

# ALGORITHMES ET GRAPHS

À la fin de ce chapitre, je sais :

- ✎ parcourir un graphe en largeur et en profondeur
- ✎ utiliser une file ou un pile pour parcourir un graphe
- ✎ énoncer le principe de l'algorithme de Dijkstra (plus court chemin)

## A Parcours d'un graphe

Le parcours d'un graphe est une opération fondamentale et utilisée par de nombreux algorithmes, notamment Dijkstra et A\*. 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** 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 de type Last In First Out.
3. L'algorithme de **Dijkstra** s'applique à un graphe pondéré : il passe par le voisin le plus proche d'un sommet avant de parcourir les autres voisins de ce sommet. C'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 : la plus petite distance se situe en tête de la file.

### a Parcours en largeur

Parcourir en largeur un graphe signifie qu'on cherche à visiter tous les voisins situé à une même distance d'un sommet (sur un même niveau) avant de parcourir le reste 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 utilise une file FIFO<sup>1</sup> afin de gérer la découverte des voisins dans l'ordre de la largeur du graphe.

Pour matérialiser le parcours en largeur, on opère en repérant les sommets à visiter. Lorsqu'un sommet est découvert, il intègre l'ensemble des éléments à visiter, c'est-à-dire la file  $F$ . Lorsque le sommet a été traité, il quitte la file. Il est donc également nécessaire de garder la trace du passage sur un sommet afin de ne pas traiter plusieurs fois un même sommet : si un sommet a été visité alors il intègre l'ensemble des éléments visités.

Au fur et à mesure de sa progression, cet algorithme construit un arbre de parcours en largeur dans le graphe. La racine de cet arbre est l'origine du parcours. Comme un sommet de cet arbre n'est découvert qu'une fois, il a au plus un parent. L'algorithme 1 peut ne rien renvoyer et servir pour un traitement particulier sur chaque nœud. Il peut aussi renvoyer une trace du parcours dans l'arbre dans une structure de type liste ( $P$ ) qui enregistre le parcours.

---

**Algorithme 1** Parcours en largeur d'un graphe

---

1 : <b>Fonction</b> PARCOURS_EN_LARGEUR( $G, s$ )	▷ $s$ est un sommet de $G$
2 : $F \leftarrow$ une file FIFO vide	▷ $F$ comme file FIFO
3 : $V \leftarrow \emptyset$	▷ $V$ ensemble des sommets visités
4 : $P \leftarrow$ une liste vide	▷ $P$ comme parcours
5 :   ENFILER( $F, s$ )	
6 :   AJOUTER( $V, s$ )	
7 : <b>tant que</b> $F$ n'est pas vide <b>répéter</b>	
8 : $v \leftarrow$ DÉFILER( $F$ )	
9 :     AJOUTER( $P, v$ )	▷ ou traiter le sommet en place
10 : <b>pour</b> chaque voisin $x$ de $v$ dans $G$ <b>répéter</b>	
11 : <b>si</b> $x \notin V$ <b>alors</b>	▷ $x$ n'a pas encore été découvert
12 :       AJOUTER( $V, x$ )	
13 :       ENFILER( $F, x$ )	
14 : <b>renvoyer</b> $P$	▷ Facultatif, on pourrait traiter chaque sommet $v$ en place

---

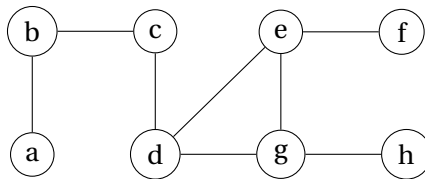


FIGURE 1 – Exemple de parcours en largeur au départ de  $a$  :  $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow g \rightarrow f \rightarrow h$ .

---

1. First In First Out

## b Terminaison et correction du parcours en largeur

La terminaison du parcours en largeur peut être prouvée en considérant le variant de boucle  $|F| + |\bar{V}|$ , c'est à dire la somme des éléments présents dans la file et du nombre de nœuds non visités. En effet, au début de la boucle, si  $n$  est l'ordre du graphe, on a  $|F| + |\bar{V}| = 1 + n - 1 = n$ . Puis, à chaque tour de boucle on retire un élément de la file et on ajoute ses  $p$  voisins en même temps qu'on marque les  $p$  voisins comme visités. L'évolution du variant s'écrit :

$$|F| - 1 + p + |\bar{V}| - p = |F| + |\bar{V}| - 1 \quad (1)$$

À chaque tour de boucle, le variant décroît donc strictement de un et atteint nécessairement zéro au bout d'un certain nombre de tours. Lorsque le variant vaut zéro  $|F| + |\bar{V}| = 0$ , on a donc  $|F| = 0$  et  $|\bar{V}| = 0$ . La file est nécessairement vide et tous les nœuds ont été visités. L'algorithme se termine alors. La structure de donnée file permet de garantir la correction du parcours en largeur d'abord.

