ALGORITHMES ET GRAPHES

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)
- ☞ expliquer l'intérêt d'un tri topologique
- ☞ schématiser le concept de forte connexité
- ☞ expliquer ce qu'est un arbre recouvrant
- ☞ expliquer l'intérêt d'un graphe biparti

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. Les sommets passent dans une **pile** de type Last In First Out.
3. L'algorithme de **Dijkstra** 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 petite distance en tête donc.

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 développement comme l'algorithme de Dijkstra et de Prim (cf. algorithmes 5 et 9). Il utilise une file FIFO [1] 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 (C) qui enregistre le chemin parcouru.

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 FIFO vide                            ▷ F comme file FIFO
3 :     V ← ∅                                             ▷ V ensemble des sommets visités
4 :     C ← une liste vide                                ▷ C comme chemin
5 :     ENFILER(F,s)
6 :     AJOUTER(V,s)
7 :     tant que F n'est pas vide répéter
8 :         v ← DÉFILER(F)
9 :         AJOUTER(C, v)                                 ▷ ou traiter le sommet en place
10 :        pour chaque voisin x de v dans G répéter
11 :            si x ∉ V alors                            ▷ x n'a pas encore été découvert
12 :                AJOUTER(V, x)
13 :                ENFILER(F, x)
14 :    renvoyer C          ▷ Facultatif, on pourrait traiter chaque sommet v en place
```

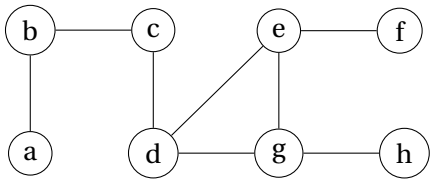


FIGURE 1 – Exemple de parcours en largeur au départ de a : a ⟶ b ⟶ c ⟶ d ⟶ e ⟶ g ⟶ f ⟶ h.

1. First In Fisrt Out

b Terminaison et correction du parcours en largeur

La terminaison du parcours en largeur peut être prouvée en considérant le variant de boucle $|F| + |\overline{V}|$, 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 $|F| + |\overline{V}| = 1 + n - 1 = n$. 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 :

$$|F| - 1 + p + |\overline{V}| - p = |F| + |\overline{V}| - 1 \tag{1}$$

À chaque tour de boucle, le variant décroit donc strictement de un et atteint nécessairement zéro au bout d'un certain nombre de tours. Lorsque le variant vaut zéro $|F| + |\overline{V}| = 0$, on a donc $|F| = 0$ et $|\overline{V}| = 0$. 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[2]. 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 ou 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.
- Hérédité : 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.
- Conclusion : 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 parcours 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 un 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 ⇜⇝ Parcours en profondeur

Le parcours en profondeur d'un graphe (cf. algorithme 2) est un algorithme qui utilise une pile P afin gérer la découverte des voisins dans l'ordre de la profondeur du graphe. Il peut aussi s'exprimer récursivement (cf. algorithme 3).

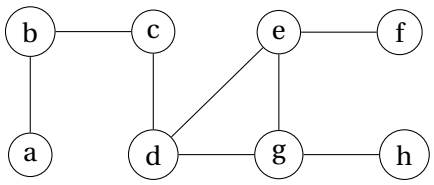


FIGURE 2 – Exemple de parcours en profondeur au départ de a : a ⟶ b ⟶ c ⟶ d ⟶ g ⟶ h ⟶ e ⟶ f

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

Algorithme 2 Parcours en profondeur d'un graphe

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

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

```
1 : Fonction REC_PARCOURS_EN_PROFONDEUR(G, s, V)             ▷ s est un sommet de G
2 :     AJOUTER(V, s)                                        ▷ s est marqué visité
3 :     pour chaque voisin x de s dans G répéter
4 :         si x ∉ V alors                                   ▷ x n'a pas encore été découvert
5 :             REC_PARCOURS_EN_PROFONDEUR(G, x, V)
```

une pile LIFO pour laquelle les opérations EMPILER et DÉPILER sont en $O(1)$. On parcours tous les sommets et chaque liste d'adjacence est parcourue une fois. Ces opérations sont donc en $O(n+m)$.

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.

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 ← une file FIFO vide                                          ▷ F comme file FIFO
3 :     V ← un ensemble vide                                            ▷ V comme visités
4 :     P ← 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 ← 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 ∉ V alors                                          ▷ x n'a pas encore été découvert
13 :                AJOUTER(V, x)
14 :                ENFILER(F, x)
15 :                P[x] ← v                                            ▷ Garder la trace du parent
16 :    c ← 0                                                           ▷ Coût du chemin
17 :    s ← b
18 :    tant que s est différent de a répéter
19 :        c ← c + w(s, P[s])                  ▷ w est la fonction de valuation du graphe G
20 :        sommet ← P[s]                                               ▷ On remonte à l'origine du chemin
21 :    renvoyer c
```

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.

