ALGORITHMES DES GRAPHES

À la fin de ce chapitre, je sais :

- ☞ parcourir un graphe en largeur et en profondeur
- ☞ démontrer la terminaison du parcours en largeur
- ☞ utiliser une file ou un pile pour parcourir un graphe
- ☞ énoncer le principe de l'algorithme de Dijkstra (plus court chemin)
- ☞ appliquer l'algorithme de Dijkstra à un graphe simple à la main en créant un tableau de résolution

A Parcours d'un graphe

■ **Définition 1 — Parcours d'un graphe.** Un parcours d'un graphe G est un ordre pour visiter chaque sommet d'un graphe.

Le parcours d'un graphe est une opération fondamentale : il ne s'agit pas tant de parcourir le graphe que de cheminer sur les sommets d'un graphe dans un certain ordre pour y effectuer des calculs. L'ordre du parcours est crucial dans le sens où il conditionne la plupart du temps la complexité de l'algorithme qui utilise un parcours. Les parcours de graphe sont utilisés par de nombreux algorithmes, notamment Dijkstra et A* mais également Edmonds-Karp.

On peut facilement mémoriser les différentes stratégies en observant les types d'ensemble qui sont utilisés pour stocker les sommets à parcourir au cours de l'algorithme :

1. Le parcours en **largeur** passe par tous les voisins d'un sommet avant de parcourir les descendants de ces voisins. Les sommets passent dans une **file** d'attente, c'est-à-dire structure de données de type First In First Out.
2. Le parcours en **profondeur** passe par tous les descendants d'un voisin d'un sommet avant de parcourir tous les autres voisins de ce sommet. L'implémentation la plus simple et la plus commune est récursive. Mais on peut aussi l'implémenter de manière itérative en utilisant une *pile* de sommets, c'est-à-dire une structure de données de type Last In First Out.

B Parcours en largeur

■ **Définition 2 — Parcours en largeur.** Parcourir en largeur un graphe signifie qu'on cherche à visiter tous les voisins d'un sommet avant de parcourir les autres sommets du graphe.

Vocabulary 1 — Breadth First Search ⇜⇝ Parcours en largeur

Le parcours en largeur d'un graphe (cf. algorithme 1) est un algorithme à la base de nombreux développements comme l'algorithme de Dijkstra et de Prim (cf. algorithmes 5 et **??**). Il nécessite d'utiliser :

1. une **file d'attente**[1] qui détient la liste des sommets à visiter dans l'ordre de la largeur du graphe : les sommets sont enfilés au fur et à mesure de leur découverte. Lorsque le sommet a été visité, il quitte la file.
2. et une structure qui **enregistre** les différents sommets déjà **découverts** afin de ne pas les revisiter.

Algorithme 1 Parcours en largeur d'un graphe

```
1 : Fonction PARCOURS_EN_LARGEUR(G, s)                    ▷ s est un sommet de G
2 :     F ← une file d'attente vide                        ▷ F comme file
3 :     D ← ∅                                              ▷ D ensemble des sommets découverts
4 :     P ← une liste vide                                 ▷ P comme parcours
5 :     ENFILER(F,s)
6 :     AJOUTER(D,s)          ▷ Pour la preuve de la correction, on précise que d[s] = 0
7 :     tant que F n'est pas vide répéter
8 :         v ← DÉFILER(F)
9 :         AJOUTER(P, v)                                  ▷ ou bien traiter le sommet en place
10 :        pour chaque voisin x de v dans G répéter
11 :            si x ∉ D alors                             ▷ x n'a pas encore été découvert
12 :                AJOUTER(D, x)
13 :                ENFILER(F, x)   ▷ Pour la preuve de la correction, on ajoute ici d[x] = d[u] + 1
14 :    renvoyer P                  ▷ Facultatif, on pourrait traiter chaque sommet v en place
```

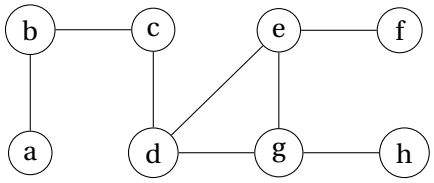

FIGURE 1 – Exemple de parcours en largeur au départ de a : a ⟶ b ⟶ c ⟶ d ⟶ e ⟶ g ⟶ f ⟶ h.

