INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Année scolaire
2016-2017

AAIA

Première partie : Algorithmique avancée pour les graphes

C. Solnon

Année scolaire
2016-2017

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

C. Solnon

TABLE DES MATIÈRES

CHAPITRE 1
MOTIVATIONS

Pour résoudre de nombreux problèmes, nous sommes amenés à dessiner des graphes, c'est-à-dire des points (appelés sommets) reliés deux à deux par des lignes (appelées arcs ou arêtes). Ces graphes font abstraction des détails non pertinents pour la résolution du problème et permettent de se focaliser sur les aspects importants.

▶ Quelques exemples de modélisation par des graphes

Réseaux de transport

Un réseau de transport (routier, ferroviaire, métro, etc) peut être représenté par un graphe dont les sommets sont des lieux (intersections de rues, gares, stations de métro, etc) et les arcs indiquent la possibilité d'aller d'un lieu à un autre (par un tronçon de route, une ligne de train ou de métro, etc). Ces arcs peuvent être valués, par exemple, par leur longueur, la durée estimée pour les traverser, ou encore un coût. Etant donné un tel graphe, nous pourrons nous intéresser, par exemple, à la résolution des problèmes suivants :

- Quel est le plus court chemin (en longueur, en durée, ou encore en coût) pour aller d'un sommet à un autre ?
- Est-il possible de passer par tous les sommets sans passer deux fois par un même arc ?
- Peut-on aller de n'importe quel sommet vers n'importe quel autre sommet ?

Réseaux sociaux

Les réseaux sociaux (LinkedIn, Facebook, etc) peuvent être représentés par des graphes dont les sommets sont des personnes et les arêtes des relations entre ces personnes. Etant donné un tel graphe, nous pourrons nous intéresser, par exemple, à la résolution des problèmes suivants :

- Combien une personne a-t-elle de relations ?
- Quelles sont les communautés (sous-ensemble de personnes en relation directe les unes avec les autres) ?
- Par combien d'intermédiaires faut-il passer pour relier une personne à une autre ?

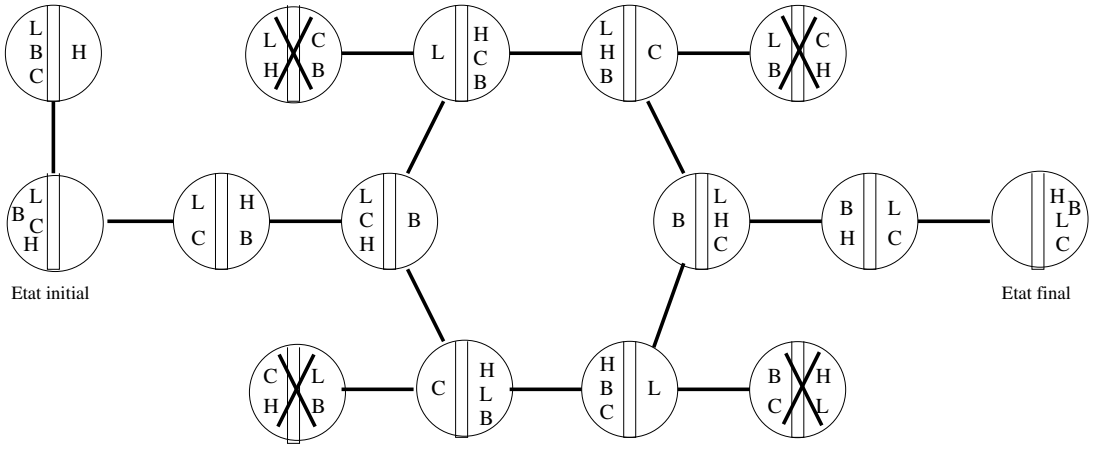
Planification

Certains problèmes peuvent être spécifiés par un état initial, un état final, un certain nombre d'états intermédiaires et des règles de transition précisant comment on peut passer d'un état à l'autre. Résoudre le problème consiste alors à trouver une suite de transitions permettant de passer de l'état initial à l'état final. Beaucoup de jeux et autres "casse-tête" peuvent être modélisés ainsi. Considérons, par exemple, le problème du chou, de la brebis et du loup :