R 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 $\forall e \in E, w(e) \geqslant 0$, 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 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 $\min\limits_{\mathcal{C}_{ab} \in G} \left(\sum\limits_{e \in \mathcal{C}_{ab}} w(e) \right)$, dans le cas d'un graphe pondéré de fonction de valuation w.

■ **Définition 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 δ_{ab}. On a enfin :

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

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[3] [**dijkstra_note_1959**] s'applique à des **graphes pondérés** $G = (V, E, w)$ **dont la valuation est positive**, c'est à dire que $\forall e \in E, w(e) \geqslant 0$. 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 ∈ V
2 :     Δ ← a                                  ▷ Δ est l'ensemble des sommets dont on connaît la distance à a
3 :     Π ← ∅                                  ▷ Π[s] est le parent de s dans le plus court chemin de a à s
4 :     d ← ∅                                  ▷ l'ensemble des distances au sommet a
5 :     ∀s ∈ V, d[s] ← w(a, s)                 ▷ w(a, s) = +∞ si s n'est pas voisin de a, 0 si s = a
6 :     tant que Δ̄ n'est pas vide répéter      ▷ Δ̄ : sommets dont la distance n'est pas connue
7 :         Choisir u dans Δ̄ tel que d[u] = min(d[v], v ∈ Δ̄)
8 :         Δ = Δ ∪ {u}                        ▷ On prend la plus courte distance à a dans Δ̄
9 :         pour x ∈ Δ̄ répéter                 ▷ Ou bien x ∈ N_G(u) ∩ Δ̄, pour tous les voisins de u dans Δ̄
10 :            si d[x] > d[u] + w(u, x) alors
11 :                d[x] ← d[u] + w(u, x)      ▷ Mises à jour des distances des voisins
12 :                Π[x] ← u                   ▷ Pour garder la tracer du chemin le plus court
13 :    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 a.** 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 qui n'ont pas encore été découverts. En effet, si x n'est pas un voisin de u, alors il n'existe par 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'algoritme 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 $a \to c \to e \to b$. 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.

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

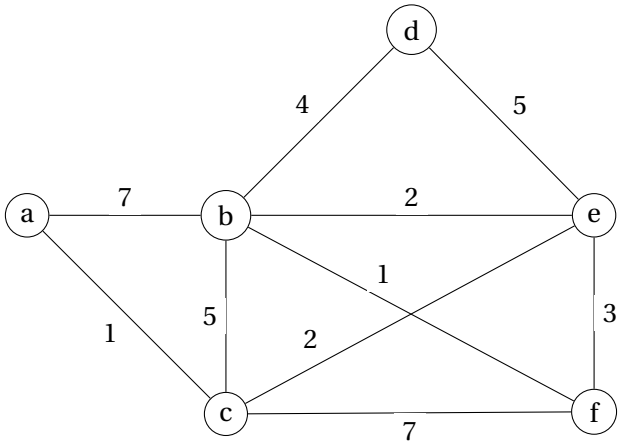


FIGURE 3 – Graphe pondéré à valeurs positives pour l'application de l'algorithme de Dijsktra.

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 Δ 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 Δ. Ceci s'écrit :

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

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 Δ. On peut formuler cet invariant de boucle.

$\mathcal{I}$: à chaque fin de tour de boucle on a

$$\forall u \in \Delta, d[u] = \delta_{au} \tag{4}$$
$$\forall u \in \bar{\Delta}, d[u] = \min(d[v] + w[v,u], v \in \Delta) \tag{5}$$

À l'entrée de la boucle, l'ensemble Δ 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 Δ. 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 alors montrer que $d[u] = \delta_{au}$.

u entre dans Δ, c'est à dire que $u \in \bar{\Delta}$ et $\forall v \in \bar{\Delta}, d[u] \leqslant d[v]$. Considérons un autre chemin de a à u passant par un sommet v de $\bar{\Delta}$. Comme on a $d[u] \leqslant d[v]$, 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

$$\delta_{au} = \delta_{av} + \delta_{vu} \tag{6}$$
$$\delta_{au} \geqslant \delta_{av} \tag{7}$$

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

$$d[v] = \min(d[x] + w[x,v], x \in \Delta) \tag{8}$$
$$= \min(\delta_{ax} + w[x,v], x \in \Delta) \tag{9}$$
$$= \delta_{av} \tag{10}$$

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

$$d[u] \leqslant d[v] = \delta_{av} \tag{11}$$
$$\leqslant \delta_{av} \tag{12}$$
$$\leqslant \delta_{au} \tag{13}$$

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 $n \in \mathbb{N}^\star$ 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 :

$$C(n,m) = (n-1)c_t + mc_d \tag{14}$$

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 Dijsktra. 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 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 creux. 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!

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**].