Parcourir un graphe, à partir d'un sommet de départ  $s$ , cela veut dire trouver un chemin partant de  $s$  vers tous les sommets du graphe<sup>2</sup>. On remarque que tous les sommets du graphe sont à un moment ou un autre de l'algorithme **visités** et admis dans l'ensemble  $V$  : ceci vient du fait qu'on procède de proche en proche en ajoutant tous les voisins d'un sommet, sans distinction.

La correction peut se prouver en utilisant l'invariant de boucle  $\mathcal{I}$  : «Pour chaque sommet  $v$  ajouté à  $V$  et enfilé dans  $F$ , il existe un chemin de  $s$  à  $v$ .»

- Initialisation : à l'entrée de la boucle,  $s$  est ajouté à  $V$  et est présent dans la file  $F$ . Le chemin de  $s$  à  $s$  existe trivialement.
- Conservation : on suppose que l'invariant est vérifié jusqu'à une certaine itération. On cherche à montrer qu'il l'est toujours à la fin de l'itération suivante. Lors de l'exécution de cette itération, un sommet  $v$  est défilé. Ce sommet faisait déjà parti de  $V$  et par hypothèse, comme l'invariant était vérifié jusqu'à présent, il existe un chemin de  $s$  à  $v$ . Puis, les voisins de  $v$  sont ajoutés à  $V$  et enfilés. Comme ils sont voisins, il existe donc un chemin de  $v$  à ces sommets et donc il existe un chemin de  $s$  à ces sommets. À la fin de l'itération, l'invariant est donc vérifié.
- Terminaison : à la fin de l'algorithme, il existe un chemin de  $s$  vers tous les sommets du graphe visités (ajoutés à  $V$ ). Tous les sommets ont été parcourus.

## c Complexité du parcours en largeur

La complexité de cet algorithme est lié, comme toujours, aux structures de données utilisées. Soit  $G$  un graphe d'ordre  $n$  et de taille  $m$  implémenté par une liste d'adjacence. On a choisi une file FIFO pour laquelle les opérations ENFILER et DÉFILER sont en  $O(1)$ . On parcourt tous les sommets et chaque liste d'adjacence est parcourue une fois. Ces opérations sont donc en  $O(n + m)$ .

---

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.

**R** Utiliser une liste d'adjacence pour implémenter le graphe est très important dans ce cas car cela permet d'accéder rapidement aux voisins : le coût de cette opération est l'accès à un élément de la liste. Si on avait utilisé une matrice, on aurait été obligé de rechercher les voisins à chaque étape.

**R** Si le graphe est complet, on note que la complexité  $O(n + m)$  est en fait une complexité en  $O(n^2)$  car  $2|E| = n(n - 1)$  d'après le lemme des poignées de main (cf t??).

**R** Si l'on avait choisi un type tableau dynamique (typiquement le type `list` en Python) au lieu d'une file FIFO pour implémenter  $F$ , alors l'opération DÉFILER ferait perdre du temps : en effet, le tableau serait réécrit dans sa totalité à chaque fois qu'une opération DÉFILER aurait lieu car on retirerait alors le premier élément du tableau et il faudrait donc allouer un autre espace mémoire à ce nouveau tableau. Une fois encore, le choix de la structure de données est important pour que l'algorithme soit efficace.

**R** L'ensemble  $V$  n'est pas indispensable dans l'algorithme 1. On pourrait se servir de la liste qui enregistre le parcours. Néanmoins, son utilisation permet de bien découpler la sortie de l'algorithme (le chemin  $C$ ) de son fonctionnement interne et ainsi de prouver la terminaison.

## d Parcours en profondeur

Parcourir en profondeur un graphe signifie qu'on cherche à emprunter d'abord les arêtes du premier sommet trouvé avant de parcourir les voisins de ce sommet et le reste du graphe.