1. structure de données de type First In Fisrt Out

(R) La structure de donnée file permet de garantir la correction du parcours en largeur d'abord : on fait entrer en premier dans la file les voisins puis les descendants des voisins.

a Terminaison du parcours en largeur

La terminaison du parcours en largeur peut être prouvée en considérant le variant de boucle

$$v = |F| + |\overline{D}| \tag{1}$$

c'est-à-dire la somme des éléments présents dans la file et du nombre de sommets non découverts.

Démonstration. À l'entrée de la boucle, si n est l'ordre du graphe, on a $|F| + |\overline{D}| = 1 + n - 1 = n$. Puis, à chaque itération, on retire un élément de la file et on ajoute ses p voisins en même temps qu'on marque les p voisins comme découverts. L'évolution du variant au cours d'une itération s'écrit :

$$v = (|F|_d - 1 + p) + (|\overline{D}|_d - p) = |F|_d + |\overline{D}|_d - 1 \tag{2}$$

où l'on note $|F|_d$ et $|\overline{D}|_d$ les valeurs au début de l'itération de $|F|$ et $|\overline{D}|$.

À chaque tour de boucle, ce variant v décroit donc strictement de un et atteint nécessairement zéro au bout d'un certain nombre de tours. Lorsque le variant vaut zéro $|F| + |\overline{D}| = 0$, on a donc $|F| = 0$ et $|\overline{D}| = 0$. La file est nécessairement vide et tous les sommets ont été découverts. L'algorithme se termine puisque la condition de sortie est atteinte. ■

b Correction du parcours en largeur

Parcourir un graphe en largeur, à partir d'un sommet de départ s, cela veut dire trouver les chemins les plus courts partant de s vers tous les sommets du graphe [2]. On remarque que tous les sommets du graphe sont à un moment ou un autre de l'algorithme admis dans l'ensemble D et enfilé **une seule fois**.

Pour montrer la correction, on procède en deux temps :

parcours le parcours en largeur à partir du sommet de départ visite tous les sommets.

en largeur le chemin du sommet de départ s à un sommet u est le plus court en termes nombre de sauts [3].

Démonstration. On montre dans un premier temps que le parcours en largeur à partir du sommet de départ visite tous les sommets par l'absurde.

Imaginons qu'il existe un sommet x accessible depuis le sommet de départ s et non visité par l'algorithme du parcours en largeur. Supposons que x est le premier sommet accessible depuis s et non visité par l'algorithme. Si ce sommet est accessible depuis s, c'est qu'il existe un chemin de s à x dans le graphe. x possède donc un sommet adjacent parent sur ce chemin que l'on note p. Comme x est le premier sommet non visité, alors p a nécessairement été découvert

2. On fait l'hypothèse que le graphe est connexe. S'il ne l'est pas, il suffit de recommencer la procédure avec un des sommets n'ayant pas été parcouru.

3. c'est-à-dire en termes de longueur de chemin

précédemment par l'algorithme. Mais comme le parcours procède en largeur par sommets adjacents, si p a été découvert, alors x a été découvert également. Ce qui est absurde. Il n'existe donc pas de sommet x non visité par l'algorithme. **Donc tous les sommets sont visités.**

Puis il faut montrer qu'on parcourt les sommets selon l'ordre des voisins d'abord, c'est-à-dire dans l'ordre de la distance au sommet de départ. Le graphe n'est pas pondéré, mais on peut considérer que tous les poids du graphe valent 1 ou bien qu'on compte les sauts (franchissement des arêtes) pour atteindre un sommet. Il suffit d'ajouter la notion de distance comme précisé en commentaire sur l'algorithme 1.

On procède en utlisant l'invariant de boucle `while` : $\mathcal{I}$: **« Pour chaque sommet v qui se trouve dans la file, la distance d[v] au sommet de départ s est correcte et tous les sommets déjà découverts ont été découverts dans l'ordre croissant de distance depuis s. »**

Initialisation à l'entrée de la boucle `while`, le seul sommet dans la file est le sommet de départ qui est situé à une distance nulle de lui-même. La distance est correcte. Comme aucun autre sommet n'a encore été découvert, l'ordre est correct également. Donc $\mathcal{I}$ est vérifié à l'entrée de la boucle.