> *Un homme se trouve au bord d'une rivière qu'il souhaite traverser, en compagnie d'un loup, d'une brebis et d'un chou. Malheureusement, il ne dispose que d'une petite barque, ne pouvant porter en plus de*

lui-même qu'un seul de ses compagnons (le loup ou la brebis ou le chou). Bien sûr, la brebis refuse de rester seule avec le loup, tandis que le chou refuse de rester seul avec la brebis. Comment peut-il s'y prendre pour traverser la rivière avec ses trois compagnons et continuer son chemin ?

L'état initial est l'état où tout le monde est sur une rive de la rivière, tandis que l'état final est l'état où tout le monde est sur l'autre rive de la rivière. La règle de transition est la suivante : si l'homme est sur une rive avec certains de ses compagnons, alors il peut passer sur l'autre rive, soit seul, soit accompagné par un seul de ses compagnons se trouvant sur la même rive que lui, sous réserve qu'il ne laisse pas le loup seul avec la brebis, ou la brebis seule avec le chou. Ce problème peut être modélisé par un graphe dont les sommets représentent les états possibles, et les arêtes le fait qu'on peut passer d'un état à l'autre par une transition :



où le loup est représenté par la lettre L, le chou par C, la brebis par B et l'homme par H, et où un état est représenté par un cercle coupé en deux demi-cercles représentant les rives gauche et droite de la rivière.
Résoudre le problème revient alors à chercher un chemin allant de l'état initial à l'état final.

▶ Objectifs pédagogiques et organisation de cette partie

Dans le cours d'introduction à l'algorithmique du premier semestre, vous avez étudié des algorithmes fondamentaux pour organiser des données. Ces algorithmes ont été décrits avec un niveau de détail très proche de programmes écrits dans des langages procéduraux tels que le C.
Dans cette partie, nous allons tout d'abord introduire les définitions et concepts nécessaires pour modéliser des problèmes à l'aide de graphes (chapitre 2), puis nous présenterons les structures de données généralement utilisées pour manipuler un graphe (chapitre 3). Nous étudierons ensuite un certain nombre d'algorithmes classiques sur les graphes : algorithmes pour parcourir des graphes (chapitre 4), pour rechercher des plus courts chemins dans des graphes (chapitre 5), et pour rechercher des arbres couvrants minimaux (chapitre 6). Ces chapitres seront l'occasion d'approfondir des aspects méthodologiques concernant la validation d'algorithmes : nous prouverons que les algorithmes étudiés sont corrects, et nous étudierons leur complexité en temps. Contrairement à ce qui a été fait dans la première partie, les algorithmes pourront être introduits avec un niveau de détail moins fin, en faisant abstraction des structures de données utilisées. Ces structures de données sont bien évidemment très importantes pour une implémentation efficace en pratique. Ces aspects seront généralement discutés au moment où nous étudierons la complexité des algorithmes : bien souvent, un même algorithme peut être implémenté avec différentes structures de données, donnant lieu à différentes performances en pratique.
Enfin, cette partie se terminera par un chapitre d'ouverture à la complexité théorique des problèmes. Vous y apprendrez que certains problèmes sont plus difficiles que d'autres, et notamment que certains problèmes ne peuvent être résolus en temps polynomial, à moins que $\mathcal{P} = \mathcal{NP}$.

