38, 74, 4, 94, 68, 32, 41, 33, 11, 83, 46, 6, 0, 77, 28, 10]
row 14 : [64, 90, 24, 34, 58, 75, 37, 33, 21, 43, 42, 60, 13, 77, 0, 49, 68]
row 15 : [89, 38, 79, 62, 26, 52, 36, 44, 53, 91, 98, 37, 38, 28, 49, 0, 59]
row 16 : [22, 38, 97, 2, 62, 56, 100, 89, 19, 62, 100, 86, 92, 10, 68, 59, 0]
astar runs in :  232.43573665618896 s
Le chemin le plus court est : [0, 1, 2, 5, 6, 10, 11, 9, 7, 12, 14, 8, 16, 3, 13, 15, 4, 0]
Le cout du chemin est : 245
La taille maximale de la frontière est de : 50488
```

III)Recherche locale.

A)Le code

En ce qui concerne la recherche locale, nous avons décidé d'implémenter l'algorithme Hill-Climbing. Cette méthode consiste à partir d'une solution initiale puis de regarder chaque solution voisine, et de choisir la meilleure parmi ces solutions. À travers une boucle, cette opération est répétée jusqu'à ce que la meilleure solution parmi les voisins soit trouvée.

Dans notre code, nous avons créé une fonction hillClimbing qui prend en entrée le graphe représentant l'ensemble de nos villes. Cela est particulièrement utile car ca n'a permet de retrouver facilement la distance à parcourir entre deux villes, ainsi que pour calculer le coût de notre chemin, grâce à la fonction routeLength() qui prend en entrée le graphe ainsi que le chemin dont nous souhaitons connaitre le poids.

La première étape est de récupérer un chemin hamiltonien aléatoire grâce à la fonction randomSolution(). Ensuite nous avons utilisé currentRouteLenght() afin de connaître le poids de la somme des arêtes de ce chemin.

Il nous semblait logique qu'à partir de notre solution, si la ville i était la ville de départ, il nous faudrait alors trouver le meilleur chemin hamiltonien traversant les différentes villes tout en revenant à notre ville initiale i.

Pour définir le voisinage de chaque état, nous avons utilisé l'opération 2-opt.

```
for i in range(1,len(solution)-2):
        for j in range(i+1,len(solution)):
            if j-i == 1 : continue
            new_solution = solution[:]
            new_solution[i:j]=solution[j-1:i-1:-1]
            getNeigh.append(new_solution)
```

Cette implémentation permet de représenter le voisinage d'un chemin suivant l'opération 2-opt.

Supposons que notre chemin soit le suivant : `[4, 2, 3, 1, 0, 4]`

Le voisinage de ce chemin que retournera notre fonction sera donc le suivant :

```
[[4, 3, 2, 1, 0, 4], [4, 1, 3, 2, 0, 4], [4, 0, 1, 3, 2, 4], [4, 2, 1, 3, 0, 4], [4, 2, 0, 1, 3, 4], [4, 2, 3, 0, 1, 4]]
```

Cette liste de chemins correspond à la variable resultantNeigh dans notre fonction getNeighbors

Pour chacun de ses voisins, nous calculons le poids total, et nous gardons en mémoire lequel de ses chemins possède un coût plus petit que notre solution, à condition que notre chemin initial ne soit pas déjà le chemin ayant le plus petit coût. C'est particulièrement cette condition qui permettra de définir au final quel chemin choisir afin d'obtenir le coût minimum.

Ce morceau de code est la condition nous permettant de tester si la valeur du coût du nouveau voisin obtenu est plus petite que celle de notre valeur “minimum” actuelle.

```
if routeLength(g,new_solution)<routeLength(g,best):
    best = new_solution
    amelioration = True
```

La boucle ci-dessous permet de s’assurer que nous obtiendrons dans les variables currentSolution et currentRouteLenght le chemin hamiltonien dont le coût est le plus petit.

```
while(bestLenght<currentRouteLenght):
    currentSolution = bestNeigh
    currentRouteLenght= bestLenght
    bestNeigh,neighbours = getNeighbors(g.adjMatrix,currentSolution)
    bestLenght=routeLength(g.adjMatrix,bestNeigh)
```

Pour calculer le taux, nous avons stocké le coût de notre chemin initial et le coût de notre chemin final dans deux variables, puis nous avons effectué l’opération suivante :

```
taux = ((t_initial - t_final)/t_initial)*100
```

B)Résultats

En reprenant le graphe donné dans l’énoncé du projet, la matrice est la suivante :

