Parcours des graphes

Objectif : L'objectif de ce tp est de rendre l'élève capable :

- charger un graphe avec une structure donnée
- parcourir le graphe en largeur ou en profondeur
- déterminer les distances d'un sommet à tous les autres sommets
  - Algorithme de Dijskra A,
  - Algorithme de Dijkskra A*
  - Algorithme de Bellman & Ford
- déterminer le plus court chemin entre deux sommets

1 Réseau autoroutier français

Le Tp GRAPHES-Tp-3 a permis de stoker dans une variable `graf`, le réseau autoroutier français et le Tp GRAPHES-Tp-4 a mis en place l'algorithme de Dijkstra.

Q - 1 : *Charger le fichier joint* `autoroutes.py` *et affecter* `graf` *à la variable* `reseau`.

Q - 2 : *Afficher dans une figure* ***Python*** *l'image* `autoroutes.jpg`.

Q - 3 : *Écrire une fonction qui* `affiche` *qui prend en argument* `points`*, une liste de points stockés sous la forme de tuple et deux arguments optionnels,* `pause = 0.2` *et* `col = 'b'`*, un flottant lié au temps de pause entre deux images et une chaine de caractère permettant l'affichage en couleur des points sur la carte.*

2 Algorithmes

3 Algorithme de Dijkstra A

Algorithm 1 Algorithme de Dijkstra

entrée: `graphe`, `sommet`
résultat: `d_fin`, `pere`
dijkstra(`graphe`, `sommet`)

```
d_ini = {s : float('inf') for s in graphe}
d_ini[sommet], d_fin, pere = 0, {}, {}
tant que taille(d_ini) >0 faire
    s_k = mini_dic(d_ini)
    pour s_v dans graphe[s_k] faire
        w = graphe[s_k][s_v]
        si s_v n'est pas dans d_fin et d_ini[s_v] > d_ini[s_k] + w alors
            d_ini[s_v] = d_ini[s_k] + w
            pere[s_v] = s_k
        fin si
    fin pour
    d_fin[s_k] = d_ini[s_k]
    supprime(d_ini[s_k])
fin tant que
renvoi: d_fin, pere
```

Algorithm 2 Plus court chemin entre deux sommets

entrée: `graphe`, `depart` et `arrivee`
résultat: `chemin`
dijkstra_pcc(`graphe`, `depart`, `arrivee`)

```
d_ini = {s : float('inf') for s in graphe}
d_ini[sommet], d_fin, pere = 0, {}, {}
tant que taille(d_ini) >0 et arrivee n'est pas dans d_fin faire
    s_k = mini_dic(d_ini)
    pour s_v dans graphe[s_k] faire
        w = graphe[s_k][s_v]
        si s_v n'est pas dans d_fin et d_ini[s_v] > d_ini[s_k] + w alors
            d_ini[s_v] = d_ini[s_k] + w
            pere[s_v] = s_k
        fin si
    fin pour
    d_fin[s_k] = d_ini[s_k]
    supprime(d[s_k])
fin tant que
renvoi: trajet(pere, depart, arrivee)
```

Algorithm 3 Obtention du chemin

entrée: `pere`, `depart` et `arrivee`
résultat: `chemin`
trajet(`pere`, `depart`, `arrivee`)

```
chemin = [arrivee]
tant que arrivee différent de depart faire
    arrivee = pere[chemin[-1]]
    ajouter arrivee à chemin
fin tant que
inverser chemin
renvoi: chemin
```

4 Algorithme de Dijkstra A*

Algorithm 4 Algorithme A*

entrée: `graphe`, `depart`, `arrivee`, `dico_coords`
résultat: `chemin`
A_star(`graphe`, `depart`, `arrivee`, `dico_coords`)

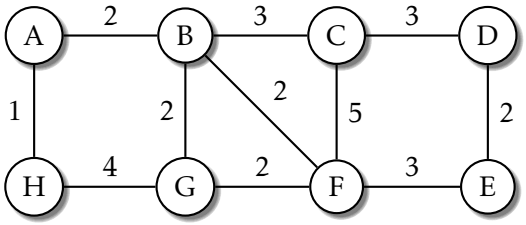
```
d_ini = {s : float('inf') for s in graphe}
d_ini[sommet], d_fin, pere = 0, {}, {}
d_heu = heuristique(graphe, arrivee, dico_coords)
tant que taille(d_ini) >0 et arrivee n'est pas dans d_fin faire
    s_k = mini_dic_heu(d_ini, d_heu)
    pour s_v dans graphe[s_k] faire
        w = graphe[s_k][s_v]
        si s_v n'est pas dans d_fin et d_ini[s_v] > d_ini[s_k] + w alors
            d_ini[s_v] = d_ini[s_k] + w
            pere[s_v] = s_k
        fin si
    fin pour
    d_fin[s_k] = d_ini[s_k]
    supprime(d[s_k])
fin tant que
renvoi: trajet(pere, depart, arrivee)
```

Algorithm 5 Recherche heuristique

entrée: `dist` un dictionnaire des distances initiales et `heu` un dictionnaire des évaluations heuristiques
résultat: `c`, la clé dont la valeur est la meilleure
mini_dic_heu(`dist`, `heu`)

```
val = float('inf')
pour k dans dic faire
    si dic[k] + heu[k] < val alors
        c, val = k, dic[k] + heu[k]
    fin si
fin pour
renvoi: c
```

4.1 Algorithme de Bellman & Ford



Principe :

> *Explorer tous les sommets dans le même ordre à chaque tour et pour chaque sommet exploré mettre à jour la distance des voisins. Faire cela jusqu'à convergence (au pire, nombre de sommets -1 fois)*

Algorithm 6 Bellman & Ford

entrée: `graphe` un graphe sous forme de dictionnaire de dictionnaires, `sommet` un sommet de `graphe`
résultat: `dist` un dictionnaire des distances de chaque sommet au sommet `départ`
bellman_ford(`graphe`, `sommet`)

```
n = taille(graphe)
dist = {s:float('inf') pour s dans graphe}
dist[sommet] = 0
pour i de 1 à n-1 faire
    pour s dans graphe faire
        pour v dans graphe[s] faire
            d = dist[s] + graphe[s][v]
            si dist[v] > d alors
                dist[v] = d
            fin si
        fin pour
    fin pour
fin pour
renvoi: dist
```

Q - 4 : *Écrire une fonction* `bellman_ford`.

On s'aperçoit que l'algorithme converge rapidement par rapport au nombre de tour de boucle initialement prévu. En effet, le pire cas est celui d'une chaine simple ouverte où le nœud de départ est à une extrémité. Pour atteindre l'autre extrémité, il faut $n-1$ itérations.

Q - 5 : *Améliorer le code pour le rendre plus pertinent.*