0.1 L'algorithme de Dijkstra

L'algorithme de Dijkstra détermine les plus courts chemins entre un sommet origine (ou source) d'un graphe G orienté pondéré avec des poids positifs et les autres sommets du graphe.
A chaque itération, l'algorithme de Dijkstra choisit le sommet dont l'estimation de plus court chemin est minimale.

Algorithme 1 : Algorithme de Dijkstra

```
1  /* G est le graphe et s l'origine du graphe
2  /* d(v,w) représente le poids de l'arête reliant les deux sommets v et w
3  /* inf est l'infini et Nil est l'objet vide
4  fonction Dijkstra(G,s)
5  /* Initialisation
6  F ⟵ filePriorite()
7  pour chaque sommet u dans G faire
8  |  u.d ⟵ inf
9  |  u.couleur ⟵ 'blanc'
10 |  u.parent ⟵ Nil
11 fin
12 s.d ⟵ 0
13 s.couleur ⟵ 'gris'
14 F.enfiler(s)
15 tant que F est non vide faire
16 |  /* on défile dans la file de priorité le sommet v de plus courte distance
17 |  /* depuis l'origine s
18 |  v=F.defiler()
19 |  pour chaque voisin w de v dans le graphe G faire
20 |  |  si w.couleur == 'blanc' alors
21 |  |  |  w.couleur ⟵ 'gris'
22 |  |  |  w.parent ⟵ v
23 |  |  |  w.d ⟵ v.d + d(v,w)
24 |  |  |  F.enfiler(w)
25 |  |  fin
26 |  |  /* Relaxation
27 |  |  si w.couleur == 'gris' et v.d + d(v,w) < w.d alors
28 |  |  |  w.d ⟵ v.d + d(v,w)
29 |  |  |  w.parent ⟵ v
30 |  |  fin
31 |  fin
32 |  v.couleur ⟵ 'noir'
33 fin
```

Pour afficher le plus court chemin entre le sommet origine et un sommet du graphe, on peut utiliser la fonction récursive Afficher_chemin

Algorithme 2 : Afficher le plus court chemin entre l'origine et un sommet quelconque

```
1  /* G est le graphe , s l'origine du graphe , x le sommet à atteidre ;
2  /* afficher(a) est une fonction qui sert à afficher a sur la sortie standard ;
3  fonction Afficher_chemin(G,s,x) ;
4  si  s == x alors
5  |  afficher(s)
6  fin
7  sinon
8  |  Afficher_chemin(G,s,x.parent)
9  |  afficher(x)
10 fin
```

Sur la structure de données **heapdict** , voici ce qu'en dit l'IA de Gemini:

Structure de données : heapdict est une librairie (qui doit être installée séparément, par exemple via pip install heapdict) qui implémente une table de hachage basée sur un tas. Elle combine les caractéristiques d'un dictionnaire Python standard avec les propriétés d'un tas.

Fonctionnalités principales :

Fonctionne comme un dictionnaire : Vous pouvez utiliser des clés et des valeurs comme dans un dictionnaire Python normal.

- **popitem()** : Retire et renvoie la paire clé-valeur avec la priorité la plus basse . Contrairement à un dictionnaire standard, popitem() renvoie l'élément avec la priorité la plus faible.
- **peekitem()** : Renvoie la paire clé-valeur avec la priorité la plus basse sans la supprimer.
- Modification efficace de la priorité : Contrairement à **heapq**, heapdict permet de modifier efficacement la priorité d'un élément existant (opération souvent appelée "decrease-key" ou "increase-key"). C'est une fonctionnalité importante pour certains algorithmes comme l'algorithme de Dijkstra ou A*.

Usage typique : Conçu pour être utilisé comme une file de priorité où vous avez besoin de stocker des éléments avec des priorités, et où il est fréquent de devoir mettre à jour la priorité d'un élément déjà présent. Il est donc utile dans les situations où la capacité de modifier la priorité d'un élément est cruciale.

```
1
2  from heapdict import heapdict
3
4  # Créer un min-heap vide
5  min_heap = heapdict()
6  # Ajouter des éléments au min-heap
7  min_heap['a'] = 10
8  min_heap['b'] = -5
9  min_heap['c'] = 20
10 min_heap['d'] = 15
11
12 print("Min-heap initial:", min_heap)
13
14 #Obtenir l'élément avec la plus petite valeur (le "sommet" du min-heap)
15 x_cle,y_valeur = min_heap.popitem()
16 print(f"L'élément le plus grand est: clé={x_cle}, valeur={y_valeur}")
```