PROJET: COMMANDE À L'AUTRE BOUT DU MONDE

1. Recontextualisation

L'année : 2020. Le contexte ? Le confinement. L'objectif ? Trouver une occupation.
Pour cela, vous décidez d'aller errer dans les profondeurs d'Internet. Après des heures à passer de pages en pages les plus bizarres les unes que les autres, vous tombez sur un objet que vous DEVEZ acheter, tant il est formidable : un déguisement de licorne pour chat. Vous n'avez plus le choix, vous le commandez. Cependant, le délai de livraison est de plus d'un mois ! Forcément, ça vient de Chine. C'est loin la Chine. Bon, ça ne vous empêchera pas de le commander.
Mais quand même, vous commencez à vous poser la question : franchement, est-ce qu'ils ne sont pas juste mal organisés ? N'y aurait-il pas moyen de rendre la livraison plus rapide ? Est-ce que les transports en bateau pourraient être optimisés ? Est-ce qu'ils choisissent vraiment... LE PLUS COURT CHEMIN ?
Et d'un coup, vous réalisez que toutes vos années d'études n'ont pas été vaines. Vous allez enfin pouvoir mettre votre savoir en application.
Oui, vous vous ennuyez vraiment pour en être arrivé là dans votre journée. Oui ce confinement risque d'être long...

2. En pratique

2.1. **Réutilisation du code existant.** On cherche à trouver le temps minimum pour aller de Ningbo en Chine jusqu'à Fos-sur-mer à côté de Marseille (où se trouve le port le plus important du coin (sujet à débat, mais pas pour aujourd'hui)).
Vous allez devoir récupérer les différentes classes et méthodes utiles déjà implémentées lors des précédents TD/TP. Plusieurs graphes représentant les trajets de transports en bateaux vont vous être fournis. Nous parlerons ici surtout de "l'instance principale" qui vous est fournie, qui sera la première disponible. D'autres arriveront au fur et à mesure des semaines pour que vous puissiez faire plus de tests.
Ces graphes vous seront fournis sous forme de matrices d'adjacence associé à un fichier csv (qui contient l'identifiant des ports, leur latitude et leur longitudes). À vous d'adapter ces matrices pour les transformer en graphes, à l'aide éventuellement du parser qui vous est fourni (voir la Section 3 pour plus de renseignements), ou de proposer d'autre méthodes si vous le souhaitez.
Appliquez l'algorithme de Dijsktra sur les différentes instances fournies. Pour l'instance principale, l'identifiant du port de Ningbo est CNNGB, et l'identifiant de Fos-sur-mer est FRFOS.

2.2. **Fonctions sur le graphe.** Les personnes s'occupant de créer les graphes des ports dans le monde sont parfois fort tête en l'air, et on peut se retrouver avec un port où aucun bateau ne peut entrer ou sortir. Ce port n'est relié à aucun autre port, rendant ainsi le graphe non-connexe. Ajouter une méthode à la classe Graphe qui vous permettra de vérifier que le graphe est bien connexe, et qui lèvera une erreur si jamais le créateur de graphe s'était endormi avant de finir.
Vous coderez également une fonction qui vous permet de calculer la densité du graphe. Cela permettra de se faire une idée d'à quel point tous les ports sont reliés les uns aux autres dans le graphe.

2.3. **L'algorithme A*.** Si vous êtes arrivés ici, c'est avec une grande fierté que vous avez du trouver LE plus court chemin allant de Ningo à Fos-sur-mer ! Mais votre livreur a-t-il choisi ce chemin ? Il est probable que dans les faits, l'algorithme A* ait été utilisé. Cet algorithme permet de trouver un chemin (sans garantie que ce soit le plus court) à l'aide d'une *heuristique*, c'est à dire, une fonction qui va nous permettre d'estimer quel est le potentiel meilleur prochain sommet. En pratique, A* est extrêmement rapide, et si l'heuristique choisie est bonne, le chemin trouvé sera souvent proche de l'optimal. Cet algorithme est très souvent utilisé pour coder les déplacements des Intelligences Artificielles dans des jeux vidéos (Age of empire et Dofus avec certitude, très probablement Assassin's Creed, League of Legends et Worlds of Warcraft d'après des observations personnelles).

