Projet d'IA autour du traveling salesman problem

Algorithme A* :

Pour l'algorithme A* nous implémentons une class Etat qui va contenir toutes les données dont nous auront besoin pour l'exécution de l'algorithme A*. Les différentes variables sont décrites dans les commentaires.

```python
class Etat : # c'est la classe qui décrit nos différents états pour la recherche informée
    def __init__(self, visited=[], tovisit=[], startingcity = '', currentcity='', costs=[], cost=0, value = (-1), cities=[], hist = []):
        self.visited = visited #les villes deja visitees
        self.tovisit = tovisit # les villes a visiter
        self.startingcity = startingcity # ville de départ qui devra etre la ville d'arrivee
        self.currentcity = currentcity # la ville ou on en est
        self.costs = costs # une liste de liste [[a,b,3], [a,c,5], etc...]
        self.cost = cost # le coût de ce chemin à l'étape currentcity servira de g
        self.value = value # resultat de f = h + g
        self.cities = cities # l'ensemble des villes
        self.hist = hist # historique des chemins empreintés
```

Ci-dessous un exemple de représentation du graph pour ce problème :

```python
exampleDonnées = [['Paris', 'Marseille', 62], ['Paris', 'Lyon', 21], ['Paris', 'Nantes', 30], ['Paris', 'Rennes', 62], ['Paris', 'Toulouse', 81], ['Paris', 'Bordeaux', 21], ['Paris', 'Lille', 62],
    ['Marseille', 'Lyon', 52], ['Marseille', 'Nantes', 28], ['Marseille', 'Rennes', 54], ['Marseille', 'Toulouse', 37], ['Marseille', 'Bordeaux', 4], ['Marseille', 'Lille', 91],
    ['Lyon', 'Nantes', 88], ['Lyon', 'Rennes', 15], ['Lyon', 'Toulouse', 80], ['Lyon', 'Bordeaux', 44], ['Lyon', 'Lille', 21],
    ['Nantes', 'Rennes', 35], ['Nantes', 'Toulouse', 14], ['Nantes', 'Bordeaux', 46], ['Nantes', 'Lille', 73],
    ['Rennes', 'Toulouse', 70], ['Rennes', 'Bordeaux', 22], ['Rennes', 'Lille', 21],
    ['Toulouse', 'Bordeaux', 19], ['Toulouse', 'Lille', 71],
    ['Bordeaux', 'Lille', 72]]
```

Les actions possibles pour chaque états sont :

ACTION($Etat_i$) = { $GoEtat_j$} pour i différent de j

Dans notre programme c'est la fonction addpath qui fait passer d'un état à l'autre.

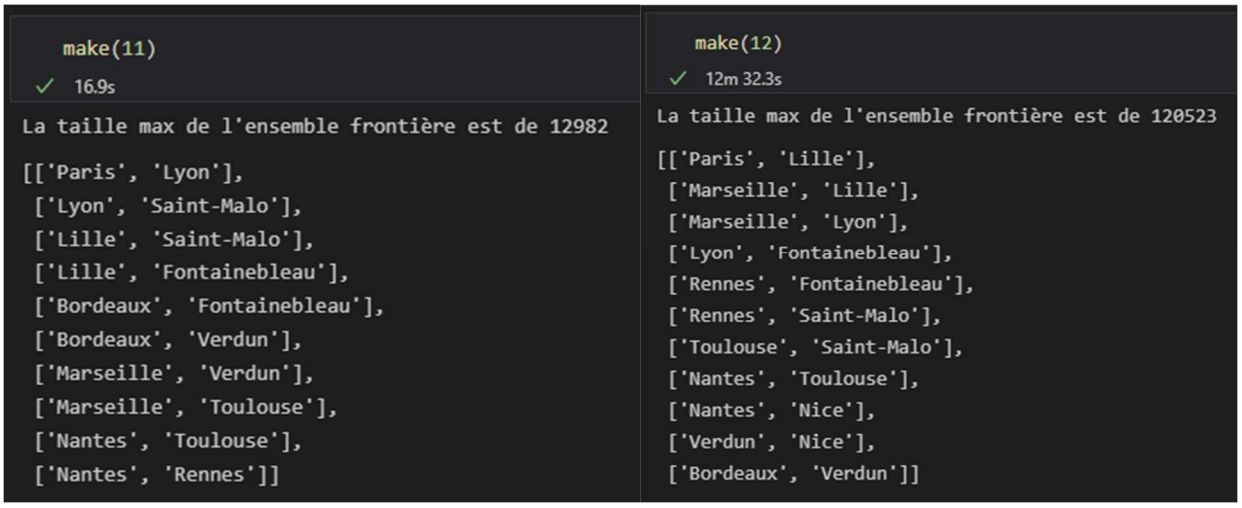
On peut donner également la fonction RESULT :

