TP13.3 - Graphes, Dijkstra et A^*

→ *Q.1) Reprenez le fichier du TP précédent.*

On rappelle qu'on a un terrain avec des obstacles donné par une matrice d'entiers. On dispose d'une fonction `graphe_pondere` qui convertit une telle matrice en graphe pondéré. On cherche maintenant à trouver le chemin le plus court entre deux sommets du graphe.

A) Dijkstra

On va maintenant programmer l'algorithme de Dijkstra qui permet de trouver un chemin de poids minimal entre deux sommets dans un graphe pondéré.
On a deux sommets `depart` et `arrivee` du graphe. Voici le plan de l'algorithme :

- on créé un dictionnaire `distances` dont les clés sont les sommets du graphe et toutes les valeurs sont `None`, sauf celle associée à `depart` qui est `(0, None)` : à la fin de l'algorithme, les valeurs de ce dictionnaire seront des couples `(distance_au_depart, predecesseur)`;
- on créé une liste `atraiter` des sommets à traiter qui contient au départ le sommet `depart`;
- tant que la liste `atraiter` est non vide :
  - on récupère le sommet `sommet` de la liste associé à la distance minimale dans `distances` et on l'enlève de la liste;
  - si ce sommet est `arrivee` on renvoie le chemin utilisé pour l'atteindre;
  - sinon, pour chaque voisin de ce sommet :
    - on calcule la distance à l'origine `dist` obtenue en ajoutant la distance de l'origine à `sommet` et le poids de l'arrête joignant `sommet` au voisin;
    - si le voisin est associé à `None` dans `distances`, on l'ajoute à la liste `atraiter`, et on lui associe le couple `(dist, sommet)` dans `distances`;
    - si le voisin n'est pas associé à `None`, on compare `dist` à la distance déjà associée dans `distances` et on met à jour le couple si besoin.

  Attention : chaque élément de `G[sommet]` est un couple `(voisin, poids_arrete)`. Quand on parcourt les voisins de `sommet`, il faut donc penser à quel élément du couple on a besoin.

Contrairement à ce qui a été dit en cours, on n'utilise pas de file de priorité dans cette implémentation.
Le résultat de cet algorithme est une liste de sommets de `G` commençant par `depart` et finissant par `arrivee`.
On va découper l'algorithme en petits morceaux.
On a déjà écrit dans le TP précédent les fonctions `init_dico` et `chemin` qu'on va pouvoir réutiliser (car le dictionnaire `distances` est du même type que `visites` du TP précédent).

→ *Q.2) Écrire une fonction `indice_min(d, L)` qui prend en paramètre un dictionnaire `d` et une liste `L` contenant des clés de `d` associées à des valeurs `(a, b)` où `a` est un nombre, et qui renvoie l'indice de l'élément `s` de `L` dont la valeur `d[s][0]` est minimale.*
Attention à ne pas confondre indice dans la liste et élément de la liste.

→ *Q.3) Écrire une fonction `extraire(d, L)` qui prend les mêmes paramètres que la fonction `indice_min`, qui supprime l'élément de `L` dont l'indice est renvoyé par `indice_min` et qui renvoie cet élément.*
On pourra utiliser `L.pop(indice)` qui supprime l'élément de `L` en position `indice` et le renvoie.

→ *Q.4) Écrire une fonction `Dijkstra(G, depart, arrivee)` qui renvoie un chemin de poids minimal entre les sommets `depart` et `arrivee` dans le graphe `G` en utilisant toutes les fonctions programmées précédemment.*

On peut maintenant faire la recherche de chemin pour notre personnage.

→ *Q.5) Créer votre matrice de taille `(n, m)` et visualisez la (elle doit avoir un* 1 *en haut à gauche et en bas à droite). À l'aide des fonctions `graphe` et `Dijkstra`, déterminer le plus court chemin allant de `(0,0)` à `(n-1, m-1)`. Vous pouvez utliser la fonction `visualise_chemin_bis` pour voir le chemin trouvé.*

B) A^*

L'algorithme de Dijkstra fonctionne correctement, mais lorsque le graphe est gros, il peut prendre beaucoup de temps. On va utiliser une variante de Dijsktra, en « guidant »le choix des sommets à prendre. Autrement dit, l'algorithme est le même que Dijkstra, mais si on note $d(i)$ la distance, on va prendre le sommet de la liste `atraiter` qui minimise la quantité $d(i)+h(i)$ où h est une fonction qui à chaque sommet associe un nombre : c'est ce qu'on appelle utiliser une **heuristique**. On va utiliser ici deux fois la distance à vol d'oiseau de l'arrivée pour la fonction h.

→ *Q.6) Écrire une fonction* `oiseau(sommet1, sommet2)` *qui prend deux sommets du graphe et renvoie deux fois la distance à vol d'oiseau entre les deux.*

→ *Q.7) Écrire une fonction* `Aetoile(G, depart, arrivee, h)` *qui fait la même chose que* `Dijkstra(G, depart, arrivee)` *mais en utilisant l'heuristique* `h`*.*

→ *Q.8) Comparer les résultats des deux fonctions en utilisant l'heuristique* `oiseau`*. On pourra rajouter un compteur aux deux fonctions pour compter combien on a fait d'itérations.*

C) Files de priorité

On va implémenter simplement la structure de file de priorité : une file sera encodée par une liste de couples `(priorite, element)`. Plus `priorite` est petit, plus l'élément est proche de la tête.

— la tête de la file sera le dernier élément de la liste ;

— pour enfiler un élément, on l'ajoute en tête de file puis on le décale progressivement jusqu'à trouver sa place ;

— pour mettre à jour une priorité, on commence par chercher la position de l'élément dans la file, puis on le décale pour le mettre à sa place.

→ *Q.9) Écrire les trois fonctions* `defiler(F)`, `enfiler(F, priorite, element)` *et* `changer_priorite(F, priorite, element)` *correspondant aux opérations sur le file de priorité* `F`*.*

Cette implémentation n'est pas optimale du tout : l'enfilage d'un élément se fait en complexité linéaire alors que l'implémentation de `PriorityQueue` fait cette opération avec une complexité logarithmique.