TP 15 compléments : Visualisation de l'algorithme de Dijkstra et algorithme A*

Dans ce TP, je vous propose d'étendre les visualisations proposées dans le TP 15 pour y inclure des visualisations permettant de mieux comprendre le fonctionnement de l'algorithme de Dijkstra.

Exercice 1 – Visualisations du TP 15

Le préambule fourni permet de manipuler des fichiers d'extension `.co` qui contiennent des positions de sommets dans un plan. On peut utiliser ces positions pour visualiser les graphes. Un tel fichier est fourni pour le graphe `large.gr`.
Deux fonctions sont fournies :

- `read_positions(file_name: str) -> list[tuple[int, int]]` qui prend en argument le nom d'un fichier `.co` et renvoie la liste des positions des sommets ;
- `afficher_positions(positions: list[tuple[int, int]]) -> None` qui prend en entrée une liste de positions et les affiche.
- `afficher_positions_gradient(positions: list[tuple[int, int]], gradient: list[float]) -> None` qui prend en entrée une liste de positions et, pour chaque position, une valeur entre 0 et 1 qui détermine la couleur du point affiché (dans un dégradé de couleur).

Dans les deux dernières fonctions, on choisit de ne pas y inclure l'instruction `plt.show()` dont l'effet est d'afficher le graphique. Cela vous permet d'y ajouter, si besoin, des affichages supplémentaires.

▶ **Question 1** Copier les fonctions les tester sur le fichier `positions_large.co` fourni.

★ **Question 2** Déterminer la ville duquel est tiré le graphe `large.gr`.

▶ **Question 3** Afficher la carte du graphe `large.gr` avec les distances minimales du sommet 0 à tous les autres sommets. *Il faut faire attention à normaliser les distances pour qu'elles soient comprises entre* 0 *et* 1.

▶ **Question 4** Tester pour différents sommets initiaux.

▶ **Question 5** Utiliser la même fonction pour afficher le plus court chemin entre deux sommets en superposition des distances minimales.

Exercice 2 – Visualisations supplémentaires

▶ **Question 1** Adapter la fonction `dijkstra` en une fonction `dijkstra_avec_limite_nb_tours(g, source, nb_tours)` qui s'arrête après un nombre de tours donné. *On pourra utiliser un compteur et l'instruction* `break` *pour arrêter la boucle prématurément. On renverra le tableau des distances.*

▶ **Question 2** Afficher la carte du graphe `large.gr` avec les distances minimales du sommet 0 à tous les autres sommets après un nombre limité de tours. *On pourra, par exemple, afficher les distances normalisées entre* 0 *et* 0.4 *pour les sommets déjà visités, et* 1 *pour les sommets à distance infinie.*

Parmi les sommets `s` du graphe, on en trouve trois types : les sommets déjà défilés par la file de priorité, les sommets enfilés qui ne l'ont pas encore été et les sommets pas encore enfilés (donc vérifiant `dist[s] == inf`). On appelle *fermés* les premiers, *ouverts* les seconds et *non atteint* les troisièmes. Dans l'algorithme de Dijkstra, quand un sommet *s* est fermé, on peut montrer que `dist[s]` est égal à la distance du sommet source à *s*, alors que ce n'est pas encore sûr pour les sommets ouverts.

▶ **Question 3** Modifier la fonction `dijkstra_avec_limite_nb_tours` pour qu'elle renvoie, en plus du tableau des distances, un tableau `fermes` qui contient les sommets fermés et un tableau `ouverts` qui contient les sommets ouverts.

▶ **Question 4** Afficher la carte du graphe `large.gr` avec les distances minimales du sommet 0 à tous les autres sommets après un nombre limité de tours en colorant avec un dégradé entre 0 et 0.4 les sommets fermés, 1 les sommets non atteints et en surimpression en noir les sommets ouverts.

On observe alors que les sommets ouverts constituent la *frontière* de la zone explorée par l'algorithme de Dijkstra. Cette frontière initialement réduite au sommet source, s'éloigne progressivement de celui-ci en englobant les sommets fermés.

Exercice 3 – Algorithme A*

Dans cet exercice, l'objectif n'est plus de déterminer les distances minimales d'un sommet source à tous les autres sommets, mais de déterminer la distance (et le plus court chemin) entre deux sommets donnés.