Conservation Supposons que $\mathcal{I}$ soit vérifié à l'entrée de la boucle. D'après cette hypothèse, la distance du sommet v défilé est correcte car v a été enfilé et marqué comme découvert à la précédente itération. À la fin de l'itération, on a marqué comme découvert tous les voisins de v qui se trouve à une distance de 1 de v. Ce sont les voisins les plus proches car ils sont directement connectés à v. Donc les sommets découverts le sont bien dans l'ordre croissant de distance. $\mathcal{I}$ est conservé par les instructions du corps de la boucle.

Conclusion Comme l'invariant est vérifié à l'entrée de la boucle et qu'il est conservé par les instructions de la boucle, $\mathcal{I}$ est vérifié à la fin de la boucle. Par conséquent, le parcours en largeur est correct : tous les sommets ont été découverts dans l'ordre croissant des distances.

Le parcours en largeur est donc correct. ■

c Complexité du parcours en largeur

La complexité de cet algorithme est lié, comme toujours, aux structures de données utilisées. Pour qu'elle soit optimale, on considère donc que :

1. G, un graphe d'ordre n et de taille m, est implémenté par une **liste d'adjacence** g. Rechercher les voisins d'un sommet est alors une opération en $O(1)$,
2. la file d'attente F possède une complexité en $O(1)$ pour les opérations ENFILER et DÉFILER,
3. les listes P et D possèdent une complexité en $O(1)$ pour l'ajout d'un élément AJOUTER,
4. savoir si un sommet est dans D est un opération en $O(1)$.

Selon ces options choisies, lors du parcours en largeur, les instructions des boucles s'effectuent donc en un temps constant c.

$$C(n) = \sum_{k=0}^{n-1} \left(1 + \sum_{v \in g[k]} c \right) \tag{3}$$

Dans le pire des cas, le graphe est complet : chaque sommet possède donc $n-1$ voisins et la boucle intérieur fait un nombre maximal d'itérations. La complexité dans le pire des cas est alors quadratique :

$$C(n) = \sum_{k=0}^{n-1} \left(1 + \sum_{v \in g[k]} c \right) = n + \sum_{k=0}^{n-1} \sum_{i=1}^{n-1} 1 = n + \frac{n(n-1)}{2} = O(n^2) \tag{4}$$

Si l'on considère un cas quelconque, c'est-à-dire un graphe non complet, on peut exprimer la complexité différemment. En déroulant les deux boucles, c'est-à-dire en écrivant les instructions les unes après les autres explicitement, on se rend compte que l'on parcours :

1. tous les sommets une fois et toutes les arêtes deux fois si le graphe n'est pas orienté,
2. tous les sommets une fois et toutes les arêtes une fois si le graphe est orienté.

Dans le cas d'un graphe non orienté, cela s'exprime :

$$C(n) = \sum_{k=0}^{n-1} \left(1 + \sum_{v \in g[k]} c \right) = n + 2 \sum_{a \in A} 1 = n + 2m = O(n+m) \tag{5}$$

C'est pourquoi on note généralement la complexité du parcours en largeur $O(n+m)$. C'est une expression qui montre que si le graphe est peu dense, alors la complexité du parcours n'est pas vraiment quadratique. Par contre, dans le pire des cas, le graphe est complet, $m = \frac{n(n-1)}{2}$ d'après le lemme des poignées de mains et on retrouve bien une complexité quadratique.

R Si on avait utilisé une matrice d'adjacence, on aurait été obligé de rechercher les voisins à chaque étape, c'est à dire de balayer une ligne de la matrice. Cette opération aurait eu un coût en $O(n)$. La complexité temporelle du parcours en serait impactée.

R Utiliser une liste Python pour implémenter une file d'attente n'est pas optimal. Si l'opération `append(x)` permet bien d'enfiler l'élément à la fin de la liste en une complexité $O(1)$, l'opération `pop(0)` qui permet de récupérer le premier élément de la liste (DÉFILER) est de complexité $O(n)$, car il est nécessaire de réécrire tout le tableau dynamique sous-jacent.