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 :     Π ← un dictionnaire vide        ▷ Π[s] est le parent de s dans le plus court chemin de a à s
3 :     d ← ensemble des distances au sommet a
4 :     d[s] ← w(a, s)                  ▷ w(a, s) = +∞ si s n'est pas voisin de a, 0 si s = a
5 :     pour _ de 1 à |V| − 1 répéter                          ▷ Répéter n − 1 fois
6 :         pour (u, v) = e ∈ E répéter                        ▷ Pour toutes les arêtes du graphe
7 :             si d[v] > d[u] + w(u, v) alors                 ▷ Si le chemin est plus court par là...
8 :                 d[v] ← d[u] + w(u, v)                      ▷ Mises à jour des distances des voisins
9 :                 Π[v] ← u                                   ▷ Pour garder la tracer du chemin
10 :    renvoyer d, Π
```

■ **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.

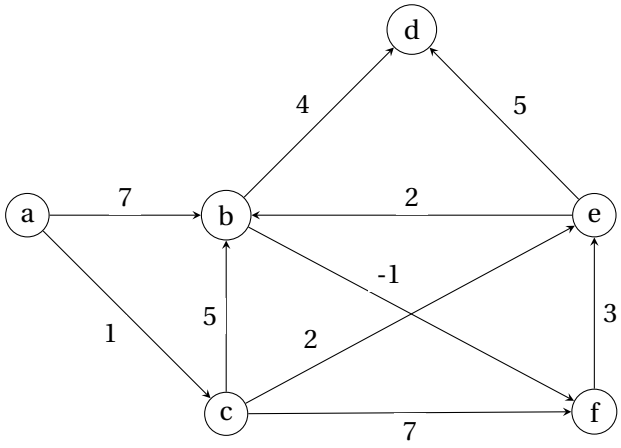


FIGURE 4 – Graphe pondéré et orienté à valeurs positives et négatives pour l'application de l'algorithme de Bellman-Ford.

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

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

■ **Exemple 4 — Protocole de routage RIP.** Le protocole de routage RIP utilise l'algorithme de Bellman-Ford pour trouver les plus cours 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 du **plus court chemin entre toutes les paires de sommets d'un graphe orienté et pondéré**. 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

$$\forall i, j \in [\![1, |\, V \,|]\!], M = \begin{cases} w(v_i, v_j) & \text{si } (v_i, v_j) \in E \\ +\infty & \text{si } (v_i, v_j) \notin E \\ 0 & \text{si } i = j \end{cases} \tag{15}$$

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

$$M = \begin{pmatrix} 0 & 8 & +\infty & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & +\infty & 0 & +\infty \\ +\infty & 2 & 9 & 0 \end{pmatrix} \tag{16}$$

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.

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 $\{v_1, v_2, \ldots, v_p\}$. On construit ainsi une suite de matrice finie $\left(M_p\right)_{p \in [\![0,n]\!]}$ avec $M_0 = M$.

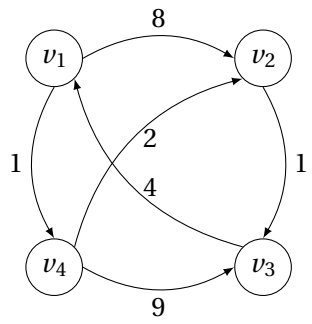


FIGURE 5 – Exemple de graphe orienté et pondéré pour expliquer le concept de matrice d'adjacence.

Supposons qu'on dispose de M_p. Considérons un chemin $\mathcal{C}$ entre v_i et v_j dont la longueur est minimale et dont les sommets intermédiaires sont dans $\{v_1, v_2, \ldots, v_{p+1}\}$, $p \leqslant n$. Pour un tel chemin :

- soit $\mathcal{C}$ passe par v_{p+1}. Dans ce cas, $\mathcal{C}$ est la réunion de deux chemins dont les sommets sont dans $\{v_1, v_2, \ldots, v_{p+1}\}$: celui de v_i à v_{p+1} et celui de v_{p+1} à v_j.
- soit $\mathcal{C}$ ne passe pas par v_{p+1}.

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

Disposer d'une formule de récurrence entre M_{p+1} et M_p 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 :

$$\forall p \in [\![1, n]\!], \forall i, j \in [\![1, n]\!], M_{p+1}(i, j) = \min\big(M_p(i, j), M_p(i, p+1) + M_p(p+1, j)\big) \tag{17}$$

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.

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 ← la matrice d'adjacence de G
3 :     pour p de 1 à |V| répéter
4 :         pour i de 1 à |V| répéter
5 :             pour i de 1 à |V| répéter
6 :                 M(i, j) = min(M(i, j), M(i, p+1) + M(p+1, j))
7 :     renvoyer M
```

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

■ **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 :

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

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

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

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

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

d A*

L'algorithme A*[4] 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 nécessairement 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.