 **Vocabulary 2 — Depth First Search**  $\longleftrightarrow$  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 REC_PARCOURS_EN_PROFONDEUR( $G, s, V$ )                                 $\triangleright s$  est un sommet de  $G$ 
2:   AJOUTER( $V, s$ )                                                                 $\triangleright s$  est marqué visité
3:   pour chaque voisin  $x$  de  $s$  dans  $G$  répéter
4:     si  $x \notin V$  alors                                                             $\triangleright x$  n'a pas encore été découvert
5:       REC_PARCOURS_EN_PROFONDEUR( $G, x, V$ )

```

---

La complexité de cet algorithme est lié, comme toujours, aux structures de données utilisées. Soit  $G$  un graphe d'ordre  $n$  et de taille  $m$  implémenté par une liste d'adjacence. On a choisi une pile LIFO pour laquelle les opérations EMPILER et DÉPILER sont en  $O(1)$ . On parcourt tous les sommets et chaque liste d'adjacence est parcourue une fois. Ces opérations sont donc en

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

---

```

1: Fonction PARCOURS_EN_PROFONDEUR( $G, s$ )                                ▷  $s$  est un sommet de  $G$ 
2:    $P \leftarrow$  une file vide                                              ▷  $P$  comme pile
3:    $V \leftarrow$  un ensemble vide                                          ▷  $V$  comme visités
4:    $C \leftarrow$  un liste vide                                             ▷  $C$  pour le parcours
5:   EMPILER( $P, s$ )
6:   tant que  $P$  n'est pas vide répéter
7:      $v \leftarrow$  DÉPILER( $P$ )
8:     AJOUTER( $C, v$ )
9:     si  $v \notin V$  alors                                              ▷  $v$  n'a pas encore été découvert
10:      AJOUTER( $V, v$ )
11:      pour chaque voisin  $x$  de  $v$  dans  $G$  répéter
12:        EMPILER( $P, x$ )
13:   renvoyer  $C$                                                          ▷ Facultatif, on pourrait traiter chaque sommet  $v$  en place

```

---

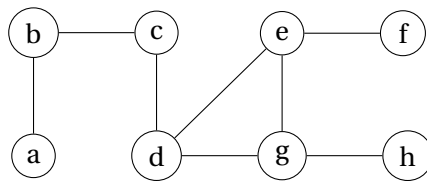


FIGURE 2 – Exemple de parcours en profondeur au départ de  $a$  :  $a \rightarrow b \rightarrow c \rightarrow d \rightarrow g \rightarrow h \rightarrow e \rightarrow f$

$O(n + m)$ .

## B Trouver un chemin dans un graphe

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),
- 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, c'est à dire sans tenir compte des distances déjà parcourues.

**R** Il faut noter néanmoins que le chemin trouvé et la distance associée issue de l'algorithme 4 n'est pas nécessairement la meilleure, notamment car on ne tient pas compte de la distance parcourue jusqu'au sommet recherché.

---

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

---

```

1: Fonction CD_PEL( $G, a, b$ )                                ▷ Trouver un chemin de  $a$  à  $b$  et la distance associée
2:    $F \leftarrow$  une file FIFO vide                               ▷  $F$  comme file FIFO
3:    $V \leftarrow$  un ensemble vide                                 ▷  $V$  comme visités
4:    $P \leftarrow$  un dictionnaire vide                             ▷  $P$  comme parent
5:   ENFILER( $F, s$ )
6:   AJOUTER( $V, s$ )
7:   tant que  $F$  n'est pas vide répéter
8:      $v \leftarrow$  DÉFILER( $F$ )
9:     si  $v$  est le sommet  $b$  alors                                ▷ Objectif atteint, early exit
10:      sortir de la boucle
11:     pour chaque voisin  $x$  de  $v$  dans  $G$  répéter
12:       si  $x \notin V$  alors                                       ▷  $x$  n'a pas encore été découvert
13:         AJOUTER( $V, x$ )
14:         ENFILER( $F, x$ )
15:          $P[x] \leftarrow v$                                        ▷ Garder la trace du parent
16:    $c \leftarrow 0$                                                ▷ Coût du chemin
17:    $s \leftarrow a$ 
18:   tant que  $s$  est différent de  $b$  répéter
19:      $c \leftarrow c + w(s, P[s])$                                 ▷  $w$  est la fonction de valuation du graphe  $G$ 
20:     sommet  $\leftarrow P[s]$                                        ▷ On remonte à l'origine du chemin
21:   renvoyer  $c$ 

```

---

Ⓡ Si le graphe n'est pas pondéré, cet algorithme fonctionne néanmoins, il suffit de compter le nombre de sauts pour évaluer la distance (la fonction de valuation vaut toujours 1).

## C Plus courts chemins dans les graphes pondérés

**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.

**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  $\forall e \in E, w(e) \geq 0$ , 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 1 — 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  $\min_{\mathcal{C}_{ab} \in G} \left( \sum_{e \in \mathcal{C}_{ab}} w(e) \right)$ , dans le cas d'un graphe pondéré de fonction de valuation  $w$ .

■ **Définition 2 — 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  $\delta_{ab}$ . On a enfin :

$$\delta_{ab} = \min_{\mathcal{C}_{ab} \in G} \left( \sum_{e \in \mathcal{C}_{ab}} w(e) \right) \quad (2)$$

On se propose maintenant d'étudier les algorithmes les plus célèbres qui illustrent, dans différentes configurations, le concept de plus court chemin dans un graphe. La majorité de ces algorithmes reposent sur le principe d'optimalité de Bellman et la programmation dynamique et qui a déjà été énoncé dans le théorème ??. On peut formuler ce principe ainsi : **toute sous-chaîne entre  $p$  et  $q$  d'un plus court chemin entre  $a$  et  $b$  est un plus court chemin entre  $p$  et  $q$ .**

### a Algorithme de Dijkstra

L'algorithme de Dijkstra<sup>3</sup> [dijkstra\_note\_1959] s'applique à des **graphes pondérés**  $G = (V, E, w)$  **dont la valuation est positive**, c'est à dire que  $\forall e \in E, w(e) \geq 0$ . C'est un algorithme glouton op-

3. à prononcer "Daillekstra"



timal (cf. informatique commune) qui trouve les plus courts chemins entre un sommet particulier  $a \in V$  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. Dans ce but, il **parcourt en largeur le graphe en choisissant les voisins les plus proches en premier**.

---

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

---

