Algorithme de Dijkstra

Implémentation du graphe représentant le réseau

La représentation du réseau figure 1 sera implémentée en mémoire par la classe `Graphe_pondere()`. Dans cette classe la représentation du graphe est réalisé par un dictionnaire d'adjacence (appelé communément liste d'adjacence).

Dans ce dictionnaire les clés portent le nom des sommets. Les valeurs constituent les liste des voisins du nœud associés à l'étiquette de l'arête liant le nœud au voisin correspondant (tuple).

Dans l'exemple représenté figure 1, le dictionnaire d'adjacence est :

```
self.__adj → {'R1' : [('R2' , 0.1) , ('R3' , 1)] , 'R2' : [('R1' , 0.1) , ('R3' , 10)] , 'R3' : [('R1' , 1), ('R2' , 10)]
```

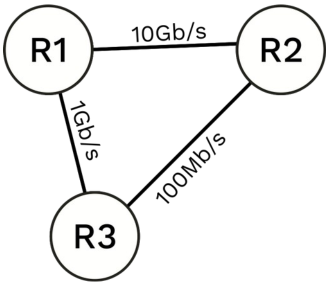

Figure 1: Exemple de graphe

Cet objet `graphe_pondere()` est construit avec la suite d'instructions :

```
>>> network = Graphe_pondere()
>>> network.ajouter_arete('R1', 'R2', 0.1)
>>> network.ajouter_arete('R1', 'R3', 1)
>>> network.ajouter_arete('R2', 'R3', 10)
```

La classe `Graphe_pondere()` est disponible dans le fichier graphe.py

Q1. **Compléter** les méthodes `voisins()` et poids() de la classe `Graphe_pondere()` afin de respecter la spécification associée.

Q2. **Compléter** la fonction `test_graphe()` en respectant les consignes notées en commentaire.

Q3. **Implémenter** avec la classe `graphe_pondere()`, le graphe défini figure 3. **Vérifier** la validité du graphe.

Description de l'algorithme

L'algorithme de Dijkstra peut être implanté de la façon suivante :

- ✗ Trois listes sont initialisées :
  - ➔ la liste des sommets déjà explorés `lst_e = []`;
  - ➔ la liste des sommets atteignables par une arête depuis un sommet déjà exploré et qui n'ont pas encore été explorés : `lst_v = [start]`;
  - ➔ le dictionnaire des distances à partir de `start` et des pères (sommets permettant d'accéder de manière optimale au sommet cible) : `table_dist`. Initialement, `table_dist` est composée de n fois `[float('inf'), None]` sauf `table_dist[start] = [0,None]`. La valeur `float('inf')` est utilisée pour

dénoter un sommet qui n'est pas encore accessible, elle est plus grande que tout nombre, permettant des comparaisons aisées.

✗ Tant que `lst_v` n'est pas vide, on sélectionne le sommet `s1` de `lst_v` qui a la plus courte distance à `start`. On supprime ce sommet de `lst_v` et on l'ajoute à `lst_e`.

✗ Pour chaque voisin `s2` de `s1` on regarde s'il est dans lst_e. Si ce n'est pas le cas, on on met à jour `table_dist` :

```
table_dist[s2] = (table_dist[s1][0] + graph.poids(s1,s2), s1)
```

✗ On connaît alors la plus courte distance de i à tout sommet accessible et, en remontant dans l'ordre les pères, on peut reconstituer le chemin qui a cette longueur.

La fonction dijkstra() disponible dans le fichier disjkstra.py calcule la table des distances selon l'algorithme de Dijkstra.

Q4. **Relever** la table des distances calculée par la fonction dans le cas du graphe défini figure 2 (graphe étudié lors du précédent TD). **Vérifier** et **valider** le résultat obtenu.

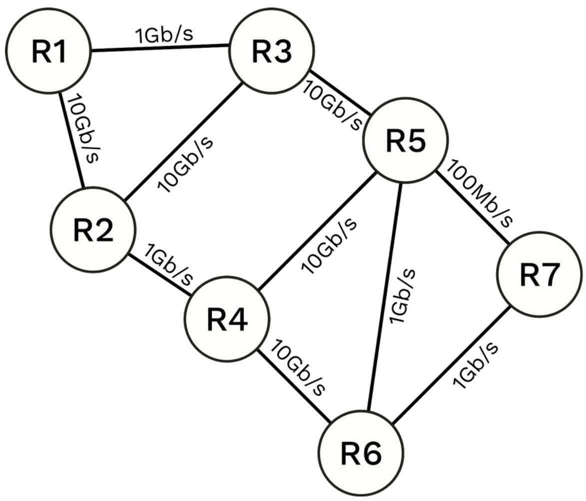

Figure 2: Exemple de réseau maillé

Q5. **Relever** dans le tableau suivant, pour chaque itération de la boucle while, l'état des variables `lst_e`, `lst_v` et `table_dist`

lst_v	**lst_e**	**s1**	**table_dist**						
			R1	R2	R3	R4	R5	R6	R7
['R1']	[ ]	x	[0, N]	[+∞, N]	[+∞, N]	[+∞, N]	[+∞, N]	[+∞, N]	[+∞, N]

Q6. **Comparer** cette table à celle définie lors du travail dirigé précédent.

Recherche du plus court chemin

Q7. **Compléter** la fonction `chemin()` afin d'extraire de la table de distance le plus court chemin entre deux sommets du graphe