CHAPITRE 2
DÉFINITIONS

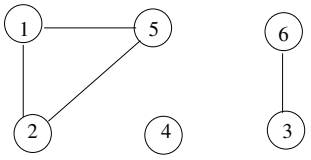
Un graphe définit une relation binaire sur un ensemble d'éléments. Plus précisément, un **graphe** est défini par un couple $G = (S, A)$ tel que S est un ensemble fini de sommets (aussi appelés nœuds), et $A \subseteq S \times S$ est un ensemble de couples de sommets définissant une relation binaire sur S. L'**ordre** d'un graphe est le nombre de ses sommets.
Un graphe peut être orienté ou non, selon que la relation binaire est asymétrique ou symétrique.

2.1 Graphes non orientés

Dans un graphe non orienté, la relation binaire définie par A est symétrique.
Plus précisément, un graphe $G = (S, A)$ est **non orienté** si pour tout couple de sommets $(s_i, s_j) \in S \times S$:

$$(s_i, s_j) \in A \Leftrightarrow (s_j, s_i) \in A$$

Une paire $\{s_i, s_j\} \in A$ est appelée une **arête**, et est représentée graphiquement par s_i—-s_j.
Par exemple,



représente le graphe non orienté $G = (S, A)$ avec

$$\begin{aligned} S &= \{1, 2, 3, 4, 5, 6\} \\ A &= \{\{1, 2\}, \{1, 5\}, \{5, 2\}, \{3, 6\}\} \end{aligned}$$

Un graphe non-orienté est **simple** s'il ne comporte pas de boucle (arête reliant un sommet à lui-même), et s'il ne comporte jamais plus d'une arête entre deux sommets. Un graphe non orienté qui n'est pas simple est un **multi-graphe**. Dans le cas d'un multi-graphe, A n'est plus un ensemble mais un multi-ensemble d'arêtes. Nous ne considèrerons dans ce cours que des graphes non orientés simples.
Un graphe non-orienté est **complet** s'il comporte une arête $\{s_i, s_j\}$ pour toute paire de sommets différents $(s_i, s_j) \in S^2$.

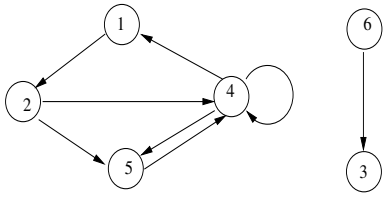
Un sommet s_i est **adjacent** à un autre sommet s_j s'il existe une arête entre s_i et s_j. L'ensemble des sommets adjacents à un sommet s_i est défini par :

$$adj(s_i) = \{s_j | \{s_i, s_j\} \in A\}$$

Le **degré** d'un sommet s_i, noté $d^\circ(s_i)$, est le nombre d'arêtes incidentes à ce sommet.
Autrement dit, $d^\circ(s_i) = \mid adj(s_i) \mid$ (où $|E|$ dénote le cardinal de l'ensemble E).

2.2 Graphes orientés

Dans un **graphe orienté**, la relation binaire définie par A n'est pas symétrique et les couples $(s_i, s_j) \in A$ sont orientés, c'est-à-dire que (s_i, s_j) est un couple ordonné, où s_i est le sommet initial, et s_j le sommet terminal. Un couple (s_i, s_j) est appelé un **arc**, et est représenté graphiquement par $s_i \to s_j$. Par exemple,



représente le graphe orienté $G = (S, A)$ avec

$$\begin{aligned} S &= \{1, 2, 3, 4, 5, 6\} \\ A &= \{(1, 2), (2, 4), (2, 5), (4, 1), (4, 4), (4, 5), (5, 4), (6, 3)\} \end{aligned}$$

Un graphe orienté est un **p-graphe** s'il comporte au plus p arcs entre deux sommets ; il est dit **élémentaire** s'il ne contient pas de boucle. Nous ne considèrerons dans ce cours que des graphes orientés qui sont des 1-graphes.
Un graphe orienté est **complet** s'il comporte un arc (s_i, s_j) et un arc (s_j, s_i) pour tout couple de sommets différents $(s_i, s_j) \in S^2$.
Un sommet s_i est **successeur** (resp. **prédécesseur**) d'un autre sommet s_j s'il existe un arc de s_i vers s_j (resp. de s_j vers s_i). Les ensembles de sommets prédécesseurs et successeurs d'un sommet s_i sont définis par :

$$\begin{aligned} succ(s_i) &= \{s_j | (s_i, s_j) \in A\} \\ pred(s_i) &= \{s_j | (s_j, s_i) \in A\} \end{aligned}$$