```

1: Fonction DIJKSTRA( $G = (V, E, w), a$ )    ▷ Trouver les plus courts chemins à partir de  $a \in V$ 
2:    $\Delta \leftarrow a$                         ▷  $\Delta$  est l'ensemble des sommets dont on connaît la distance à  $a$ 
3:    $\Pi \leftarrow \emptyset$                     ▷  $\Pi[s]$  est le parent de  $s$  dans le plus court chemin de  $a$  à  $s$ 
4:    $d \leftarrow \emptyset$                     ▷ l'ensemble des distances au sommet  $a$ 
5:    $\forall s \in V, d[s] \leftarrow w(a, s)$         ▷  $w(a, s) = +\infty$  si  $s$  n'est pas voisin de  $a$ , 0 si  $s = a$ 
6:   tant que  $\bar{\Delta}$  n'est pas vide répéter    ▷  $\bar{\Delta}$  : sommets dont la distance n'est pas connue
7:     Choisir  $u$  dans  $\bar{\Delta}$  tel que  $d[u] = \min(d[v], v \in \bar{\Delta})$ 
8:      $\Delta = \Delta \cup \{u\}$                 ▷ On prend la plus courte distance à  $a$  dans  $\bar{\Delta}$ 
9:     pour  $x \in \bar{\Delta}$  répéter              ▷ Ou bien  $x \in \mathcal{N}_G(u) \cap \bar{\Delta}$ , pour tous les voisins de  $u$  dans  $\bar{\Delta}$ 
10:      si  $d[x] > d[u] + w(u, x)$  alors
11:         $d[x] \leftarrow d[u] + w(u, x)$     ▷ Mises à jour des distances des voisins
12:         $\Pi[x] \leftarrow u$                 ▷ Pour garder la tracer du chemin le plus court
13:   renvoyer  $d, \Pi$ 

```

---

**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  $a$ .** Ce sommet  $u$  est alors inséré dans l'ensemble  $\Delta$  : c'est la **phase de transfert** de  $u$  de  $\bar{\Delta}$  à  $\Delta$ .
2. Lors de la boucle *pour*, on met à jour les distances des voisins de  $u$  qui n'ont pas encore été découverts. En effet, si  $x$  n'est pas un voisin de  $u$ , alors il n'existe pas d'arête entre  $u$  et  $x$  et  $w(u, x) = +\infty$ . La mise à jour de la distance n'a donc pas lieu, on n'a pas trouvé une meilleure distance à  $x$ . C'est la **phase de mise à jour des distances** des voisins de  $u$ .

L'algorithme de Dijkstra procède donc de proche en proche.

■ **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  $a \rightarrow c \rightarrow e \rightarrow b$ . Le plus court de  $a$  à  $f$  est  $a \rightarrow c \rightarrow e \rightarrow f$ . C'est la structure de données  $\Pi$  qui garde en mémoire le prédécesseur (parent) d'un sommet sur le chemin le plus court qui permettra de reconstituer les chemins.

$\Delta$	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



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

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

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

D'après le principe d'optimalité, tout chemin plus court vers un sommet de  $\bar{\Delta}$  passera nécessairement par un sommet de  $\Delta$ . Ceci s'écrit :

$$\forall u \in \bar{\Delta}, d[u] = \min (d[v] + w[v, u], v \in \Delta) \quad (3)$$

On souhaite montrer qu'à la fin de chaque tour de boucle tant que (lignes 6-12),  $d$  contient les distances les plus courtes vers tous les sommets de  $\Delta$ . On peut formuler cet invariant de boucle.

J : à chaque fin de tour de boucle on a

$$\forall u \in \Delta, d[u] = \delta_{au} \quad (4)$$

$$\forall u \in \bar{\Delta}, d[u] = \min (d[v] + w[v, u], v \in \Delta) \quad (5)$$

À l'entrée de la boucle, l'ensemble  $\Delta$  ne contient que le sommet de départ  $a$ . On a  $d[a] = 0$ , ce qui est la distance minimale. Pour les autres sommets de  $\bar{\Delta}$ ,  $d$  contient :

- une valeur infinie si ce sommet n'est pas un voisin de  $a$ , ce qui, à cette étape de l'algorithme est le mieux qu'on puisse trouver,