4. prononcer *A étoile* ou *A star*

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

■ **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 $(s_0, s_1, \ldots, s_n)$, on $\forall 0 \leqslant i < j \leqslant n, c(s_j) \geqslant c(s_i)$.

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 :

$$c(p) = d(p) + h(p) \tag{23}$$

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 :

$$\begin{aligned} c(p) &= d(p) + h(p) && (24)\\ &= d(s) + w(s,p) + h(p) && (25)\\ &= d(s) + h(s) + w(s,p) - h(s) + h(p) && (26)\\ &= c(s) + w(s,p) - h(s) + h(p) && (27) \end{aligned}$$

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 :     Δ ← a
3 :     Π ←
4 :     d̃ ← l'ensemble des distances au sommet a
5 :     ∀s ∈ V, d̃[s] ← w̃(a, s)          ▷ Le graphe est partiel, l'heuristique fait le reste
6 :     tant que Δ̄ n'est pas vide répéter          ▷ Δ̄ : sommets dont la distance n'est pas connue
7 :         Choisir u dans Δ̄ tel que d̃[u] = min(d̃[v], v ∈ Δ̄)
8 :         Δ = Δ ∪ {u}
9 :         pour x ∈ Δ̄ répéter          ▷ Ou bien x ∈ N_G(u) ∩ Δ̄, pour tous les voisins de u dans Δ̄
10 :            si d̃[x] > d̃[u] + w̃(u, x) alors
11 :                d̃[x] ← d̃[u] + w̃(u, x)
12 :                Π[x] ← u                    ▷ Pour garder la tracer du chemin le plus court
13 :    renvoyer d̃, Π
```

D Arbres recouvrants

Les arbres recouvrants sont des éléments essentiels de l'industrie, notamment du point de vue de l'efficacité, de la robustesse et de l'optimisation. En ce sens, ils sont également indispensables à toute vision durable de notre développement. L'idée d'un arbre recouvrant est de ne sélectionner que certaines arêtes dans un graphe pondéré afin de garantir un service optimale (en fonction des pondérations). Les arbres recouvrants sont donc utilisés dans les domaines des réseaux d'énergie, des télécommunications, des réseaux de fluides mais également en intelligence artificielle et en électronique. Les deux algorithmes phares pour construire des arbres recouvrants sont l'algorithme de Kruskal [**kruskal_shortest_1956**] et l'algorithme de Prim [**prim_shortest_1957**].

a Algorithme de Prim

L'algorithme de Prim est un algorithme glouton optimal qui s'applique aux graphes pondérés connexes. Pour construire l'arbre, l'algorithme part d'un sommet et fait croitre l'arbre en choisissant un sommet dont la distance est la plus faible et n'appartenant pas à l'arbre, garantissant ainsi l'absence de cycle.

Algorithme 9 Algorithme de Prim, arbre recouvrant

```
1 : Fonction PRIM(G = (V, E, w))
2 :     T ← ∅                                  ▷ la sortie : l'ensemble des arêtes de l'arbre recouvrant
3 :     S ← s un sommet quelconque de V
4 :     tant que S ≠ V répéter
5 :         (u, v) ← min (w(u, v), u ∈ S, v ∈ E)                          ▷ Choix glouton !
6 :         S ← S ∪ {v}
7 :         T ← T ∪ {(u, v)}
8 :     renvoyer T
```

La complexité de cet algorithme, si l'on utilise un tas binaire, est en $O(m \log n)$.

b Algorithme de Kruskal

L'algorithme de Kruskal (cf. algorithme 10) est un algorithme glouton optimal qui s'applique aux graphes pondérés. Le graphe peut ne pas être connexe et dans ce cas on obtient un forêt d'arbres recouvrants. Pour construire la forêt et effectuer un choix d'arête, il ordonne les arêtes du graphe par ordre de pondération croissante.

Si on utilise une structure de tas ainsi La complexité de cet algorithme est en $O(m \log m)$ si $m = |E|$ est le nombre d'arêtes du graphe.

Algorithme 10 Algorithme de Kruskal, arbre recouvrant