```
row 0 : [0, 21, 9, 21, 1]
row 1 : [21, 0, 4, 8, 20]
row 2 : [9, 4, 0, 6, 7]
row 3 : [21, 8, 6, 0, 11]
row 4 : [1, 20, 7, 11, 0]
```

Notre code commence par générer un chemin hamiltonien aléatoire. Prenons en tant qu’exemple le chemin suivant : [4,0,2,3,1,4]

Ce chemin a initialement un coût de 44. Notre objectif est de trouver le chemin dont le coût est le plus petit. Pour cela, voici les voisins de ce chemin :

[[4, 2, 0, 3, 1, 4], [4, 3, 2, 0, 1, 4], [4, 1, 3, 2, 0, 4], [4, 0, 3, 2, 1, 4], [4, 0, 1, 3, 2, 4], [4, 0, 2, 1, 3, 4], [4, 2, 0, 1, 3, 4], [4, 1, 2, 0, 3, 4], [4, 3, 1, 2, 0, 4], [4, 0, 1, 2, 3, 4], [4, 0, 3, 1, 2, 4]]

Apres exécution de notre algorithme, le chemin [4, 0, 2, 1, 3, 4] a été désigné comme étant le plus intéressant, avec un coût de 33.

On voit obtient un taux d’amélioration de 25%. Le temps d’exécution est très rapide, seulement 0.0003 secondes.

Considérons un autre exemple. Décidons de choisir 25 villes, dont la distance maximale entre chaque ville est 30. Après une initialisation aléatoire de notre graphe, nous lancons notre algorithme. Le chemin initial est le suivant :