C Parcours en profondeur

■ **Définition 3 — Parcours en profondeur.** Parcourir en profondeur un graphe signifie qu'on cherche à visiter tous les voisins descendants d'un sommet qu'on découvre avant de parcourir les autres sommets découverts en même temps.

Vocabulary 2 — Depth First Search ⇜⇝ Parcours en profondeur

Le parcours en profondeur d'un graphe s'exprime naturellement récursivement (cf. algorithme 2). Il peut également s'exprimer de manière itérative (cf. algorithme 3) en utilisant une pile P afin gérer la découverte des voisins dans l'ordre de la profondeur du graphe.

Algorithme 2 Parcours en profondeur d'un graphe (version récursive)

```
1 : Fonction PARCOURS_EN_PROFONDEUR(G, s, D, C)          ▷ s est un sommet de G
2 :     AJOUTER(C, s)                                    ▷ s est ajouté au parcours du graphe
3 :     AJOUTER(D, s)                                    ▷ s est marqué découvert
4 :     pour chaque voisin x de s dans G répéter
5 :         si x ∉ D alors                               ▷ x n'a pas encore été découvert
6 :             PARCOURS_EN_PROFONDEUR(G, x, D, C)
```

Algorithme 3 Parcours en profondeur d'un graphe (version itérative)

```
1 : Fonction PARCOURS_EN_PROFONDEUR(G, s)                ▷ s est un sommet de G
2 :     P ← une pile vide                                ▷ P comme pile
3 :     D ← un ensemble vide                             ▷ D comme découverts
4 :     C ← un liste vide                                ▷ C pour le parcours
5 :     EMPILER(P,s)
6 :     tant que P n'est pas vide répéter
7 :         v ← DÉPILER(P)
8 :         AJOUTER(C, v)
9 :         pour chaque voisin x de v dans G répéter
10 :            si x ∉ E alors
11 :                EMPILER(P, x)
12 :                AJOUTER(D, x)
13 :    renvoyer C          ▷ Facultatif, on pourrait traiter chaque sommet v en place
```

La complexité de cet algorithme peut être calculé facilement si on considère son expression récursive (cf. algorithme 2). Soit un graphe $G = (S, A)$ possédant n sommets et m arêtes. On considère qu'en dehors des appels récursifs, la complexité de l'algorithme est constante. Par ailleurs, on effectue un nombre d'appels récursifs égal au nombre de voisin du sommet en cours d'exploration. C'est pourquoi, la complexité s'exprime :

$$T(S, A) = 1 + v_0 + T(S \setminus \{s_0\}, A \setminus \{a_0\}) \tag{6}$$

où s_0 désigne le sommet d'indice 0, v_0 le nombre de voisins du sommet s_0 et a_0 les arêtes de s_0

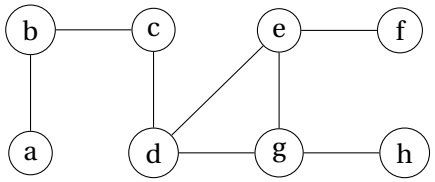

FIGURE 2 – Exemple de parcours en profondeur au départ de a : a ⟶ b ⟶ c ⟶ d ⟶ g ⟶ h ⟶ e ⟶ f

à ses voisins. On en déduit alors :

$$T(S,A) = 1 + v_0 + 1 + v_1 + T(S \setminus \{s_0, s_1\}, A \setminus \{a_0, a_1\}) \tag{7}$$
$$= 1 + 1 + 1 + \ldots + v_0 + v_1 + v_2 + \ldots + T(S \setminus \{s_0, s_1, s_2, \ldots\}, A \setminus \{a_0, a_1, a_2, \ldots\}) \tag{8}$$
$$= n + \sum_{k=0}^{m-1} v_k \tag{9}$$
$$= n + m \tag{10}$$

La complexité de l'algorithme du parcours en profondeur est donc la même que celles du parcours en largeur et on la note $O(n+m)$.

