Algorithme de recherche du plus court chemin

Ce document explique dans les grandes lignes l'algorithme de Dijkstra en théorie de graphes et son application pour trouver le plus court chemin entre deux points dans une grille de labyrinthe. Ensuite, une implémentation de cet algorithme en Python est expliquée.

1 Algorithme de Dijkstra

1.1 Dans un graphe

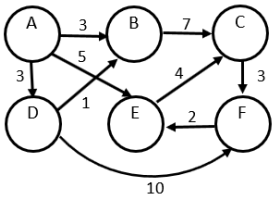


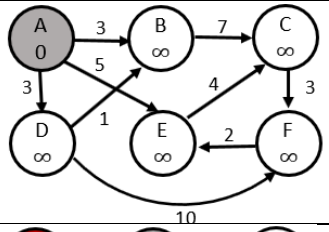
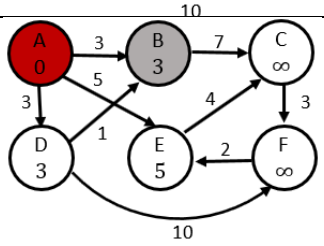
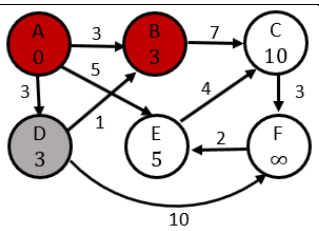
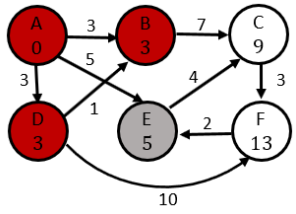
Figure 1: Exemple de graphe orienté pondéré

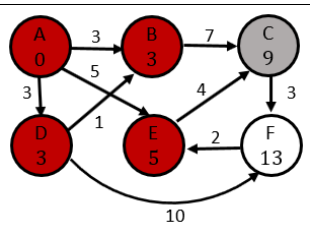
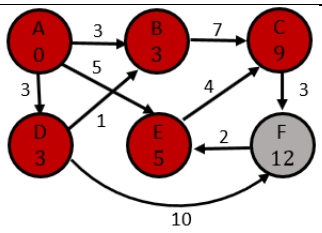
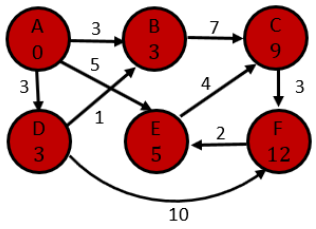
Un graphe est dit orienté lorsque les arrêtes peuvent être parcouru dans un sens unique. Une arrête orientée est appelée un arc. Un graphe est dit pondéré lorsque ses arrêtes sont étiquette par un nombre réel positif ou nul.

L'algorithme de Dijkstra s'applique sur un graphe pondéré, on le supposera aussi orienté dans ce papier.

L'algorithme de Dijkstra permet de calculer la distance, c'est-à-dire le poids du chemin entre un sommet de départ et tout les autres sommets du graphes.

Prenons, par exemple, le graphe de la Figure 1 et prenons A comme sommet de départ. Voyons étape par étapes l'algorithme de Dijkstra :

N° étape	Description étape	Graphes à la fin de l'étape
1	On note pour chaque sommet *d[sommet]* la distance de ce sommet au sommet A. On initialise les distances par *d[A]=0* et *d[tous les autre sommet]=+inf.*	
2	On cherche tous les voisins non parcourus de A : il y a B, D et E. On met à jour les distance des voisins : *d[B]=d[A]+3=3, d[D]=d[A]+3=0* et *d[E]=d[A]+5=5* Le sommet A est noté comme déjà parcouru (colorié en rouge) et le nouveau sommet d'intérêt (colorié en gris) et le voisin avec la distance minimal, ici B ou D. Prenons B.	
3	On répète l'étape 2 mais en partant de B. Si deux chemins mènent au même commet, on garde la distance minimale. Le nouveau sommet d'intérêt est D.	
4	On répète l'étape 2 mais en partant de D. Le nouveau sommet d'intérêt est E.	

5	On répète l'étape 2 mais en partant de E. Le nouveau sommet d'intérêt est C.	
6	On répète l'étape 2 mais en partant de C. Le nouveau sommet d'intérêt est F.	
7	On répète l'étape 2 mais en partant de F. Tous les sommets ont été parcouru, l'algorithme s'arrête. On sait alors que le plus court chemin pour aller de A à F a un poids de 12. Pour savoir le chemin il aurait fallu noter les antécédents des sommets à chaque changement des distances.	