```
1 : Fonction KRUSKAL(G = (V, E, w))
2 :     T ← ∅                          ▷ la sortie : l'ensemble des arêtes de l'arbre recouvrant
3 :     pour k de 1 à |E| répéter
4 :         e ← l'arête de pondération la plus faible de E          ▷ Choix glouton!
5 :         si (S, T ∪ {e}, w) est un graphe acyclique alors
6 :             T ← T ∪ {e}
7 :     renvoyer T
```

E Tri topologique d'un graphe orienté

a Ordre dans un graphe orienté acyclique

Dans un graphe orienté (cf. définition **??**) acyclique, les arcs définissent un **ordre partiel**, le sommet à l'origine de l'arc pouvant être considéré comme le prédécesseur du sommet à l'extrémité de l'arc. Par exemple, sur la figure 6, a et b sont des prédécesseurs de d et e est un prédécesseur de g. Mais ces arcs ne disent rien de l'ordre entre e et h, l'ordre n'est pas total.

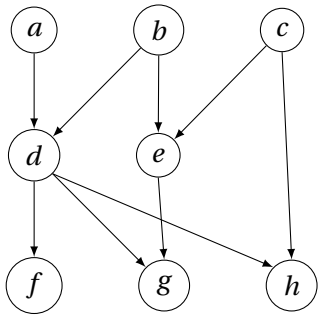


FIGURE 6 – Exemple de graphe orienté acyclique

L'algorithme de tri topologique permet de créer un ordre total $\preceq$ sur un graphe orienté acyclique. Formulé mathématiquement :

$$\forall (v,u) \in V^2, (v,u) \in E \Longrightarrow v \preceq u \tag{28}$$

Sur l'exemple de la figure **??**, plusieurs ordre topologiques sont possibles. Par exemple :

- a,b,c,d,e,f,g,h
- a,b,d,f,c,h,e,g

b Tri topologique et détection de cycles dans un graphe orienté

L'algorithme de tri topologique permet de construire un ordre dans un graphe orienté acyclique. C'est en fait un parcours en profondeur du graphe qui construit une pile en ajoutant le

concept de date à chaque sommet : un date de début qui correspond au début du traitement du sommets et une date de fin qui correspond à la fin du traitement du sommet par l'algorithme. La pile contient à la fin les sommets dans un ordre topologique, les sommets par ordre de date de fin de traitement.

Au cours du parcours en profondeur, un sommet passe tout d'abord de l'ensemble des sommets non traités à l'ensemble des sommets en cours de traitement (date de début). Puis, lorsque la descente est finie (plus aucun arc ne sort du sommet courant), le sommet passe de l'ensemble en cours de traitement à l'ensemble des sommets traités (date de fin).

Algorithme 11 Algorithme de tri topologique