D Trouver un chemin dans un graphe non pondéré

On peut modifier l'algorithme 1 de parcours en largeur d'un graphe pour trouver un chemin reliant un sommet à un autre et connaître la longueur de la chaîne qui relie ces deux sommets. Il suffit pour cela de :

- garder la trace du prédécesseur (**parent**) du sommet visité sur le chemin,
- sortir du parcours dès qu'on a trouvé le sommet cherché (**early exit** ou sortie anticipée),
- calculer le coût du chemin associé.

Le résultat est l'algorithme 4.

Opérer cette recherche dans un graphe ainsi revient à chercher dans **toutes les directions**.

R Un graphe non pondéré peut-être vu comme un graphe pondéré dont la fonction de valuation vaut toujours 1. La distance entre deux sommets peut alors être interprétée comme le nombre de sauts nécessaires pour atteindre un sommet.

Algorithme 4 Longueur d'une chaîne via un parcours en largeur d'un graphe non pondéré

```
1 : Fonction LONGUEUR_CHAÎNE(G, s, b)          ▷ Trouver la longueur du chemin de s à b
2 :     F ← une file d'attente vide             ▷ F comme file
3 :     D ← un ensemble vide                    ▷ D comme découverts
4 :     P ← un dictionnaire vide                ▷ P comme parent
5 :     ENFILER(F,s)
6 :     AJOUTER(D,s)
7 :     tant que F n'est pas vide répéter
8 :         v ← DÉFILER(F)
9 :         si v = b alors                      ▷ Objectif atteint, early exit
10 :            sortir de la boucle
11 :        pour chaque voisin x de v dans G répéter
12 :            si x ∉ D alors                  ▷ x n'a pas encore été découvert
13 :                AJOUTER(D, x)
14 :                ENFILER(F, x)
15 :                P[x] ← v                    ▷ Garder la trace du parent de x sur le chemin
16 :    c ← 0                                   ▷ Longueur du chemin
17 :    u ← b
18 :    tant que u ≠ s répéter
19 :        c ← c + 1
20 :        sommet ← P[u]                       ▷ On remonte à l'origine du chemin
21 :    renvoyer c
```

E Plus courts chemins dans les graphes pondérés

Théorème 1 — Existence d'un plus court chemin. Dans un graphe pondéré **sans pondérations négatives**, il existe toujours un plus court chemin.

Démonstration. Un graphe pondéré possède un nombre fini de sommets et d'arêtes (cf. définition **??**). Il existe donc un nombre fini de chaînes entre les sommets du graphe. Comme les valuations du graphe ne sont pas négatives, c'est-à-dire que $\forall e \in E, w(e) \geqslant 0$, l'ensemble des longueurs de ces chaînes est **une partie non vide de $\mathbb{N}$: elle possède donc un minimum**. Parmi ces chaînes, il en existe donc nécessairement une dont la longueur est la plus petite, le plus court chemin. ■

■ **Définition 4 — Plus court chemin entre deux sommets d'un graphe.** Le plus court chemin entre deux sommets a et b d'un graphe G est une chaîne $\mathcal{C}_{ab}$ qui relie les deux sommets a et b et :

- qui comporte un minimum d'arêtes si G est un graphe non pondéré,
- dont le poids cumulé est le plus faible, c'est à dire $\min\limits_{\mathcal{C}_{ab} \in G} \left(\sum\limits_{e \in \mathcal{C}_{ab}} w(e) \right)$, dans le cas d'un graphe pondéré de fonction de valuation w.

■ **Définition 5 — Distance entre deux sommets.** La distance entre deux sommets d'un graphe est la longueur d'un plus court chemin entre ces deux sommets. Pour deux sommets a et b, on la note δ_{ab}. On a enfin :

$$\delta_{ab} = \min_{\mathcal{C}_{ab} \in G} \left(\sum_{e \in \mathcal{C}_{ab}} w(e) \right) \tag{11}$$

a Algorithme de Dijkstra

L'algorithme de Dijkstra [4] [**dijkstra_note_1959**] s'applique à des **graphes pondérés** $G = (S, A, w)$ **dont la valuation est positive**, c'est à dire que $\forall e \in E, w(e) \geqslant 0$. C'est un algorithme **glouton optimal** qui trouve les plus courts chemins entre un sommet particulier $s_0 \in S$ et tous les autres sommets d'un graphe. Pour cela, l'algorithme classe les différents sommets par ordre croissant de leur distance minimale au sommet de départ [5].