Le **demi-degré extérieur** d'un sommet s_i, noté $d^{\circ+}(s_i)$, est le nombre d'arcs partant de s_i, et son **demi-degré intérieur**, noté $d^{\circ-}(s_i)$, est le nombre d'arcs arrivant à s_i.
Autrement dit, $d^{\circ+}(s_i) = \mid succ(s_i) \mid$ et $d^{\circ-}(s_i) = \mid pred(s_i) \mid$.

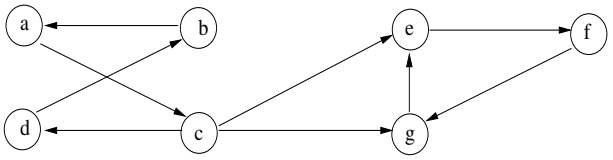
2.3 Graphes partiels et sous-graphes

Un **graphe partiel** est le graphe obtenu en supprimant certains arcs ou arêtes : un graphe $G' = (S, A')$ est un graphe partiel d'un autre graphe $G = (S, A)$ si $A' \subseteq A$.
Un **sous-graphe** est le graphe obtenu en supprimant certains sommets et tous les arcs ou arêtes incidents aux sommets supprimés : un graphe $G' = (S', A')$ est un sous-graphe d'un autre graphe $G = (S, A)$ si $S' \subseteq S$ et $A' = A \cap S' \times S'$. Nous dirons que G' est le sous-graphe de G **induit** par l'ensemble de sommets S'.

2.4 Cheminements et connexités

Dans un graphe orienté $G = (S, A)$, un chemin est une séquence de sommets reliés par des arcs : un **chemin** d'un sommet $u \in S$ vers un sommet $v \in S$ est une séquence de sommets $< s_0, s_1, s_2, ..., s_k >$ telle que $u = s_0$, $v = s_k$, et $(s_{i-1}, s_i) \in A$ pour tout $i \in [1; k]$. La **longueur** du chemin est le nombre d'arcs dans le chemin, c'est-à-dire k. Nous noterons $u \rightsquigarrow v$ le fait qu'il existe un chemin de u vers v, et nous dirons dans ce cas que v est accessible à partir de u. Un chemin est **élémentaire** si les sommets qu'il contient sont tous distincts. Un chemin $< s_0, s_1, ..., s_k >$ forme un **circuit** si $s_0 = s_k$ et si le chemin comporte au moins un arc ($k \geq 1$). Ce circuit est **élémentaire** si en plus les sommets $s_1, s_2, ..., s_k$ sont tous distincts. Une **boucle** est un circuit de longueur 1.
Ces différentes notions se retrouvent dans les graphes non orientés. Dans ce cas, nous parlerons de **chaine** au lieu de chemin, et de **cycle** au lieu de circuit.
Un graphe non orienté est **connexe** si chaque sommet est accessible à partir de n'importe quel autre. Autrement dit, si pour tout couple de sommets distincts $(s_i, s_j) \in S^2$, il existe une chaine entre s_i et s_j. Une **composante connexe** d'un graphe non orienté G est un sous-graphe G' de G qui est connexe et maximal, c'est-à-dire qu'aucun autre sous-graphe connexe de G ne contient G'. Par exemple, le graphe suivant est composé de 2 composantes connexes : la première est le sous-graphe induit par $\{a, b, c, d\}$ et la seconde est le sous-graphe induit par $\{e, f, g\}$.