- le poids de l'arête venant de  $a$  s'il s'agit d'un voisin, ce qui, à cette étape de l'algorithme est le mieux que l'on puisse trouver également.

On peut donc affirmer que  $d$  contient les distances entre  $a$  et tous les sommets de  $\Delta$ . L'invariant est vérifié à l'entrée de la boucle.

On se place maintenant à une étape quelconque de la boucle. Notre hypothèse  $\mathcal{H}$  est que toutes les itérations précédentes sont correctes. À l'entrée de la boucle on sélectionne un sommet  $u$ , le premier de la file de priorités. Il nous faut alors montrer que  $d[u] = \delta_{au}$ .

$u$  entre dans  $\Delta$ , c'est à dire que  $u \in \bar{\Delta}$  et  $\forall v \in \bar{\Delta}, d[u] \leq d[v]$ . Considérons un autre chemin de  $a$  à  $u$  passant par un sommet  $v$  de  $\bar{\Delta}$ . Comme on a  $d[u] \leq d[v]$ , cet autre chemin sera au moins aussi long que  $d[u]$ , sauf s'il existe des arêtes de poids négatif (ce qui n'est pas le cas).

Formellement, on peut écrire cela ainsi

$$\delta_{au} = \delta_{av} + \delta_{vu} \quad (6)$$

$$\delta_{au} \geq \delta_{av} \quad (7)$$

Par ailleurs, comme  $v$  appartient à  $\bar{\Delta}$ , il vérifie l'hypothèse d'induction. On a donc :

$$d[v] = \min (d[x] + w[x, v], x \in \Delta) \quad (8)$$

$$= \min (\delta_{ax} + w[x, v], x \in \Delta) \quad (9)$$

$$= \delta_{av} \quad (10)$$

la deuxième ligne étant obtenue grâce à l'hypothèse d'induction également.

$$d[u] \leq d[v] = \delta_{av} \quad (11)$$

$$\leq \delta_{av} \quad (12)$$

$$\leq \delta_{au} \quad (13)$$

Or,  $d[u]$  ne peut pas être plus petit que la distance de  $a$  à  $u$ . On a donc finalement  $d[u] = \delta_{au}$ .

$d$  contient donc les distances vers tous les sommets à la fin de l'exécution de l'algorithme.

Terminaison de l'algorithme : avant la boucle *tant que*,  $\bar{\Delta}$  possède  $n - 1$  éléments, si  $n \in \mathbb{N}^*$  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 Dijkstra 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  $\Delta$  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 :

$$C(n, m) = (n - 1)c_t + mc_d \quad (14)$$

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 Dijkstra. C'est un protocole qui permet d'automatiser le routage sur les réseaux internes des opérateurs de télécommunication. 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 la 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<sup>a</sup>. 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 creux. Ce n'est pas rentable de créer des graphes de télécommunications complets, même si ce serait intéressant pour le consommateur et très robuste!

<sup>a</sup>. Cela fonctionne grâce au principe d'optimalité de Bellman!

## b Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford calcule les plus courts chemins depuis un sommet de départ, comme l'algorithme de Dijkstra. Cependant, il s'applique à des **graphes pondérés et orientés**

dont les pondérations peuvent être négatives mais sans cycles de longueur négative [bellman\_routing\_1958, ford\_jr\_network\_1956, moore\_shortest\_1959].

**(R)** Soit un graphe pondéré non orienté qui possède des poids négatifs. Comme les arêtes d'un graphe non orienté sont des cycles, on ne peut pas appliquer l'algorithme de Bellman-Ford. On lui préférera alors Dijkstra.

**(R)** Les poids négatifs peuvent représenter des transferts de flux (chaleur en chimie, argent en économie) et sont donc très courants.

---

**Algorithme 6** Algorithme de Bellman-Ford, plus courts chemins à partir d'un sommet donné

---