L'algorithme de Dijkstra est **un parcours en largeur qui utilise une file de priorités** : lorsqu'on insère un nouvel élément dans cette file, celui-ci est placé d'après son niveau de priorité, le plus prioritaire en premier. Dans notre cas, la priorité est la distance la plus faible : le sommet à la plus petite distance se situe donc en tête de la file et sera défiler en premier.

4. à prononcer "Daillekstra"
5. Ordre glouton

Algorithme 5 Algorithme de Dijkstra, plus courts chemins à partir d'un sommet donné

```
1 : Fonction DIJKSTRA(G = (S, A, w), s_0)        ▷ Trouver les plus courts chemins à partir de a ∈ S
2 :     Δ ← s_0                                  ▷ Δ est l'ensemble des sommets dont on connaît la distance à s_0
3 :     n ← |S|                                  ▷ L'ordre du graphe
4 :     Π ← tableau vide (taille n)              ▷ Π[s] est le parent de s dans le plus court chemin de s_0 à s
5 :     d ← tableau vide (taille n)              ▷ l'ensemble des distances au sommet s_0
6 :     ∀ s ∈ S, d[s] ← w(s_0, s)                ▷ w(s_0, s) = +∞ si s n'est pas voisin de s_0, 0 si s = s_0
7 :     tant que Δ̄ n'est pas vide répéter        ▷ Δ̄ : sommets dont la distance n'est pas connue
8 :         Choisir u dans Δ̄ tel que d[u] = min(d[v], v ∈ Δ̄)   ▷ Choix glouton!
9 :         Δ = Δ ∪ {u}                          ▷ On prend la plus courte distance à s_0 dans Δ̄
10 :        pour chaque voisin x de u répéter
11 :            si d[x] > d[u] + w(u, x) alors
12 :                d[x] ← d[u] + w(u, x)        ▷ Mises à jour des distances des voisins
13 :                Π[x] ← u                     ▷ Pour garder la tracer du chemin le plus court
14 :    renvoyer d, Π
```

R Il faut remarquer que les boucles imbriquées de cet algorithme peuvent être comprises comme deux étapes successives de la manière suivante :

1. On choisit un nouveau sommet u de G à chaque tour de boucle *tant que* qui est tel que $d[u]$ est la plus petite des valeurs accessibles dans $\bar{\Delta}$. **C'est le voisin d'un sommet de $\bar{\Delta}$ le plus proche de s_0.** Ce sommet u est alors inséré dans l'ensemble Δ : c'est la **phase de transfert** de u de $\bar{\Delta}$ à Δ.
2. Lors de la boucle *pour*, on met à jour les distances des voisins de u. C'est la **phase de mise à jour des distances** des voisins de u.

L'algorithme de Dijkstra procède de proche en proche, comme le parcours en largeur.

■ **Exemple 1 — Application de l'algorithme de Dijkstra.** On se propose d'appliquer l'algorithme 5 au graphe représenté sur la figure 3. Le tableau 1 représente les distances successivement trouvées à chaque tour de boucle *tant que* de l'algorithme. En rouge figurent les distances les plus courtes à a à chaque tour. On observe également que certaines distances sont mises à jour sans pour autant que le sommet soit sélectionné au tour suivant.

À la fin de l'algorithme, on note donc que les distances les plus courtes de a à b, c, d, e, f sont $[5, 1, 8, 3, 6]$. Le chemin le plus court de a à b est donc $a \to c \to e \to b$. Le plus court de a à f est $a \to c \to e \to f$. C'est la structure de données Π qui garde en mémoire le prédécesseur (parent) d'un sommet sur le chemin le plus court qui permettra de reconstituer les chemins.