```
1 : Fonction TOPO_SORT(G = (V, E))
2 :     pile ← une pile vide
3 :     état ← un tableau des états des nœuds        ▷ pas traité, en cours de traitement ou traité
4 :     Les case du tableau état sont initialisées à «pas traité»
5 :     date ← 0                                                          ▷ Date initiale
6 :     pour chaque sommet v de G répéter
7 :         si v n'est pas traité alors
8 :             TOPO_DFS(G = (V, E), pile, v, état, date)
9 :     renvoyer pile
10 : Fonction TOPO_DFS(G = (V, E), pile, v, état, date)
11 :    etat[v] ← «en cours de traitement»
12 :    Date de début de v ← date                          ▷ Par forcément nécessaire
13 :    pour chaque voisin u de v répéter
14 :        si u n'est pas traité alors TOPO_DFS(G = (V, E), pile, u, état, (date + 1))
15 :    etat[v] ← «traité»
16 :    Incrémenter la date de fin de v                    ▷ Par forcément nécessaire
17 :    EMPILER(v, pile)
```

F Composantes fortement connexes d'un graphe orienté et 2-SAT

■ **Définition 6 — Composante fortement connexe d'un graphe orienté** $G = (V, E)$. Une composante fortement connexe d'un graphe orienté G est un sous-ensemble S de ses sommets, maximal au sens de l'inclusion, tel que pour tout couple de sommets $(s, t) \in S$ il existe un chemin de s à t dans G.

Le calcul des composantes connexes d'un graphe est par exemple utilisé pour résoudre le problème 2-SAT. Dans le cadre de ce problème, on dispose d'une formule logique sous la forme conjonctive normale et chaque clause comporte deux variables. L'idée est de construire un graphe à partir de la formule de cette formule F. Supposons qu'elle soit constituée de m clauses et n variables $(v_1, v_2, \ldots, v_n)$. On élabore alors un graphe $G = (V, E)$ à $2n$ sommets et $2m$ arêtes. Les sommets représentent les n variables v_i ainsi que leur négation $\neg v_i$. Les arêtes sont construites de la manière suivante : on transforme chaque clause de F de la forme $v_i \vee v_j$ en

deux implications $\neg v_1 \Longrightarrow v_2$ ou $\neg v_2 \Longrightarrow v_1$. Cette transformation utilise le fait que la formule $a \Longrightarrow b$ est équivalent à $\neg a \vee b$.

Théorème 3 F n'est pas satisfaisable si et seulement s'il existe une composante fortement connexe contenant une variable v_i et sa négation $\neg v_i$.

On peut montrer que SAT-2 est un problème de décision polynomial. C'est une restriction à des clauses de deux variables du problème général SAT qui lui est NP-complet.

G Graphes bipartis et couplage maximum

Théorème 4 — Caractérisation des graphes bipartis. Un graphe est biparti si et seulement s'il ne possède aucun cycle de longueur impair.

Pour savoir si un graphe est biparti on pourrait donc rechercher les cycles et évaluer leurs longueurs. Il existe cependant une autre approche. Un graphe biparti est un graphe bicolorable comme le montre la figure **??**.

a Couplage dans un graphe biparti

■ **Définition 7 — Couplage.** Un couplage Γ dans un graphe non orienté $G = (V, E)$ est un ensemble d'arêtes deux à deux non adjacentes. Formellement :

$$\forall (e_1, e_2) \in E^2, e_1 \neq e_2 \Longrightarrow e_1 \cap e_2 = \emptyset \tag{29}$$

c'est à dire que les sommets de e_1 et e_2 ne sont pas les mêmes.