RESULT($Etat_i$, $GoEtat_j$) = $Etat_j$

Les données sont générées à partir d'un nombre de ville qui est donné par l'utilisateur. Ensuite le programme génère une liste de listes dans le format illustré ci-dessus. Ces données sont équivalentes aux données d'une matrice symétrique. Le nom des villes correspond à une vile d'arrivée ou de départ et le chiffre à la distance.

On peut aisément entrer les données pour 10 villes avec un temps d'exécution raisonnable, au-delà le temps d'exécution devient trop long c'est pourquoi je n'ai pas permis au programme de le faire.

Néanmoins vous pouvez voir des exemples ci-dessous.



On peut constater que le temps dépend de l'ensemble frontière à calculer. C'est pourquoi pour 11 villes nous avons un temps d'exécution court. Nous avons été chanceux et la frontière est petite taille.

Nous pouvons établir un tableau avec les temps constatés en secondes pour les différentes tailles de graph :

Nombre de villes	Temps d'exécution constaté
3	voilà le temps moyen pour 3 villes : 0.0014703679084777832
4	voilà le temps moyen pour 4 villes : 0.0028325629234313963
5	voilà le temps moyen pour 5 villes : 0.008053736686706543
6	voilà le temps moyen pour 6 villes : 0.026782453060150146
7	voilà le temps moyen pour 7 villes : 0.12208118915557861
8	voilà le temps moyen pour 8 villes : 0.4515434503555298
9	voilà le temps moyen pour 9 villes : 3.5976643800735473
10	voilà le temps moyen pour 10 villes : 12.190378284454345

Le Temps moyen a été calculé avec ces lignes de code. Le nombre d'itération est de 50. Le code se trouve en commentaire à la toute fin.

Algorithme Hill Climbing :

L'algorithme de Hill Climbing est bien plus rapide car il trouve une solution « locale » au problème.

Nous commençons par implémenter la classe Local qui représente un état de la recherche. On peut voir ci-dessous les différents attributs de la class.

```python
class Local:
    def __init__(self, circuit=[], cost = 0, costs=[]):
        self.circuit = circuit #le chemin qu'on prend
        self.cost = cost #le cout courant
        self.costs = costs # la matrice avec tous les couts
```

Les actions possibles pour chaque Local sont :

ACTION($Local_i$) = {HillClimb} pour i différent de j

Dans notre programme c'est la fonction qui fait passer d'un état à l'autre.

On peut donner également la fonction RESULT :

RESULT($Local_i$, HillClimb) = $Local_F$ avec $Local_F$ qui est le résultat

Les données sont représentées dans une matrice symétrique. Avec le coût entre les villes généré aléatoirement.

```
[[  0  90  55  78 132 168 129  75  59  97]
 [ 90   0  91  48 123  28  78  51  61 123]
 [ 55  91   0  64  62  65  91  61 169 130]
 [ 78  48  64   0  66 107 101  78  46  23]
 [132 123  62  66   0  67 132 131  89  55]
 [168  28  65 107  67   0  69 113  98 103]
 [129  78  91 101 132  69   0 131  83  71]
 [ 75  51  61  78 131 113 131   0  36 139]
 [ 59  61 169  46  89  98  83  36   0  95]
 [ 97 123 130  23  55 103  71 139  95   0]]
```

