TP 15 : Algorithme de Dijkstra

Dans ce TP, on va implémenter un algorithme permettant de calculer le plus court chemin entre un sommet source et tous les autres sommets du graphe. Cet algorithme est appelé *algorithme de Dijkstra*.

Exercice 1 – File de priorité

Les files de priorité sont une structure de donnée linéaire similaire aux piles et aux files : on y ajoute et enlève des éléments un par un, mais chaque élément inséré est associé à une *priorité*, et les éléments retirés d'abord sont ceux de plus petite priorité.
On manipulera les files de priorité en utilisant la classe *heapq* de Python. Cette classe permet de manipuler des listes comme des files de priorité, en utilisant la fonction `heappush` pour ajouter un élément et `heappop` pour le retirer.

Python
```python
import heapq
heap = [] # initialisation d'une file de priorité vide
heapq.heappush(heap, (3, "Jeanne")) # ajout d'un élément de priorité 3
heapq.heappush(heap, (1, "Sophie")) # ajout d'un élément de priorité 1
heapq.heappush(heap, (2, "Thomas"))  # ajout d'un élément de priorité 2
print(heapq.heappop(heap)) # affiche (1, "Sophie")
```

Pour éviter de se tromper, on va définir une interface pour manipuler ces files de priorité. On considère des priorités de type `float` et des éléments de type `Any` (de n'importe quel type).

▶ **Question 1** Écrire une fonction `creer_file()` qui renvoie une file de priorité vide.

▶ **Question 2** Écrire une fonction `ajouter(file, priorite, element)` qui ajoute un élément `element` de priorité `priorite` à la file de priorité `file`.

▶ **Question 3** Écrire une fonction `retirer(file)` qui retire et renvoie l'élément de priorité minimale de la file de priorité `file`.

▶ **Question 4** Écrire une fonction `est_vide(file)` qui renvoie `True` si la file de priorité `file` est vide, et `False` sinon.

▶ **Question 5** Écrire dans l'interpréteur les instructions correspondant aux actions suivantes : créer une file de priorité, y ajouter les éléments `"Brioche"`, `"Croissant"` et `"Pain de mie"` de priorités respectives 3, 1 et 2, puis retirer l'élément de priorité minimale.

Exercice 2 – Algorithme de Dijkstra

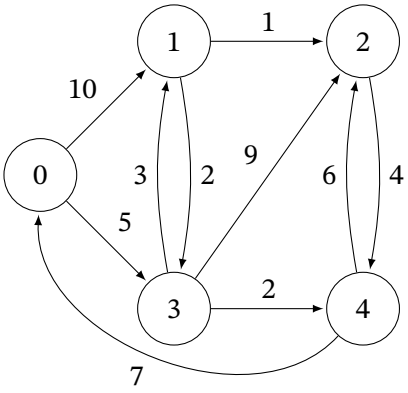
L'algorithme de Dijkstra est un parcours de graphe qui utilise à son avantage une structure de *file de priorité* pour parcourir les sommets du graphe dans l'ordre croissant de leur distance à un sommet source donné. Il ne s'applique qu'à la condition que les poids des arcs du graphe considéré soient positifs.
Comme pour la fonction `distance`, on utilisera un tableau `dist` pour stocker les distances du sommet source aux autres sommets. Le principe est le suivant pour un graphe à n sommets :

- On initialise `dist` un tableau de taille n avec des distances infinies[a] sauf pour le sommet source qui est à distance 0.
- On crée une file de priorité `file` et on y ajoute le sommet source avec une priorité de 0.
- Tant que la file n'est pas vide, on retire le sommet `i` de priorité minimale : pour chaque successeur `j` de `i`, si la distance de `j` est plus grande que la distance de `i` plus le poids de l'arc `i` → `j`, on met à jour la distance de `j` et on ajoute `j` à la file avec la nouvelle distance.
- À la fin, le tableau `dist` contient les distances minimales du sommet source à tous les autres sommets.

▶ **Question 1** Écrire une fonction `dijkstra(g: list[list[tuple[int, float]]], source: int) -> list[float]` qui prend en argument un graphe `g` sous forme de liste d'adjacence et un sommet `source` et

renvoie un tableau `dist` tel que `dist[i]` contient la distance minimale du sommet `source` au sommet `i`.

▶ **Question 2** Tester votre fonction sur le graphe tiré du fichier `small.gr` fourni. On rappelle que ce graphe est le suivant :



▶ **Question 3** Déterminer pour chaque graphe le sommet le plus lointain au sommet 0 et la distance correspondante.

▶ **Question 4** Écrire une fonction `plus_court_chemin(g: list[list[tuple[int, float]]], source: int, cible: int) -> list[int]` qui prend en argument un graphe `g` sous forme de liste d'adjacence, un sommet `source` et un sommet `cible` et renvoie la liste des sommets formant le plus court chemin du sommet `source` au sommet `cible`. *Pour cela, on pourra utiliser un tableau supplémentaire* `pred` *qui stocke pour chaque sommet le sommet précédent dans le parcours.*

a. On utilisera la valeur `inf` du module `math` pour représenter l'infini.

Exercice 3 – Visualisation

Le fichier `positions.py` fourni sur le cahier de prépas 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 ;
- `draw_positions(positions: list[tuple[int, int]]) -> None` qui prend en entrée une liste de positions et les affiche.
- `draw_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).

▶ **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.