Algorithme de Dijkstra

Structures de données

Remarque 1. Ouverture hors programme de terminale NSI : l'algorithme de Dijkstra n'est pas exigible au baccalauréat ; il prolonge le chapitre sur les parcours de graphes.

① Principe de l'algorithme

Pour déterminer le plus court chemin entre deux sommets d'un graphe pondéré, on pourrait essayer d'énumérer tous les chemins possibles et comparer leurs poids. Mais cette approche devient impraticable dès que le graphe est un peu grand.

E. W. Dijkstra, informaticien néerlandais (1930–2002), a proposé en 1959 un algorithme bien plus efficace. Son principe repose sur une idée fondamentale : si le plus court chemin de s_0 à un sommet S passe par les sommets intermédiaires $s_1, s_2, \ldots, s_k$, alors chaque sous-chemin $s_0 \to s_1 \to \cdots \to s_i$ est lui-même un plus court chemin de s_0 à s_i.

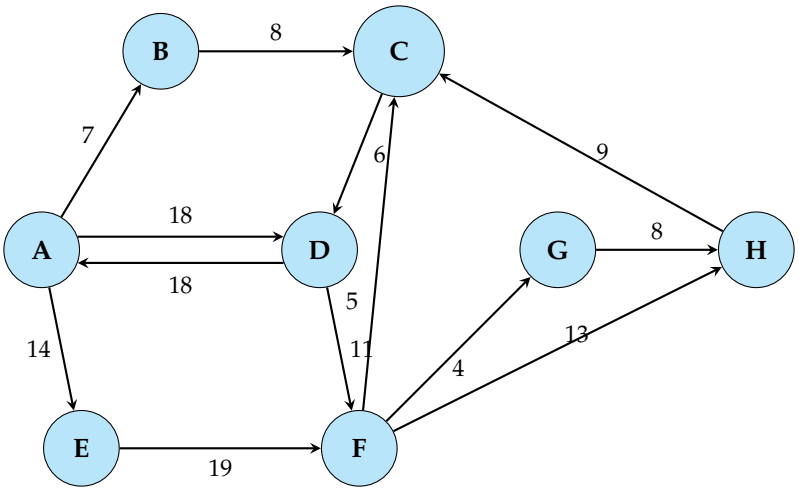
L'algorithme construit la solution de proche en proche :

1. Initialiser la distance du sommet de départ à 0 et celle de tous les autres sommets à $+\infty$.
2. Sélectionner parmi les sommets non encore traités celui dont la distance provisoire est **minimale**.
3. Pour chacun de ses voisins, vérifier si le passage par ce sommet offre un chemin plus court : si oui, **mettre à jour** la distance et le prédécesseur.
4. Répéter jusqu'à avoir traité tous les sommets.

Condition importante : l'algorithme de Dijkstra ne fonctionne que si tous les poids sont **positifs ou nuls**.

② Exemple détaillé

On cherche le plus court chemin de A à H dans le graphe orienté pondéré suivant :



2.1 Tableau de Dijkstra

On remplit le tableau étape par étape. La notation 7_A signifie « distance provisoire 7, en venant de A ».

Étape	A	B	C	D	E	F	G	H	Sélectionné
Init	0	∞	∞	∞	∞	∞	∞	∞	**A** **(0)**
1		7_A	∞	18_A	14_A	∞	∞	∞	**B** **(7)**
2			15_B	18_A	14_A	∞	∞	∞	**E** **(14)**
3			15_B	18_A		33_E	∞	∞	**C** **(15)**
4				18_A		33_E	∞	∞	**D** **(18)**
5						29_D	∞	∞	**F** **(29)**
6							33_F	42_F	**G** **(33)**
7								41_G	**H** **(41)**

Détail des mises à jour : à l'étape 4, le sommet D est atteint directement depuis A (distance 18), ce qui est meilleur que le passage par B et C ($7 + 8 + 6 = 21$). À l'étape 5, F est mis à jour : $18 + 11 = 29 < 33$.