Ces différentes notions de connexités se retrouvent dans les graphes orientés, en remplaçant naturellement la notion de chaine par celle de chemin : un graphe orienté est **fortement connexe** si chaque sommet est accessible à partir de n'importe quel autre. Autrement dit, si pour tout couple de sommets distincts (s_i, s_j), nous avons $s_i \rightsquigarrow s_j$ et $s_j \rightsquigarrow s_i$. Par exemple, le graphe suivant contient 2 composantes fortement connexes : la première est le sous-graphe induit par $\{a, b, c, d\}$ et la seconde est le sous-graphe induit par $\{e, f, g\}$.



La **fermeture transitive** d'un graphe $G = (S, A)$ est le graphe $G^f = (S, A^f)$ tel que

$$A^f = \{(s_i, s_j) \in S^2 \mid s_i \rightsquigarrow s_j\}$$

Autrement dit, G^f contient un arc (arête) entre deux sommets s'ils sont reliés par un chemin (chaîne). Considérons par exemple le graphe suivant :

Sa fermeture transitive est :

2.5 Arbres et arborescences

Les arbres et les arborescences sont des graphes particuliers très souvent utilisés en informatique pour représenter des données, entre autres.
Etant donné un graphe non orienté comportant n sommets, les propriétés suivantes sont équivalentes pour caractériser un **arbre** :

1. G est connexe et sans cycle,
2. G est sans cycle et possède $n-1$ arêtes,
3. G est connexe et admet $n-1$ arêtes,
4. G est sans cycle, et en ajoutant une arête, on crée un et un seul cycle élémentaire,
5. G est connexe, et en supprimant une arête quelconque, il n'est plus connexe,
6. Il existe une chaine et une seule entre 2 sommets quelconques de G.

Une **forêt** est un graphe dont chaque composante connexe est un arbre.
Une **arborescence** est un graphe orienté sans circuit admettant une racine $s_0 \in S$ telle que, pour tout autre sommet $s_i \in S$, il existe un chemin unique allant de s_0 vers s_i.

CHAPITRE 3

STRUCTURES DE DONNÉES POUR REPRÉSENTER UN GRAPHE

Dans les algorithmes que nous étudierons par la suite, nous utiliserons la notation illustrée dans l'algorithme 1 pour appliquer un même traitement à tous les sommets successeurs d'un sommet s_i donné.

Algorithme 1 : Exemple de notation pour appeler une procédure *traiter* sur tous les successeurs d'un sommet s_i

```
1 pour tout sommet s_j ∈ succ(s_i) faire
2     traiter(s_j)
```

La complexité de cette séquence dépend bien évidemment de la structure de données utilisée pour représenter le graphe (en supposant que la procédure *traiter* a une complexité constante). Il existe deux structures de données classiques pour représenter un graphe : les matrices d'adjacence et les listes d'adjacence. Dans la suite, nous allons supposer que le graphe à représenter comporte n sommets et p arcs, et que les sommets sont numérotés de 0 à $n-1$.

3.1 Représentation par matrice d'adjacence

La matrice d'adjacence d'un graphe $G = (S, A)$ est une matrice M de taille $n \times n$ telle que $M[s_i][s_j] = 1$ si $(s_i, s_j) \in A$, et $M[s_i][s_j] = 0$ sinon. Considérons par exemple les deux graphes de la figure 3.1

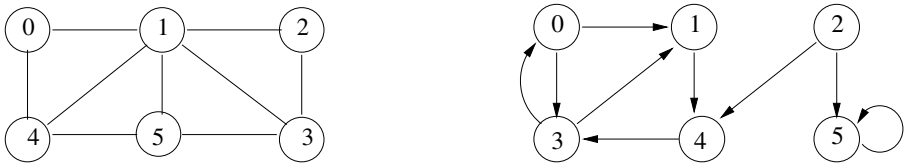


FIGURE 3.1 – Exemple de graphe non orienté (à gauche) et orienté (à droite)

Les matrices d'adjacence de ces deux graphes sont :