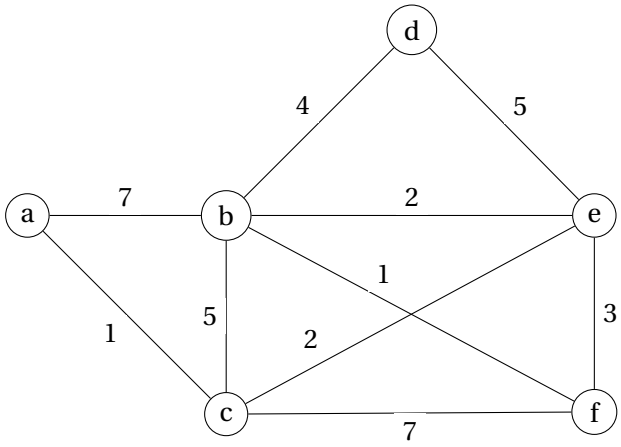

FIGURE 3 – Graphe pondéré à valeurs positives pour l'application de l'algorithme de Dijsktra.

Δ	a	b	c	d	e	f	$\bar{\Delta}$
{}	0	7	1	$+\infty$	$+\infty$	$+\infty$	$\{a,b,c,d,e,f\}$
$\{a\}$	.	7	1	$+\infty$	$+\infty$	$+\infty$	$\{b,c,d,e,f\}$
$\{a,c\}$	.	6	.	$+\infty$	3	8	$\{b,d,e,f\}$
$\{a,c,e\}$	.	5	.	8	.	6	$\{b,d,f\}$
$\{a,c,e,b\}$	.	.	.	8	.	6	$\{d,f\}$
$\{a,c,e,b,f\}$	.	.	.	8	.	.	$\{d\}$
$\{a,c,e,b,f,d\}$	.	.	.	.	.	.	{}

TABLE 1 – Tableau d des distances au sommet a successivement trouvées au cours de l'algorithme de Dijkstra appliqué au graphe de la figure 3

Théorème 2 — L'algorithme de Dijkstra se termine et est correct.

Démonstration. **Correction partielle de l'algorithme** : à chaque étape de cet algorithme, on peut distinguer deux ensembles de sommets : l'ensemble Δ est constitué des éléments dont on connaît la distance la plus courte à s_0 et l'ensemble complémentaire $\bar{\Delta}$ qui contient les autres sommets.

On note :

- δ_u la distance la plus courte du sommet s_0 au sommet u.
- $d[u]$ la distance trouvée par l'algorithme entre le sommet s_0 le sommet u.

On souhaite démontrer la correction en montrant que $\mathcal{I} : \forall x \in \Delta, d[x] = \delta_x$ est un invariant de boucle.

1. Avant la boucle : Δ ne contient que le sommet s_0. Or, $d[s_0] = 0$ et $\delta_{s_0} = 0$. Donc, l'invariant est vérifié à l'entrée de la boucle.

2. Pour une itération quelconque, on suppose que l'invariant $\mathcal{I}$ est vérifié à l'entrée de la boucle. Un sommet u est sélectionné dans $\bar{\Delta}$. Pour ce sommet u, qui n'appartient pas encore à Δ, on souhaite montrer qu'à la fin de l'itération $d[u] = \delta_u$.
   On procède par l'absurde en supposant que $d[u] \neq \delta_u$ et qu'il existe un plus court chemin P de s_0 à u tel que la longueur de ce chemin $\lambda(P)$ soit **strictement** plus petite que $d[u]$:

$$\lambda(P) < d[u]$$

   Ce chemin P démarre d'un sommet de Δ et le quitte au bout d'un certain temps pour atteindre u. Soit xy la première arête quittant Δ de ce chemin P : $x \in \Delta$ et $y \in \bar{\Delta}$. Soit P_x le chemin de s_0 à x. Ce chemin est un plus court chemin, par hypothèse d'induction et $d[x] = \delta_x$. On a donc :

$$\lambda(P_x) + w(x,y) = d[x] + w(x,y) \leqslant \lambda(P) < d[u] \tag{12}$$

   Comme y est un sommet adjacent à x, la distance à y a été mise à jour par l'algorithme précédemment. On a donc :

$$d[y] \leqslant d[x] + w(x,y) \tag{13}$$

   De plus, comme on a sélectionné u dans $\bar{\Delta}$ tel que la distance soit minimale au sommet de départ, et comme $y \in \bar{\Delta}$, on a également :

$$d[u] \leqslant d[y] \tag{14}$$

   En combinant ces équations, on aboutit à la contradiction suivante : $d[u] < d[u]$. Un chemin tel que P n'existe donc pas et $d[u] = \delta_u$. L'invariant $\mathcal{I}$ est donc vérifié à la fin de l'itération.
3. Comme l'invariant $\mathcal{I}$ est vérifié à l'entrée de la boucle et qu'il n'est pas modifié par les instructions de la boucle, on en déduit qu'il est vrai à la fin de la boucle. L'algorithme de Dijsktra est donc correct.