C'est donc à présent votre tour de l'implémenter ! Voici le pseudo-code de cet algorithme :

algorithme A_star(G, s, t)
Entrées : $G = (S, A, w)$ un graphe pondéré et s et t deux sommets de G
Sortie : Le poids du chemin découvert et un dictionnaire p qui à chaque sommet associe son parent, permettant de retrouver le chemin parcourut.

```
1  F = s                initialement, la file de priorité F ne contient que le sommet de départ
2  p = EMPTY            le dictionnaire p, qui permet de récupérer le chemin, est vide au départ
3  closed_list = EMPTY  le dictionnaire qui contient la liste des sommets déjà traités
4  g_score[s] = 0       le coût du sommet de départ jusqu'au sommet courant
5  f_score[s] = g_score[s] + heuristic(s,t)   le coût estimé du sommet de départ jusqu'au sommet
                                               d'arrivée en passant par le sommet courant
6  tant que F ≠ ∅ faire
7      u = Extraire_le_min(F)
8      si u = t alors
9          renvoyer p, g_score[t]
10     ajouter u dans closed_list
11     pour chaque sommet v ∈ G.voisins_sortants(u) faire
12         si v ∉ closed_list alors
13             tentative_g_score := g_score[u] + w(u,v)
14             si v ∉ F ou tentative_g_score < g_score[v] alors
15                 p[v] = u
16                 g_score[v] := tentative_g_score
17                 f_score[v] := g_score[v] + heuristic(v,t)
18                 si v ∉ F alors
19                     ajouter v dans F
20 renvoyer ERREUR
```

La fonction Extraire_le_min(F) sera différente de celle utilisée dans Dijkstra. Ici, le "min" que l'on cherche à récupérer, c'est le sommet ayant un `f_score` minimum appartenant à la file de priorité F.
L'heuristique que nous allons utiliser ici est toute simple : pour chaque sommet, vous calculerez la distance "à vol d'oiseau". Pour cela, les coordonnées GPS de chaque port (c'est à dire, chaque sommet du graphe) vous sont fournis. Vous pourrez alors implémenter une classe Coordinate, ayant trois attributs `x`, `y` et `name` et une méthode `distance` qui prend un deuxième Coordinate en argument, et calcule la distance entre ce point et `self`. Vous devrez implémenter la formule de Haversine. Vous ne la connaissez pas ? N'oubliez jamais, Google est votre meilleur ami (ou tout autre moteur de recherche qui aurait une place de prédilection dans votre coeur). Et ça fait parti de vos compétences à acquérir d'aller chercher des informations par vous même !

Voici les résultats attendus pour l'instance principale :

Dijkstra : CNNGB -> WP28 -> WP27 -> WP25 -> WP23 -> WP22 -> WP21 -> suezWP2 -> suezWP1 -> MTMAR -> WP20 -> WP19 -> MarseilleFos -> FRFOS
Temps : 430.9177216229062 heures

A* : CNNGB -> WP4 -> WP10 -> canalWP1 -> canalWP2 -> WP12 -> WP14 -> WP15 -> WP16 -> WP17 -> MarseilleFos -> FRFOS
Temps : 636.7721634594322 heures

3. Format des fichiers d'entrées

Des matrices d'adjacence vous sont fournies pour représenter différents exemples de graphes. Un fichier parser.py vous est donné également, qui vous permettra de lire les fichiers fournis, et stocker

les valeurs dans des listes. A vous ensuite d'utiliser ces listes comme vous le souhaitez afin de coder ce projet.
Pour chaque instance de problème, deux fichiers sont nécessaires : coordinates.csv et adjacency_matrix.txt. Le fichier coordinates.csv contient 4 colonnes :

(1) L'indice du tableau où sera stocké le port courant
(2) Le nom du port courant
(3) La latitude du port courant
(4) La longitude du port courant

Attention, les latitudes et longitudes sont données en degrés ! Il est fort probable que vous ayez besoin de les convertir en radians pour appliquer la formule de Haversine. Le parser lira ce fichier en entrée, créer un objet Coordinate pour chaque port, et stocke ces ports dans un tableau `coordinates`.
Quant à lui, le fichier adjacency_matrix.txt contient n lignes et n colonnes (où n est le nombre de port/sommets). La valeur à la ligne i, colonne j indique le poids de l'arrête qui va du sommet i au sommet j. Si la valeur est -1, l'arrête n'existe pas. Si c'est 0, alors i est égal à j. Toute autre valeur indique le temps nécessaire pour aller du port i au port j en bateau (en heures). Le parser va lire ce fichier, créer une liste de liste (une matrice donc) et stocke chaque valeur de sorte que adjacency_matrix[i][j] contienne le poids de l'arrête qui va de i à j.