	0	1	2	3	4	5
0	0	1	0	0	1	0
1	1	0	1	1	1	1
2	0	1	0	1	0	0
3	0	1	1	0	0	1
4	1	1	0	0	0	1
5	0	1	0	1	1	0

	0	1	2	3	4	5
0	0	1	0	1	0	0
1	0	0	0	0	1	0
2	0	0	0	0	1	1
3	1	1	0	0	0	0
4	0	0	0	1	0	0
5	0	0	0	0	0	1

Si le graphe est valué (par exemple, si des distances sont associées aux arcs), les informations associées aux arcs pourront être mémorisées dans chaque cellule de la matrice M. Dans le cas de graphes non orientés, la matrice M est symétrique par rapport à sa diagonale descendante. Dans ce cas, il est possible de ne mémoriser que la composante triangulaire supérieure de la matrice d'adjacence, mais en divisant ainsi la taille mémoire par deux nous compliquons légèrement les traitements car il sera alors nécessaire de faire un test avant d'accéder à $M[i][j]$.

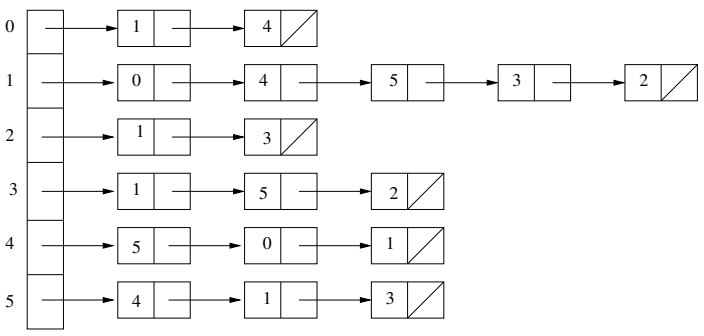
Complexité. La complexité en mémoire d'une représentation par matrice d'adjacence est $\mathcal{O}(n^2)$. La complexité en temps de la fonction déterminant si un arc existe entre deux sommets i et j donnés est $\mathcal{O}(1)$. En revanche, pour accéder à la collection des successeurs d'un sommet s_i, un itérateur devra parcourir en totalité la ligne $M[s_i]$ de la matrice, quel que soit le degré du sommet. Dans ce cas, la complexité de la séquence décrite dans l'algorithme 1 sera $\mathcal{O}(n)$.

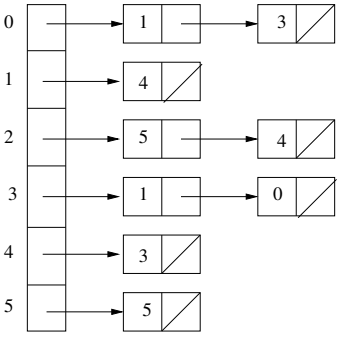
▶ Puissances de la matrice d'adjacence.

La $k^{\text{ième}}$ puissance de la matrice d'adjacence d'un graphe, notée M^k, est obtenue en multipliant M par elle-même k fois (pour tout $k > 0$). Autrement dit, $M^1 = M$ et $M^k = M * M^{k-1}, \forall k > 1$.
La $k^{\text{ième}}$ puissance de M permet de dénombrer les chemins de longueur k. Plus précisément, étant donnés deux sommets s_i et s_j, $M^k[s_i][s_j]$ donne le nombre de chemins de longueur k allant de s_i à s_j. Cette propriété peut être facilement démontrée récursivement en vérifiant que $M^1[s_i][s_j]$ donne le nombre de chemins de longueur 1 allant de s_i à s_j, *i.e.*, le nombre d'arcs (s_i, s_j), puis en montrant que M^k donne le nombre de chemins de longueur k dès lors que M^{k-1} donne le nombre de chemins de longueur $k-1$. Notons que pour calculer M^k, il faudra faire k multiplications. Si le graphe comporte n sommets, chaque multiplication nécessite $\mathcal{O}(n^3)$ opérations, de sorte que le calcul de M^k se fait en $\mathcal{O}(kn^3)$. Il est possible d'améliorer cette complexité en exploitant le fait que si k est pair $M^k = (M^{k/2})^2$ et si k est impair $M^k = M \cdot (M^{k/2})^2$, de sorte qu'il faudra $\mathcal{O}(\log_2(k))$ multiplications de matrices au lieu de k.