Plus court chemin entre deux sommets

Entrée :
- un graphe pondéré g représenté par une liste d'adjacence ;
- un sommet source s ;
- un sommet cible c.

Sortie : La distance entre ce sommet source et ce sommet cible.

On peut utiliser l'algorithme de Dijkstra pour résoudre ce problème : c'est ce que l'on a fait dans la fonction `plus_court_chemin` du TP 15. Cependant, cet appel est plus coûteux que nécessaire puisque l'on parcoure le graphe *en entier* alors que l'on pourrait s'arrêter dès que le sommet cible est défilé.

▶ **Question 1** Écrire une fonction `dijkstra_deux_sommets_optimise(g: list[list[tuple[int, float]]], source: int, cible: int) -> float` qui prend en argument un graphe g sous forme de liste d'adjacence, un sommet `source` et un sommet `cible` et renvoie la distance entre ces deux sommets. *Lorsque l'on défile de la file de priorité le sommet cible, on peut arrêter immédiatement la boucle avec l'instruction* `break`.

▶ **Question 2** Tester cette fonction sur le graphe `large.gr` pour différents couples de sommets. Comparer son temps d'exécution avec celui de la fonction `plus_court_chemin` pour des sommets plus ou moins proches géographiquement. On pourra également afficher le plus court chemin trouvé en ajoutant un tableau `pred` qui stocke pour chaque sommet le sommet précédent dans le parcours, ce qui permettra de reconstruire le plus court chemin.

Si l'on observe que l'algorithme de Dijkstra est plus rapide grâce à cette optimisation que sans elle pour des sommets proches, on constate également que l'exploration du graphe depuis le sommet initial s ne dépend pas de la position du sommet cible c : par exemple, si le sommet s est au centre (géographiquement) du graphe et le sommet c à l'Ouest, l'algorithme de Dijkstra va explorer le graphe dans toutes les directions jusqu'à trouver c. L'objectif de l'algorithme A* est de prendre en compte la position de c pour explorer en priorité les sommets s'en rapprochant et en évitant d'explorer les sommets qui s'en éloignent. Grâce au fichier `positions_large.co`, on peut définir une distance entre deux sommets qui correspondent à la distance euclidienne entre les deux points correspondants.

▶ **Question 3** Écrire une fonction `distance_vol_oiseau(positions : list[tuple[int, int]], s1 : int, s2 : int) -> float` qui prend en argument un tableau de positions de sommets, un sommet s_1 et un sommet s_2 et renvoie la distance euclidienne entre les deux sommets correspondants.

Cette distance permet d'adapter les priorités avec lesquelles les sommets sont enfilés dans la file de priorité. Au lieu d'insérer la distance connue à ce stade entre s et x comme priorité du sommet x, on peut utiliser la somme de cette distance et de la distance euclidienne entre x et c.

▶ **Question 4** Implémenter cette stratégie en une fonction `a_star(g: list[list[tuple[int, float]]], positions: list[tuple[int, int]], source: int, cible: int) -> float` qui renvoie la distance entre les sommets source et cible et le plus court chemin entre ces deux sommets. *On pourra utiliser un tableau supplémentaire* `pred` *qui stocke pour chaque sommet le sommet précédent dans le parcours.*

▶ **Question 5** Vérifier sur des exemples que les fonctions `dijkstra_deux_sommets_optimise` et `a_star` renvoient les mêmes résultats, et comparer leurs temps d'exécution pour des sommets éloignés.

Pour visualiser la différence entre ces deux algorithmes, on peut comparer les sommets ouverts et fermés pour chaque algorithme en visualisant les sommets parcourus par chacun d'eux.

▶ **Question 6** Adapter la fonction `dijkstra_deux_sommets_optimise` et `a_star` pour qu'elle renvoie le tableau de distance et le plus court chemin obtenus lors de l'exploration du graphe entre les sommets source et cible.

▶ **Question 7** Afficher la carte du graphe `large.gr` avec les distances minimales du sommet 0 à tous les autres sommets en colorant avec un dégradé entre 0 et 0.4 les sommets fermés, 1 les sommets non atteints et le plus court chemin entre les sommets source et cible en surimpression en noir pour les deux algorithmes.