Exemple d'implémentation en pseudo-code (source Wikipédia) :

```
Entrées : G = (S, A) un graphe avec une pondération positive poids des arcs, s_deb un sommet de S

P := ∅
d[a] := +∞ pour chaque sommet a
d[s_deb] = 0
Tant qu'il existe un sommet hors de P
    Choisir un sommet a hors de P de plus petite distance d[a]
    Mettre a dans P
    Pour chaque sommet b hors de  P voisin de a
       Si d[b] > d[a] + poids(a, b)
           d[b] = d[a] + poids(a, b)
           prédécesseur[b] := a
    Fin Pour
Fin Tant Que
```

1.2 Dans une grille de labyrinthe

On considère un labyrinthe codé par une matrice dont les éléments sont soit 0 soit 255, 0 (resp. 255) correspondant à une case où l'on peut (resp. ne peut pas) se déplacer.

Une telle matrice se transforme facilement en un graphe pondéré, chaque élément correspondant à un sommet relié par des arrêtes à ses quatre voisins (haut, bas, gauche, droite). Les arrêtes sont pondérés par des poids calculés par :

$$poid\ entre\ les\ sommets\ (i,j)\ et\ (k,l) = 0.1 + (matrice[i,j] - matrice[k,l])^2$$

Cette définition assure un poids non nul et le poids d'un chemin passant à travers un mur du labyrinthe est tellement grand que ce chemin ne sera jamais emprunté.

Dans le cas de la recherche du plus court chemin dans un labyrinthe, il est essentiel de se stocker le chemin parcouru. Pour cela, lorsque l'on actualise la distance d'un sommet, il faut stocker le sommet précédent dans le chemin qui donne cette distance minimale.

2 Implémentation en Python 3

L'implémentation est assez directe avec le pseuso-code donné précédemment. Le seul point compliqué est la gestion de la liste des sommets non parcourus et la recherche du sommet d'intérêt (=de distance minimale) dans cette liste. Pour cela nous utilisons une structure en tas-min (*min-heap* en anglais). Cela permet de réduire la complexité de l'algorithme.

Un tas-min est une structure de donnée sous forme d'arbre binaire complet : chaque parent a une valeur plus petite (ou égale) que ses 2 enfants et un enfant peut avoir à son tour un enfant uniquement si son parent a deux enfants. Une telle structure peut être représenté dans une liste : le parent racine est à la position 0 dans la liste et ses enfants sont aux position s1 et 2. Ensuite les enfants de 1 sont aux positions 1+2=3 et 1+2+1=4. Ainsi, les enfants d'un élément à la position i sont aux position 2i+1 (enfant de gauche) et 2i+2 (enfant de droite). Et le parent d'un élément à la position i est à la position (i-1)/2 arrondi à l'entier inférieur.

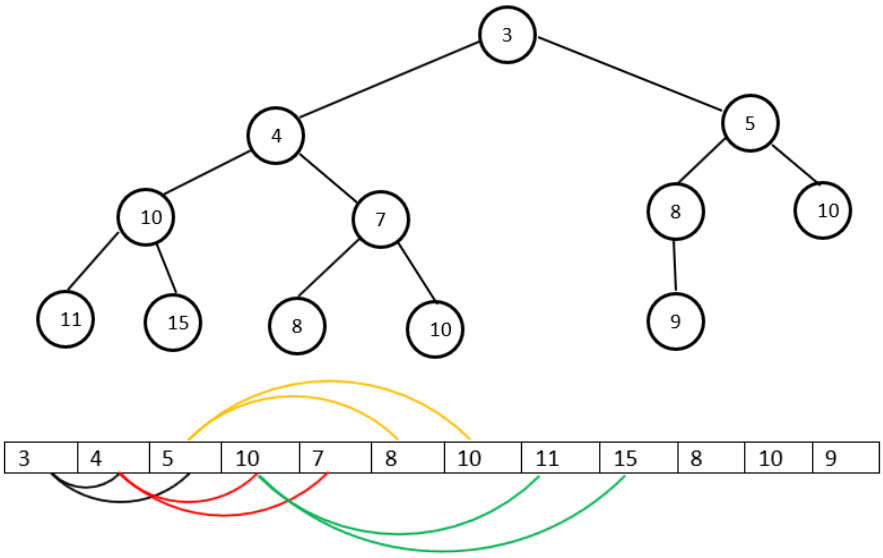


Figure 2: exemple de tas-min en graphe et en liste

Dans l'implémentation de l'algorithme de Dijkstra nous avons besoin de deux fonctions sur les tas-min :

- Une fonction qui permet en partant d'un indice i donné de réordonner tous les éléments d'indice supérieur ou égale à i pour respecter la structure d'un tas-min, fonction appelé *bubble_down* prenant en argument la liste à ordonner et l'indice i et renvoyant la liste réordonnée.

- Une fonction qui permet en partant d'un indice i donné de réordonner tous les éléments d'indice inférieur ou égale à i pour respecter la structure d'un tas-min, fonction appelé *bubble_up* prenant en argument la liste à ordonner et l'indice i et renvoyant la liste réordonnée