```

1: Fonction BELLMAN_FORD( $G = (V, E, w), a$ )
2:    $\Pi \leftarrow$  un dictionnaire vide       $\triangleright \Pi[s]$  est le parent de  $s$  dans le plus court chemin de  $a$  à  $s$ 
3:    $d \leftarrow$  ensemble des distances au sommet  $a$ 
4:    $d[s] \leftarrow w(a, s)$                  $\triangleright w(a, s) = +\infty$  si  $s$  n'est pas voisin de  $a$ , 0 si  $s = a$ 
5:   pour _ de 1 à  $|V| - 1$  répéter           $\triangleright$  Répéter  $n - 1$  fois
6:     pour  $(u, v) = e \in E$  répéter       $\triangleright$  Pour toutes les arêtes du graphe
7:       si  $d[v] > d[u] + w(u, v)$  alors     $\triangleright$  Si le chemin est plus court par là...
8:          $d[v] \leftarrow d[u] + w(u, v)$      $\triangleright$  Mises à jour des distances des voisins
9:          $\Pi[v] \leftarrow u$                $\triangleright$  Pour garder la tracer du chemin
10:  renvoyer  $d, \Pi$ 

```

---

■ **Exemple 3 — Application de l'algorithme de Bellman-Ford.** On se propose d'appliquer l'algorithme 6 au graphe pondéré et orienté représenté sur la figure 4. On note qu'il contient une pondération négative de  $b$  à  $f$  mais pas de cycle à pondération négative. Le tableau 2 représente les distances successivement trouvées à chaque itération.

On observe que le chemin de  $a$  à  $f$  emprunte bien l'arc de pondération négative.

Il faut noter que l'algorithme a convergé avant la fin de l'itération dans cas. C'est un des axes d'amélioration de cet algorithme.

N° d'itération	a	b	c	d	e	f
1	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
2	0	5	1	8	3	6
3	0	5	1	8	3	4
4	0	5	1	8	3	4
5	0	5	1	8	3	4

TABLE 2 – Tableau  $d$  des distances au sommet  $a$  successivement trouvées au cours de l'algorithme de Bellman-Ford appliqué au graphe de la figure 4



FIGURE 4 – Graphe pondéré et orienté à valeurs positives et négatives pour l'application de l'algorithme de Bellman-Ford.

La complexité de l'algorithme 6 est en  $O(nm)$  si  $n$  est l'ordre du graphe et  $m$  sa taille.

■ **Exemple 4 — Protocole de routage RIP.** Le protocole de routage RIP utilise l'algorithme de Bellman-Ford pour trouver les plus courts chemins dans un réseau de routeur. Il est moins adapté que OSPF pour les grands réseaux.

### c Algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall [floyd\_algorithm\_1962, roy\_transitivite\_1959, warshall\_theorem\_1962] est l'application de la programmation dynamique à la recherche de **l'existence d'un chemin entre toutes les paires de sommets d'un graphe orienté et pondéré**. Les distances trouvées sont les plus courtes. Les pondérations du graphe peuvent être négatives mais on exclue tout circuit de poids strictement négatif.

Soit un graphe orienté et pondéré  $G = (V, E, w)$ .  $G$  peut être modélisé par une matrice d'adjacence  $M$