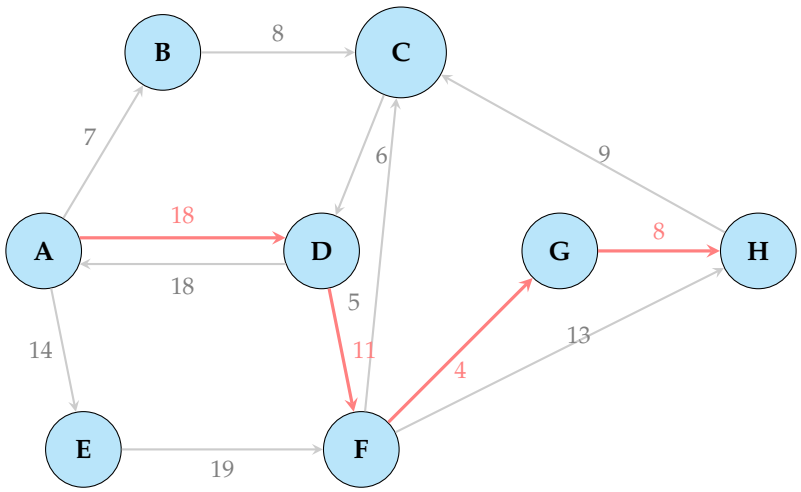
2.2 Reconstitution du chemin

On remonte les prédécesseurs depuis H :

$$H \xleftarrow{G} G \xleftarrow{F} F \xleftarrow{D} D \xleftarrow{A} A$$

Le plus court chemin est donc :

$$A \to D \to F \to G \to H \quad \text{(distance : 41)}$$



③ Implémentation en Python

On représente le graphe pondéré par un dictionnaire de dictionnaires :

```
graphe = {
    "A": {"B": 7, "E": 14, "D": 18},
    "B": {"C": 8},
    "C": {"D": 6},
    "D": {"A": 18, "F": 11},
    "E": {"F": 19},
    "F": {"G": 4, "H": 13, "C": 5},
    "G": {"H": 8},
    "H": {"C": 9}
}
```

3.1 Algorithme de Dijkstra

```
def dijkstra(graphe, depart):
    distances = {s: float("inf") for s in graphe}
    distances[depart] = 0
    predecesseurs = {s: None for s in graphe}
    non_traites = list(graphe.keys())

    while len(non_traites) > 0:
        sommet = min(non_traites,
                     key=lambda s: distances[s])
        non_traites.remove(sommet)

        for voisin, poids in graphe[sommet].items():
            nouvelle_dist = distances[sommet] + poids
            if nouvelle_dist < distances[voisin]:
                distances[voisin] = nouvelle_dist
                predecesseurs[voisin] = sommet

    return distances, predecesseurs
```

`float("inf")` est une valeur flottante plus grande que tout nombre. Elle représente ici une distance infinie : au départ, aucun sommet autre que `depart` n'est atteint.

3.2 Reconstitution du chemin

```
def plus_court_chemin(graphe, depart, arrivee):
    distances, predecesseurs = dijkstra(graphe, depart)
    chemin = []
    sommet = arrivee
    while sommet is not None:
        chemin.append(sommet)
        sommet = predecesseurs[sommet]
    chemin.reverse()
    return chemin, distances[arrivee]

>>> plus_court_chemin(graphe, "A", "H")
(['A', 'D', 'F', 'G', 'H'], 41)
```

④ Complexité

Avec l'implémentation ci-dessus (recherche du minimum par parcours linéaire), la complexité est $O(n^2)$ où n est le nombre de sommets.

On peut l'améliorer à $O((n + m) \log n)$ en utilisant un **tas** (*heap*). Un tas est une structure de données qui permet d'insérer un élément et d'extraire le minimum en temps $O(\log n)$, au lieu de $O(n)$ pour un parcours linéaire. Grâce à cette structure, la sélection du sommet de distance minimale, qui est l'opération coûteuse de chaque itération, devient beaucoup plus rapide. Cela permet d'appliquer Dijkstra à des graphes contenant des millions de sommets. Ici, m désigne le nombre d'arêtes. Cette optimisation est hors programme.

Applications : GPS (calcul d'itinéraire), RATP (recherche de trajet), protocole OSPF (routage réseau).