Terminaison de l'algorithme : avant la boucle *tant que*, $\bar{\Delta}$ possède $n-1$ éléments, si $n \in \mathbb{N}^\star$ est l'ordre du graphe. À chaque tour de boucle *tant que*, l'ensemble $\bar{\Delta}$ décroît strictement d'un élément et atteint donc nécessairement zéro. Le cardinal de $\bar{\Delta}$ est donc un variant de boucle. L'algorithme se termine lorsque le cardinal de $\bar{\Delta}$ atteint zéro. ■

La complexité de l'algorithme de Dijsktra dépend de l'ordre n du graphe considéré et de sa taille m. La boucle *tant que* effectue exactement $n-1$ tours. La boucle *pour* effectue à chaque fois un nombre de tour égal au nombre d'arêtes non découvertes qui partent du sommet u considéré et vont vers un sommet voisin de $\bar{\Delta}$. On ne découvre une arête qu'une seule fois, puisque le sommet u est transféré dans Δ au début de la boucle. Au final, on exécute donc la mise à jour des distances un nombre de fois égal à la taille m du graphe, c'est à dire son nombre d'arêtes. En notant la complexité du transfert c_t et la complexité de la mise à jour des distances c_d et en déroulant la boucle *tant que*, on peut écrire :

$$C(n,m) = (n-1)c_t + mc_d \tag{15}$$

Les complexités c_d et c_t dépendent naturellement des structures de données utilisées pour implémenter l'algorithme.

Si on choisit une implémentation de d par un tableau, alors on a besoin de rechercher le minimum des distances pour effectuer le transfert : cela s'effectue au prix d'un tri du tableau au minimum en $c_t = O(n \log n)$. Un accès aux éléments du tableau pour la mise à jour est en $c_d = O(1)$. On a donc $C(n) = (n-1)O(n \log n) + mO(1) = O(n^2 \log n)$.

Si d est implémentée par une file à priorités (un tas) comme le propose Johnson [**johnson_efficient_1977**], alors on a $c_t = O(\log n)$ et $c_d = O(\log n)$. La complexité est alors en $C(n) = (n+m)\log n$. Cependant, pour que le tas soit une implémentation pertinente, il est nécessaire que $m = O(\frac{n^2}{\log n})$, c'est à dire que le graphe ne soit pas complet, voire un peu creux!

■ **Exemple 2 — Usage de l'algorithme de Dijkstra .** Le protocole de routage OSPF implémente l'algorithme de Dijsktra. C'est un protocole qui permet d'automatiser le routage sur les réseaux internes des opérateurs de télécommunications. Les routeurs sont les sommets du graphe et les liaisons réseaux les arêtes. La pondération associée à une liaison entre deux routeurs est calculée à partir des performances en termes de débit de la liaison. Plus une liaison possède un débit élevé, plus la distance diminue.

OSPF est capable de relier des centaines de routeurs entre eux, chaque routeur relayant les paquets IP de proche en proche en utilisant le plus court chemin de son point de vue [a]. Le protocole garantit le routage des paquets par les plus courts chemins en temps réel. Chaque routeur calcule ses propres routes vers toutes les destinations, périodiquement. Si une liaison réseau s'effondre, le routeurs en sont informés et recalculent d'autres routes immédiatement. La puissance de calcul nécessaire pour exécuter l'algorithme sur un routeur, même dans le cas d'un réseau d'une centaine de routeur, est relativement faible car la plupart des réseaux de télécommunications sont des graphes relativement peu denses. Ce n'est pas rentable de créer des graphes de télécommunications complets, même si ce serait intéressant pour le consommateur et très robuste!

a. Cela fonctionne grâce au principe d'optimalité de Bellman!