$$\forall i, j \in \llbracket 0, |V| - 1 \rrbracket, M = \begin{cases} w(v_i, v_j) & \text{si } (v_i, v_j) \in E \\ +\infty & \text{si } (v_i, v_j) \notin E \\ 0 & \text{si } i = j \end{cases} \quad (15)$$

Un exemple de graphe associé à la matrice d'adjacence :

$$M_{\text{init}} = \begin{pmatrix} 0 & 8 & +\infty & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & +\infty & 0 & +\infty \\ +\infty & 2 & 9 & 0 \end{pmatrix} \quad (16)$$

est donné sur la figure 5. Sur cet exemple, le chemin le plus court de  $v_4$  à  $v_3$  vaut 3 et passe par  $v_2$ .

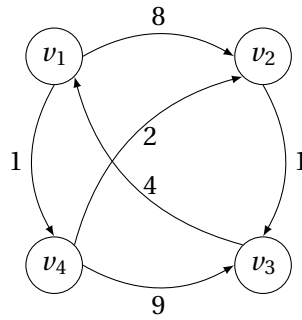


FIGURE 5 – Exemple de graphe orienté et pondéré pour expliquer le concept de matrice d'adjacence.

Pour trouver le plus court chemin entre deux sommets, on essaye tous les chemins de toutes les longueurs possibles et on ne garde que les plus courts. Chaque étape  $p$  de l'algorithme de Floyd-Warshall est donc constitué d'un allongement **éventuel** du chemin par le sommet  $v_p$ . À l'étape  $p$ , on associe une matrice  $M_p$  qui contient la longueur des chemins les plus courts d'un sommet à un autre passant par des sommets de l'ensemble  $\{v_0, v_1, \dots, v_p\}$ . On construit ainsi une suite de matrice finie  $(M_p)_{p \in \llbracket 0, n-1 \rrbracket}$  et on initialise la matrice  $M$  avec avec  $M_{\text{init}}$ .

Supposons qu'on dispose de  $M_{p-1}$ . Considérons un chemin  $\mathcal{C}$  entre  $v_i$  et  $v_j$  dont la longueur est minimale et dont les sommets intermédiaires sont dans  $\{v_0, v_1, \dots, v_{p-1}\}$ ,  $p \leq n$ . Pour un tel chemin :

- soit  $\mathcal{C}$  passe par  $v_p$ . Dans ce cas,  $\mathcal{C}$  est la réunion de deux chemins dont les sommets sont dans  $\{v_0, v_1, \dots, v_p\}$  : celui de  $v_i$  à  $v_p$  et celui de  $v_p$  à  $v_j$ .
- soit  $\mathcal{C}$  ne passe pas par  $v_p$ .

Entre ces deux chemins, on choisira le chemin le plus court.

Disposer d'une formule de récurrence entre  $M_p$  et  $M_{p-1}$  permettrait de montrer que le problème du plus court chemin entre deux sommets d'un graphe orienté et pondéré est à sous-structure optimale. On pourrait alors utiliser la programmation dynamique pour résoudre le problème. Or, on peut traduire notre explication ci-dessus par la relation de récurrence suivante :

$$\forall p \in \llbracket 0, n-1 \rrbracket, \forall i, j \in \llbracket 0, n-1 \rrbracket, M_p(i, j) = \min(M_{p-1}(i, j), M_{p-1}(i, p) + M_{p-1}(p, j)) \quad (17)$$

Pour  $p = 0$ , on pose  $M_{-1} = M_{\text{init}}$ .

L'algorithme de Floyd-Warshall 7 n'est que le calcul de la suite de ces matrices. C'est un bel exemple de programmation dynamique.

**R** Cet algorithme effectue le même raisonnement que Bellman-Ford mais avec une vision globale, à l'échelle du graphe tout entier, pas uniquement par rapport à un sommet de départ.

**Algorithme 7** Algorithme de Floyd-Warshall, plus courts chemins entre toutes les paires de sommet

---