■ **Définition 8 — Sommets couplés, sommets exposés.** Un sommet est couplé s'il fait parti d'une arête de Γ. Un sommet est exposé s'il ne fait pas parti des arêtes de Γ, c'est à dire il n'est pas couplé.

■ **Définition 9 — Couplage maximal.** Un couplage maximal est tel que si on lui ajoute une arête, ce n'est plus un couplage. Il est donc maximal au sens de l'inclusion.

■ **Définition 10 — Couplage de cardinal maximum.** Un couplage de cardinal maximum est un couplage contenant le plus grand nombre d'arêtes possible.

■ **Exemple 6 — Affectation des cadeaux sous le sapin .** Au pied du sapin de Noël, un papa a disposé six cadeaux dont les paquets sont tous différents et numérotés de 0 à 5 [a]. Il a décidé que les cadeaux seraient répartis en fonction des paquets que les enfants préfèrent.

Ses cinq enfants expriment donc leurs préférences. Le papa pourra-t-il affecter un cadeau à chaque enfant et faire en sorte que ce cadeau soit un de leurs préférés ?

Évidemment la réponse à cette question dépend des préférences émises par les enfants.

Supposons qu'ils se soient exprimés ainsi :

Alix 0,2

Brieuc 1,3,4,5

Céline 1,2

Dimitri 0,1,2

Enora 2

On peut représenter par un graphe biparti cette situation comme sur la figure 7. Dans ce cas précis, comme il y a quatre enfants qui ne veulent que trois les trois premiers cadeaux, il n'y a pas de solution. Mais si Enora avait choisit 4 et 5 ???

a. Ce papa est informaticien!

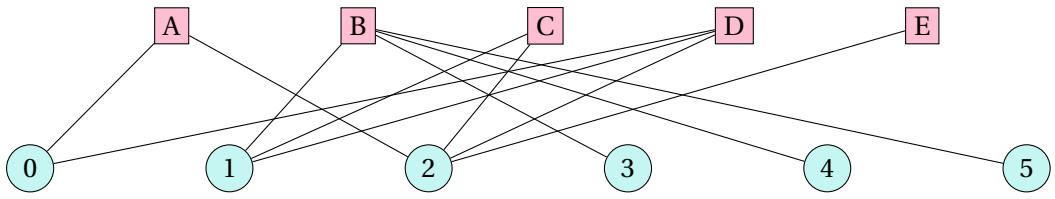


FIGURE 7 – Exemple de graphe biparti pour un problème d'affectation sans solution.

R Si les arêtes du graphe sont valuées (graphe pondéré), alors la recherche d'un couplage de cardinal maximum et de poids minimum dans un graphe biparti est en fait le problème de l'**affectation des ressources**. Ceci n'est pas au programme.

b Chemin augmentant

Pour résoudre le problème de trouver un couplage de cardinal maximum, on peut procéder en suivant l'algorithme 12. Il s'agit de construire un chemin augmentant pour atteindre un couplage de cardinal maximum.

■ **Définition 11 — Chemin alternant.** Une chemin alternant dans un graphe non orienté G et pour un couplage Γ est tel que les arêtes appartiennent successivement à Γ et $E \setminus \Gamma$.

■ **Définition 12 — Chemin augmentant.** Un chemin augmentant est un chemin alternant dont les extrémités sont des sommets exposés, c'est à dire qui n'appartiennent pas au couplage Γ.

La stratégie de l'algorithme de recherche d'un couplage de cardinal maximum est la suivante : à partir d'un couplage Γ, on construit un nouveau couplage de cardinal supérieur à l'aide d'un chemin augmentant comme le montre la figure 8.

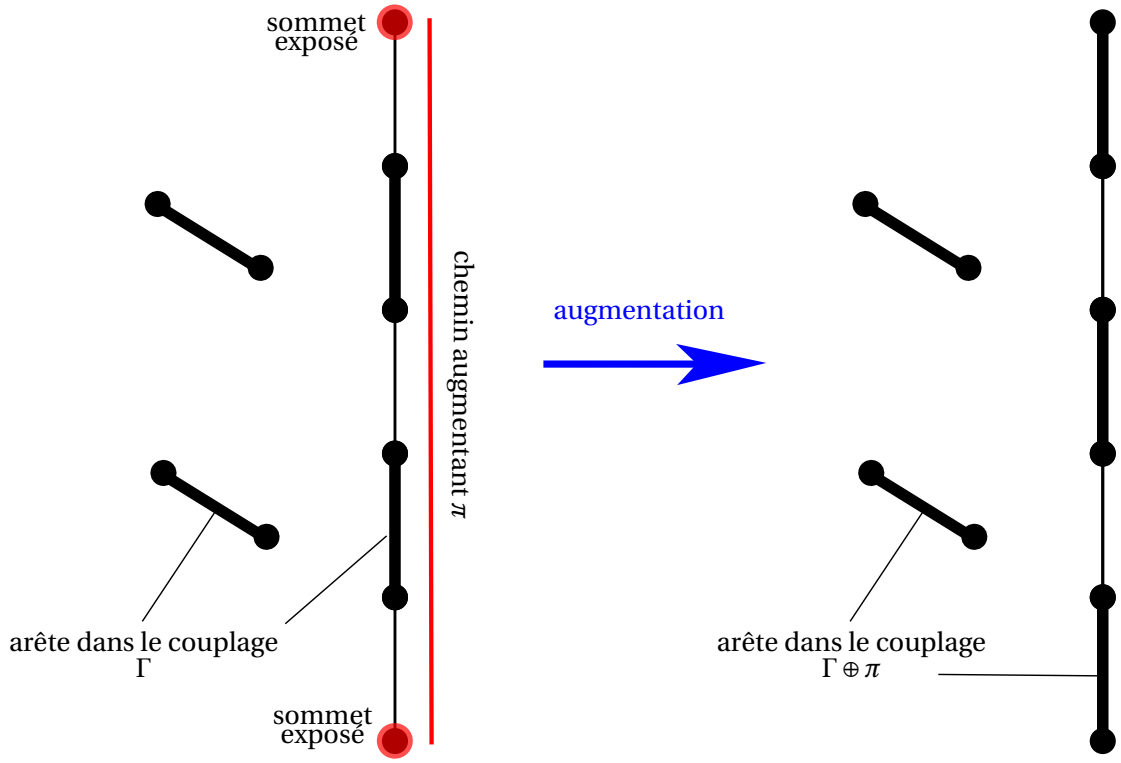


FIGURE 8 – Illustration de la construction d'un couplage de cardinal supérieur grâce à un chemin augmentant. (Source originale : Wikimedia Commons)

Dans un graphe **biparti**, il est facile d'augmenter la taille d'un couplage jusqu'au cardinal maximum :

1. s'il existe deux sommets exposés reliés par une arête, il suffit d'ajouter cette arête au couplage. Puis, on appelle récursivement l'algorithme sur ce nouveau couplage.
2. sinon il faut trouver un chemin augmentant π dans le graphe.
   (a) s'il n'y en a pas, l'algorithme est terminé.