```
[9, 18, 10, 1, 24, 7, 11, 3, 5, 19, 6, 4, 22, 13, 0, 21, 20, 8, 15, 14, 17, 12, 2, 16, 23, 9]
```

Possédant un coût de 497, et tout en partant de la ville 9, on trouve notre objectif de plus court chemin avec le chemin suivant :

`[9, 24, 3, 14, 19, 20, 11, 8, 16, 22, 17, 12, 5, 2, 13, 15, 23, 4, 21, 18, 7, 10, 1, 6, 0, 9]`, pour un coût total de 68. Le temps d'exécution augmente légèrement, avec 3,79 secondes.

Cela reste normal car le nombre de ville est relativement petit.

Lorsque nous avons exécuté notre code sur une initialisation de 30 villes, un chemin final avec un coût de 150 (dont le coût initial était de 789) est trouvé en 10,10 secondes.

Pour le même nombre de ville, avec une distance de 20 , on trouve un plus court chemin en 17 secondes.

En effet, lorsque nous prenons un nombre de villes plus conséquents, le temps d'exécution est beaucoup plus long.

D'ailleurs, considérons l'exemple suivant :

De manière aléatoire, nous avons généré un graphe représentant 40 villes dont la distance maximum est 50.

La matrice est trop grande à afficher mais voici le chemin initial :

[33, 32, 2, 8, 18, 38, 3, 11, 5, 12, 36, 25, 28, 7, 4, 35, 0, 15, 22, 1, 29, 27, 31, 10, 34, 17, 14, 39, 20, 6, 30, 26, 21, 23, 24, 16, 19, 37, 9, 13, 33]

Ce chemin à un coût de 1085. Après exécution de l'algorithme, le chemin trouvé est le suivant :

[33, 24, 31, 12, 15, 13, 1, 20, 18, 4, 5, 28, 29, 32, 8, 35, 27, 19, 25, 2, 0, 14, 39, 17, 34, 6, 7, 16, 9, 38, 11, 37, 3, 21, 26, 30, 23, 22, 36, 10,33] avec un coût de 177. On a un taux d'amélioration de 83.69%, cependant le temps d'exécution est beaucoup plus long, il prend 145.38 secondes.

IV) Comparaison des algorithmes.

Taille de la matrice	Matrice adjacente	Temps de l'algorithme Astar	Temps de l'algorigme Hill Climbing

5	[0, 2, 17, 7, 3] [2, 0, 9, 2, 8] [17, 9, 0, 15, 13] [7, 2, 15, 0, 15] [3, 8, 13, 15, 0]	0.0s	0.0 s
7	[0, 2, 6, 6, 17, 1, 7] [2, 0, 17, 12, 19, 3, 20] [6, 17, 0, 7, 19, 11, 13] [6, 12, 7, 0, 16, 20, 1] [17, 19, 19, 16, 0, 1, 5] [1, 3, 11, 20, 1, 0, 17] [7, 20, 13, 1, 5, 17, 0]	0.0 s	0.0 secondes
10	[0, 30, 45, 25, 31, 17, 41, 36, 24, 43] [30, 0, 26, 42, 19, 31, 13, 28, 2, 41] [45, 26, 0, 4, 4, 13, 34, 33, 22, 29] [25, 42, 4, 0, 37, 49, 1, 6, 15, 41] [31, 19, 4, 37, 0, 40, 2, 30, 18, 28] [17, 31, 13, 49, 40, 0, 37, 17, 48, 50] [41, 13, 34, 1, 2, 37, 0, 34, 48, 6] [36, 28, 33, 6, 30, 17, 34, 0, 43, 9] [24, 2, 22, 15, 18, 48, 48, 43, 0, 22] [43, 41, 29, 41, 28, 50, 6, 9, 22, 0]	0.08 s	0.0055 secondes
12	[0, 37, 15, 2, 11, 55, 25, 68, 90, 13, 61, 24] [37, 0, 75, 86, 75, 66, 31, 62, 15, 4, 39, 69] [15, 75, 0, 94, 95, 36, 82, 64, 40, 9, 23, 16] [2, 86, 94, 0, 81, 11, 48, 20, 32, 86, 8, 10] [11, 75, 95, 81, 0, 94, 8, 49, 13, 66, 25, 68] [55, 66, 36, 11, 94, 0, 5, 71, 45, 46, 79, 76] [25, 31, 82, 48, 8, 5, 0, 66, 77, 58, 4, 56] [68, 62, 64, 20, 49, 71, 66, 0, 1, 2, 88, 43] [90, 15, 40, 32, 13, 45, 77, 1, 0, 100, 52, 35] [13, 4, 9, 86, 66, 46, 58, 2, 100, 0, 8, 54] [61, 39, 23, 8, 25, 79, 4, 88, 52, 8, 0, 53] [24, 69, 16, 10, 68, 76, 56, 43, 35, 54, 53, 0]	5.66 s	0.0238 secondes
15	row 0 : [0, 30, 47, 49, 37, 12, 38, 36, 23, 26, 10, 9, 44, 5, 9] row 1 : [30, 0, 11, 36, 21, 47, 16, 28, 46, 16, 10, 17, 42, 17, 16] row 2 : [47, 11, 0, 17, 29, 35, 9, 4, 21, 10, 43, 43, 46, 29, 37] row 3 : [49, 36, 17, 0, 23, 14, 37, 34, 7, 44, 40, 36, 2, 43, 35] row 4 : [37, 21, 29, 23, 0, 34, 45, 20, 8, 23, 48, 36, 37, 23, 44] row 5 : [12, 47, 35, 14, 34, 0, 20, 43, 41, 14, 5, 45, 33, 37, 13] row 6 : [38, 16, 9, 37, 45, 20, 0, 12, 23, 12, 12, 1, 45, 48, 37] row 7 : [36, 28, 4, 34, 20, 43, 12, 0, 29, 19, 44, 38, 23, 6, 11] row 8 : [23, 46, 21, 7, 8, 41, 23, 29, 0, 28, 23, 24, 6, 9, 49] row 9 : [26, 16, 10, 44, 23, 14, 12, 19, 28, 0, 50, 34, 30, 40, 36] row 10 : [10, 10, 43, 40, 48, 5, 12, 44, 23, 50, 0, 13, 49, 48, 43] row 11 : [9, 17, 43, 36, 36, 45, 1, 38, 24, 34, 13, 0, 1, 4, 16] row 12 : [44, 42, 46, 2, 37, 33, 45, 23, 6, 30, 49, 1, 0, 5, 41] row 13 : [5, 17, 29, 43, 23, 37, 48, 6, 9, 40, 48, 4, 5, 0, 25] row 14 : [9, 16, 37, 35, 44, 13, 37, 11, 49, 36, 43, 16, 41, 25, 0]	87.6185 s	0.0882 secondes

V)Exécution.

Pour lancer notre code, il suffit simplement de l'exécuter. La version de Python utilisée est Python 3.9.1

Pour l'instanciation du graphe, vous avez 2 possibilités :

- Instancier le graphe de manière totalement aléatoire. Il suffira simplement d'indiquer au programme le nombre de villes que vous souhaitez, ainsi que la distance maximum possible entre ces villes
- Ou bien l'instancier de manière manuelle, c'est à dire qu'il faudra indiquer le nombre de villes à rentrer, puis donner à notre programme la distance entre chaque ville. Du fait que la

matrice est symétrique, pour une distance entre une ville A et une ville B, il suffira simplement d'indiquer la distance entre A et B.

Ces deux manières d'initialisation permettent rapidement de nous donner des instances de graphe utilises pour nos résultats. En utilisant la manière manuelle, nous avons pu comparer et observer la cohérence de nos résultats avec d'autres instances.

VI)Conclusion

Pour conclure ce projet, suite à différents résultats d'observation , nous avons remarqué que certaines fois la valeur du chemin optimal pouvait varier selon le choix de l'algorithme. En effet, pour des graphes assez gros, bien que le temps d'exécution puisse être un peu plus long, l'algorithme de recherche informée A* obtient une stratégie plus optimale que pour les algorithmes de recherche locale. Cependant, avec un graphe beaucoup plus gros en terme de villes et de distances, nous avons fait la déduction que l'algorithme de recherche locale était le mieux adapté.