```

1: Fonction FLOYD_WARSHALL( $G = (V, E, w)$ )
2:    $M \leftarrow$  la matrice d'adjacence de  $G$ 
3:   pour  $p$  de 0 à  $|V| - 1$  répéter
4:     pour  $i$  de 0 à  $|V| - 1$  répéter
5:       pour  $j$  de 0 à  $|V| - 1$  répéter
6:          $M(i, j) = \min(M(i, j), M(i, p) + M(p, j))$ 
7:   renvoyer  $M$ 

```

---

■ **Exemple 5 — Application de l'algorithme de Floyd-Warshall.** Si on applique l'algorithme au graphe de la figure 5, alors on obtient la série de matrices suivantes :

$$M_0 = \begin{pmatrix} 0 & 8 & +\infty & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & +\infty & 0 & +\infty \\ +\infty & 2 & 9 & 0 \end{pmatrix} \quad (18)$$

$$M_1 = \begin{pmatrix} 0 & 8 & +\infty & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & 12 & 0 & 5 \\ +\infty & 2 & 9 & 0 \end{pmatrix} \quad (19)$$

$$M_2 = \begin{pmatrix} 0 & 8 & 9 & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & 12 & 0 & 5 \\ +\infty & 2 & 3 & 0 \end{pmatrix} \quad (20)$$

$$M_3 = \begin{pmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{pmatrix} \quad (21)$$

$$M_4 = \begin{pmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{pmatrix} \quad (22)$$

#### d A\*

L'algorithme A\*<sup>4</sup> est un algorithme couteau suisse qui peut être considéré comme un algorithme de Dijkstra muni d'une heuristique : là où Dijkstra ne tient compte que du coût du chemin déjà parcouru, A\* considère ce coût et une heuristique qui l'informe sur le reste du chemin à parcourir. Il faut bien remarquer que le chemin qu'il reste à parcourir n'est pas nécessai-

---

4. prononcer *A étoile* ou *A star*



rement déjà exploré : parfois il est même impossible d'explorer tout le graphe. Si l'heuristique pour évaluer le reste du chemin à parcourir est bien choisie, alors A\* converge aussi vite voire plus vite que Dijkstra[unswmechatronics\_dijkstras\_2013].

■ **Définition 3 — Heuristique admissible.** Une heuristique  $\mathcal{H}$  est admissible si pour tout sommet du graphe,  $\mathcal{H}(s)$  est une borne inférieure de la plus courte distance séparant le sommet de départ du sommet d'arrivée.

■ **Définition 4 — Heuristique cohérente.** Une heuristique  $\mathcal{H}$  est cohérente si pour tout arête  $(s, p)$  du graphe  $G = (V, E, w)$ ,  $\mathcal{H}(s) \leq \mathcal{H}(p) + w(s, p)$ .

■ **Définition 5 — Heuristique monotone.** Une heuristique  $\mathcal{H}$  est monotone si l'estimation du coût **total** du chemin ne décroît pas lors du passage d'un sommet à ses successeurs. Pour un chemin  $(s_0, s_1, \dots, s_n)$ , on  $\forall 0 \leq i < j \leq n, c(s_j) \geq c(s_i)$ .

Soit  $G = (V, E, w)$  un graphe orienté. Soit  $d$  la fonction de distance utilisée par l'algorithme de Dijkstra (cf. algorithme 5). A\*, muni d'une fonction  $h$  permettant d'évaluer l'heuristique, calcule alors le coût total pour aller jusqu'à un sommet  $p$  comme suit :

$$c(p) = d(p) + h(p) \quad (23)$$

Le coût obtenu n'est pas nécessairement optimal, il dépend de l'heuristique.

Supposons que l'on cherche le chemin le plus court entre les sommets  $s_0$  et  $p$ . Supposons que l'on connaisse un chemin optimal entre  $s_0$  et un sommet  $s$ . Alors on peut écrire que le coût total vers le sommet  $p$  vaut :

$$c(p) = d(p) + h(p) \quad (24)$$

$$= d(s) + w(s, p) + h(p) \quad (25)$$

$$= d(s) + h(s) + w(s, p) - h(s) + h(p) \quad (26)$$

$$= c(s) + w(s, p) - h(s) + h(p) \quad (27)$$

Ainsi, on peut voir l'algorithme A\* comme un algorithme de Dijkstra muni :

- de la distance  $\tilde{d} = c$ ,
- et de la pondération  $\tilde{w}(s, p) = w(s, p) - h(s) + h(p)$ .

L'algorithme 8 donne le détail de la procédure à suivre.

---

**Algorithme 8 A\***


---

```

1: Fonction ASTAR( $G = (V, E, w), a$ ) ▷ Sommet de départ  $a$ 
2:    $\Delta \leftarrow a$ 
3:    $\Pi \leftarrow$ 
4:    $\tilde{d} \leftarrow$  l'ensemble des distances au sommet  $a$ 
5:    $\forall s \in V, \tilde{d}[s] \leftarrow \tilde{w}(a, s)$  ▷ Le graphe est partiel, l'heuristique fait le reste
6:   tant que  $\bar{\Delta}$  n'est pas vide répéter ▷  $\bar{\Delta}$  : sommets dont la distance n'est pas connue
7:     Choisir  $u$  dans  $\bar{\Delta}$  tel que  $\tilde{d}[u] = \min(\tilde{d}[v], v \in \bar{\Delta})$ 
8:      $\Delta = \Delta \cup \{u\}$ 
9:     pour  $x \in \bar{\Delta}$  répéter ▷ Ou bien  $x \in \mathcal{N}_G(u) \cap \bar{\Delta}$ , pour tous les voisins de  $u$  dans  $\bar{\Delta}$ 
10:      si  $\tilde{d}[x] > \tilde{d}[u] + \tilde{w}(u, x)$  alors
11:         $\tilde{d}[x] \leftarrow \tilde{d}[u] + \tilde{w}(u, x)$ 
12:         $\Pi[x] \leftarrow u$  ▷ Pour garder la tracer du chemin le plus court
13:   renvoyer  $\tilde{d}, \Pi$ 

```

---