3.2 Représentation par listes d'adjacence

La représentation par listes d'adjacence d'un graphe $G = (S, A)$ consiste en un tableau $succ$ de n listes, une pour chaque sommet de S. Pour chaque sommet $s_i \in S$, $succ[s_i]$ contient la liste de tous les sommets successeurs de s_i. Les sommets de chaque liste d'adjacence sont généralement chainés selon un ordre arbitraire. Par exemple, les listes d'adjacence des deux graphes de la figure 3.1 sont :





Si le graphe est valué (par exemple, si des distances sont associées aux arcs), il est possible de stocker dans les listes d'adjacence, en plus du numéro de sommet successeur, la valuation de l'arc. Dans le cas de graphes non orientés, pour chaque arête $\{s_i, s_j\}$, s_j appartiendra à la liste chainée de $succ[s_i]$, et s_i appartiendra à la liste chainée de $succ[s_j]$.

Complexité : La complexité en mémoire d'une représentation par listes d'adjacence est $\mathcal{O}(n+p)$. La complexité en temps de la fonction déterminant si un arc existe entre deux sommets s_i et s_j donnés est $\mathcal{O}(d^\circ(s_i))$. Pour accéder à la collection des successeurs d'un sommet s_i, il faudra parcourir la liste d'adjacence $succ[s_i]$ de sorte que la complexité de la séquence décrite dans l'algorithme 1 sera $\mathcal{O}(d^\circ(s_i))$. En revanche, l'accès à la collection des prédécesseurs d'un sommet est mal aisé avec cette représentation, et nécessite le parcours de toutes les listes d'adjacence de $succ$. Une solution dans le cas où il est nécessaire de connaitre les prédécesseurs d'un sommet est de maintenir, en plus de la liste des successeurs, la liste des prédécesseurs.

3.3 Itérateurs

Lors de l'implémentation d'un algorithme à l'aide d'un langage orienté objet, les parcours de collections sont généralement faits en utilisant des itérateurs, ce qui permet de rendre les procédures utilisant le parcours indépendantes de la structure de donnée utilisée pour implémenter la collection. Dans ce cas, la classe Graphe possède généralement des méthodes permettant de créer des itérateurs pour parcourir la collection des prédécesseurs et successeurs d'un sommet donné.
Par ailleurs, les sommets d'un graphe ne sont généralement pas numérotés de 0 à $n-1$, comme nous l'avons supposé précédemment. Bien souvent, les sommets sont des objets, et les attributs de ces objets donnent des informations sur les sommets. Dans ce cas, il sera nécessaire d'utiliser une structure de données permettant d'associer un numéro entre 0 et $n-1$ à chaque sommet (une table de hachage, par exemple). Par ailleurs, la classe Graphe devra posséder une méthode permettant de créer un itérateur pour parcourir la collection des sommets d'une instance de la classe.

CHAPITRE 4
PARCOURS DE GRAPHES

Note préliminaire au chapitre : De façon implicite, nous considérons dans ce chapitre des graphes orientés. Cependant tous les algorithmes de ce chapitre peuvent être étendus de façon immédiate aux graphes non orientés.

Pour déterminer l'ensemble des sommets accessibles à partir d'un sommet donné s_0, il est nécessaire de parcourir le graphe de façon systématique, en partant de s_0 et en utilisant les arcs pour découvrir de nouveaux sommets. Dans ce chapitre, nous étudions les deux principales stratégies d'exploration :

- le parcours en largeur, qui consiste à explorer les sommets du graphe niveau par niveau, à partir de s_0 ;
- le parcours en profondeur, qui consiste à partir de s_0 pour suivre un chemin le plus loin possible, jusqu'à un cul-de-sac ou l'arrivée sur un sommet déjà visité, puis à faire des retours en arrière pour reprendre tous les chemins ignorés précédemment.