La fonction importante dans cet algorithme est « deuxopt », notre implémentation de l'algorithme 2-opt. Elle prend en argument une route et la matrice avec les coûts. Elle retourne la route trouvée par le Hill Climbing dans la variable result. La fonction est expliquée plus en détail dans le commentaire du code. On prend comme chemin initial 1-2, 2-3, …, (n-1)-n et on l'améliore jusqu'à trouver un minimum local ?

```python
def deuxopt(route, matgraph):
    result = route
    continuer = True
    taille = len(route)
    while continuer:
        continuer = False
        for i in range(1, taille - 2): #On test tous les voisins dans ces deux boucles on démarre à un car on ne change pas le point de départ
            for j in range(i + 1, taille):
                if (j != (i+1)):# on ignore le cas où i et j sont consécutifs
                    if (matgraph[result[i - 1]][result[i]] + matgraph[result[j - 1]][result[j]]) - (matgraph[result[i - 1]][result[j - 1]] + matgraph[result[i]][result[j]]) > 0:
                        result[i:j] = result[j - 1:i - 1:-1] #on doit modifier le sens de parcours entre les deux arcs qu'on a modifié
                        continuer = True # si c'est à False c'est qu'on a rien pu modifier pour améliorer l'algo, on est donc dans un creux de notre hill, un minimum local quoi
    return result
```

Pour cet algorithme on va choisir de regarder l'évolution du taux d'amélioration plutôt que le temps d'exécution, l'algorithme est très rapide. Nous allons observer pour 10,100,200,500,1000 et 2000 villes un taux d'amélioration croissant. Pour calculer le taux moyen j'ai fait la moyenne entre le taux d'amélioration de 100 exécutions. Ci – dessous des résultats (j'ai retiré le code du fichier, il s'agissait d'une simple boucle sur la fonction hillclimb).

```
Le taux d'amélioration moyen pour 10 villes :  -39.24560000000001
Le taux d'amélioration moyen pour 100 villes :  -77.57690000000001
Le taux d'amélioration moyen pour 200 villes :  -82.39640000000001
Le taux d'amélioration moyen pour 500 villes :  -87.30740000000002
Le taux d'amélioration moyen pour 1000 villes :  -89.8555
Le taux d'amélioration moyen pour 2000 villes :  -91.90799999999999
```

On observe que plus nous avons de villes et plus l'amélioration est significative. Cela s'explique par un plus grand manque à gagner avec un nombre croissant de villes.

Ma représentation des données étant différente pour les deux algorithmes je n'ai pas pu directement les comparer sur le même jeu de données et dire si HillClimb trouve le résulat optimal ou non en revanche nous pouvons comparer le taux l'amélioration des deux programmes

A*	HillClimb
Taux d'amelioration moyen pour 6 villes avec astar: -26.62500000000001	Taux d'amelioration moyen pour 6 villes avec HillClimbing: -24.938
Taux d'amelioration moyen pour 7 villes avec astar: -31.755399999999995	Taux d'amelioration moyen pour 7 villes avec HillClimbing: -29.586
Taux d'amelioration moyen pour 8 villes avec astar: -41.691	Taux d'amelioration moyen pour 8 villes avec HillClimbing: -33.684
Taux d'amelioration moyen pour 9 villes avec astar: -47.77200000000001	Taux d'amelioration moyen pour 9 villes avec HillClimbing: -37.016
Taux d'amelioration moyen pour 10 villes avec astar: -49.3655	Le taux d'amélioration moyen pour 10 villes : -39.24560000000001
Taux d'amelioration moyen pour 11 villes avec astar: -48.616	Taux d'amelioration moyen pour 11 villes avec HillClimbing: -42.467

Le code utilisé pour obtenir ce résultat est en commentaire à la fin du fichier. Nous faisons la moyenne entre un certain nombre d'itération de l'algorithme. 500 itérations pour Hillclimbing et Entre 10 et 100 pour A* étant donné le temps d'exécution plus long.

Nous constatons bien que l'algorithme A* donne un meilleur résultat pour ce qui est du taux d'amélioration. En effet nous trouvons la solution optimale alors que Hilclimbing trouve un minimum local. Le taux d'amélioration augmente avec le nombre de villes pour les deux algorithmes, dans les deux cas plus il y a de villes et plus grand est le manque à gagner.

Finalement nous avons deux approches différentes qui tendent vers le même résultat. Tout d'abord, une recherche informée qui permet de trouver la solution exacte mais avec une forte complexité, de ce fait nous ne pouvons calculer la solution pour un trop grand nombre d'éléments. Ensuite, nous avons une recherche non informée qui nous donne une solution approchée mais avec une complexité beaucoup plus faible.

A installer :

pip install pycopy-copy

pip install python-math

pip install random2

pip install ipywidgets

pip install numpy

Tout le code est sur un fichier unique, il suffit de l'exécuter.
Une fois le programme lancé les indications seront affichées pour sélectionner A* ou le HillClimbing ainsi que le nombre de villes sur lequel vous voulez faire tourner le programme.