Pour appeler les fonctions du parser, vous pouvez vous inspirez de ces lignes :

```
coordinates = parse_csv("input/coordinates.csv")
adjacency_matrix = parse_adjacency_matrix("input/adjacency_matrix.txt")
```

4. Truc en plus pour vous amuser

En vous rendant sur le site geojson.io et en copiant le contenu du fichier visu.geojson vous pourrez visualiser le graphe fourni sur une carte du monde ! Si c'est pas beau la technologie.

5. Organisation

— Le projet est à rendre au plus tard le 31 décembre à 23h59. Je n'accepte plus les rendus en 2021. Cela implique qu'après cette date, vous aurez 0. Apprendre à respecter les délais est une vraie compétence !
— Si vous me rendez votre projet avant les vacances (c'est à dire avant le 20 décembre à 23h59), vous aurez le droit à un point bonus (et des vacances avec l'esprit tranquille). Cependant, si vous me rendez un projet vierge, vous aurez quand même 0.
— Vous pouvez faire le projet en binôme (je préfère même, ça me fait moins de projets à corriger). Les trinômes ou plus sont interdits. Si vous êtes trois à me rendre un projet, je divise votre note par 3. Si vous êtes 4 à me le rendre, je divise par 4. Et ainsi de suite.
— Je serais intransigeante sur les tentatives de triche. Vous vous doutez bien qu'il existe de nombreux logiciels qui permettent de détecter les similarités entre deux projets. Et si vous vous posez la question, oui je compte vérifier tous vos projets avec ce type de logiciel. En cas de projets copiés, c'est zéro pour tout le monde, je me ficherai de savoir qui est à l'origine du code.

6. Barème

Ce barème est un barème provisoire. Il est là à titre indicatif mais est susceptible d'être modifié au besoin. Sachez également que si vous me proposez un projet qui n'a pas du tout suivi la structure proposé car vous souhaitiez faire le projet différemment, il n'y a aucun problème, je noterai cela en conséquence (p.ex. si vous n'aviez pas de classe Coordinate).

— Réutilisation du code existant pour appliquer Dijkstra et trouver le plus court chemin de la Chine jusqu'à Marseille *10 points*
  — Organisation en fichiers *1 point*

  - Nommage des variables de façon cohérente (tout en anglais ou en français, respect du CamelCase et snake_case pour les objets, variables, méthodes, ...) *1 point*
  - Lisibilité du code (nommage des variables, commentaires) *1 points*
  - Création d'un graphe et application de Dijkstra à partir des données fournies *4 points*
  - Récupération du chemin (dans le bon sens) et du temps de trajet du bateau *2 points*
  - Tests *1 point*
- Fonction permettant de vérifier que le graphe est bien connexe *1 point*
- Fonction permettant de calculer la densité du graphe *1 point*
- Implémentation de l'algorithme A* pour trouver un autre chemin de la Chine à Marseille *8 points*
  - Classe Coordinate *1 point*
  - Haversine *2 points*
  - Traduction du pseudo-code en python *3 points*
  - Heuristique *1 point*
  - Extraire_le_min(F) *1 point*
  - Récupération du chemin (dans le bon sens) et du temps de trajet du bateau *0 point (bah oui quand même, vous l'avez déjà fait pour Dijkstra)*
- L'utilisation de votre propre dijkstra/graphe/autre plutôt que ceux des corrections fournies pourra vous donner des points bonus, ainsi que l'implémentation de F comme un tas.

Bon projet à tous, et n'hésitez pas à poser vos questions sur le forum, elles pourront probablement être utiles à tout le monde ! Vous pouvez également me joindre par mail : manon.scholivet@lis-lab.fr ou bien manon.scholivet@univ-amu.fr. Je ne répondrai sur aucune autre adresse mail !