   (b) sinon on effectue la différence symétrique entre le couplage Γ et l'ensemble des arêtes du chemin augmentant π pour obtenir le nouveau couplage : $\Gamma \oplus \pi = \{e \in E, e \in \Gamma \setminus \pi \cup e \in \pi \setminus \Gamma\}$. Puis, on appelle récursivement l'algorithme avec ce nouveau couplage.

Il faut noter que le cardinal du couplage n'augmente pas nécessairement lorsqu'on effectue la différence symétrique mais il ne diminue pas.

Pour trouver un chemin augmentant dans un graphe biparti $G = ((U, D), E)$, on cherche le plus court chemin entre deux sommets exposés dans un graphe orienté auxiliaire G_o construit de la manière suivante :

1. toutes les arêtes de E qui n'appartiennent pas au couplage Γ sont orientés de U vers D.
2. toutes les arêtes de Γ sont orientées de D vers U.

Le plus court chemin entre deux sommets exposés de G_o est nécessairement un chemin augmentant, son caractère alternant vient du fait que le graphe est biparti.

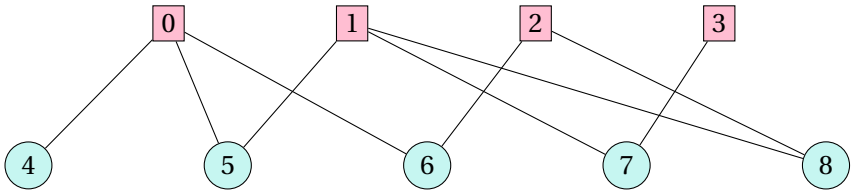
La figure 9 illustre les différentes étapes de l'algorithme. On vérifie bien que le résultat est correct : chaque enfant aura bien un cadeau qu'il avait mis sur sa liste.

Algorithme 12 Recherche d'un couplage de cardinal maximum

Entrée : un graphe biparti $G = ((U, D), E)$
Entrée : un couplage Γ initialement vide
Entrée : F_U, l'ensemble de sommets exposés de U initialement U
Entrée : F_D, l'ensemble de sommets exposés de D initialement D

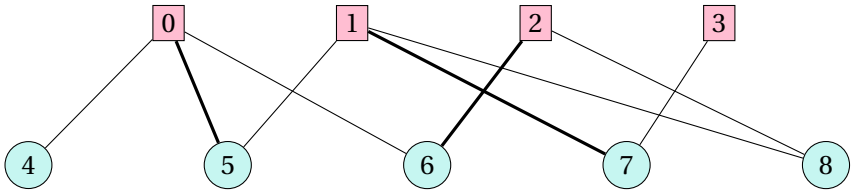
```
1 : Fonction CHEMIN_AUGMENTANT(G = (V = (U, D), E), Γ, F_U, F_D))  ▷ M est le couplage, vide initialement
2 :     si une arête e = (u, v) entre un sommet de F_U et un sommet de F_D existe alors
3 :         CHEMIN_AUGMENTANT(G, Γ ∪ {e}, F_U \ {u}, F_D \ {v})
4 :     sinon
5 :         Créer le graphe orienté G_o                ▷ ∀e ∈ E, e de U vers D si e ∉ Γ, l'inverse sinon
6 :         Calculer le plus court chemin π entre un sommet de F_U et un de F_D dans G_o
7 :         si un tel chemin π n'existe pas alors
8 :             renvoyer M
9 :         sinon
10 :            CHEMIN_AUGMENTANT(G, Γ ⊕ π, F_U \ {π_start}, F_D \ {π_end})
11 :                                ▷ π_start début du chemin π, π_end fin du chemin π et Γ ⊕ π = {e ∈ E, e ∈ Γ \ π ∪ e ∈ π \ Γ}
```

Le graphe de départ de l'algorithme est le suivant :



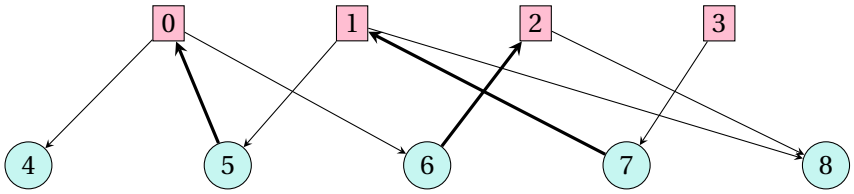
$\Gamma = \emptyset$

On effectue **trois appels récursifs** et, à chaque fois, on a trouvé une arête dont les sommets sont tous les deux exposés.



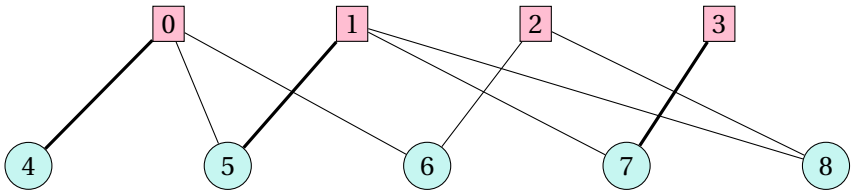
$\Gamma = \{(0,5),(1,7),(2,6)\}$

À ce stade de l'algorithme, aucun sommet exposé n'est relié par une arête à un autre sommet exposé. Donc, on construit le graphe G_o d'après le couplage Γ. On trouve le chemin le plus court entre les deux premiers sommets exposés 3 et 4 : π.



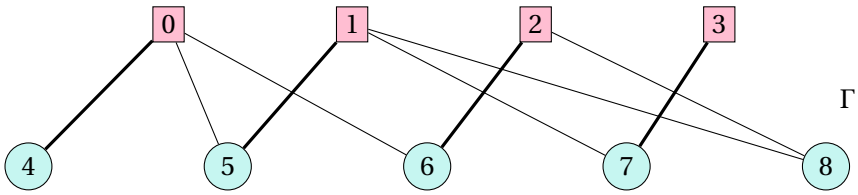
$\Gamma = \{(0,5),(1,7),(2,6)\}$
$\pi = \{3,7,1,5,0,4\}$

On en déduit un nouveau couplage $\Gamma = \Gamma \oplus \pi$:



$\Gamma = \{(0,4),(1,5),(3,7)\}$

On effectue **un appel récursif** et on trouve une arête dont les sommets sont tous les deux exposés : $(2,6)$. On effectue un dernier appel récursif et l'algorithme se termine car un seul sommet est non couplé.



$\Gamma = \{(0,4),(1,5),(3,7),(2,6)\}$

FIGURE 9 – Étapes de l'algorithme de recherche d'un couplage de cardinal maximum