Dans les deux cas, l'algorithme procède par coloriage des sommets.

- Initialement, tous les sommets sont blancs. Nous dirons qu'un sommet blanc n'a pas encore été découvert.
- Lorsqu'un sommet est "découvert" (autrement dit, quand l'algorithme arrive pour la première fois sur ce sommet), il est colorié en gris.
- Un sommet est colorié en noir lorsque tous ses successeurs sont gris ou noirs (autrement dit, lorsqu'ils ont tous été découverts).

De façon pratique, l'algorithme utilise une liste "d'attente au coloriage en noir" qui contient l'ensemble des sommets gris. Un sommet est mis dans la liste d'attente dès qu'il est colorié en gris. À chaque itération, un sommet gris de la liste d'attente fait rentrer dans la liste un (ou plusieurs) de ses successeurs qui sont encore blancs (en les coloriant en gris). Quand tous les successeurs d'un sommet gris de la liste d'attente sont soit gris soit noirs, il peut être colorié en noir et sortir de la liste d'attente.

La différence fondamentale entre le parcours en largeur et le parcours en profondeur provient de la façon de gérer cette liste d'attente au coloriage en noir : le parcours en largeur utilise une file d'attente (FIFO, étudiée au premier semestre), où le premier sommet arrivé dans la file est aussi le premier à en sortir, tandis que le parcours en profondeur utilise une pile (LIFO, étudiée au premier semestre), où le dernier sommet arrivé dans la pile est le premier à en sortir.

Notons que la notion de parcours d'un graphe a déjà été partiellement abordée au premier semestre : vous avez vu à ce moment comment parcourir un arbre binaire (qui est un graphe particulier) en profondeur et en largeur.

▶ Arborescence liée à un parcours de graphe

Au fur et à mesure du parcours, l'algorithme construit une arborescence de découverte des sommets accessibles depuis le sommet de départ s_0, appelée arborescence d'un parcours à partir de s_0. Cette arborescence contient un arc (s_i, s_j) si et seulement si le sommet s_j a été découvert à partir du sommet s_i (autrement dit, si c'est le sommet

s_i qui a colorié s_j en gris). Ce graphe est effectivement une arborescence, dans la mesure où chaque sommet a au plus un prédécesseur, à partir duquel il a été découvert. La racine de cette arborescence est le sommet de départ du parcours, s_0.
L'arborescence associée à un parcours de graphe est mémorisée dans un tableau π tel que $\pi[s_j] = s_i$ si s_j a été découvert à partir de s_i, et $\pi[s_k] = null$ si s_k est la racine, ou s'il n'existe pas de chemin de la racine vers s_k.

4.1 Parcours en largeur (Breadth First Search = BFS)

Le parcours en largeur est effectué en gérant la liste d'attente au coloriage comme une file d'attente (FIFO = First In First Out). Autrement dit, l'algorithme enlève à chaque fois le plus vieux sommet gris dans la file d'attente (ce sommet sera nommé $s_{FirstOut}$), et il introduit tous les successeurs blancs de ce sommet dans la file d'attente, en les coloriant en gris. L'algorithme 2 décrit ce principe.

Algorithme 2 : Parcours en largeur d'un graphe

```
1  Fonction BFS(g, s_0)
       Entrée       : Un graphe g et un sommet s_0 de g
       Postcondition : Retourne une arborescence π d'un parcours en largeur de g à partir de s_0
       Déclaration  : Une file (FIFO) f initialisée à vide
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5      Ajouter s_0 dans la file f et colorier s_0 en gris
6      tant que la file f n'est pas vide faire
7          Soit s_FirstOut le sommet le plus ancien dans f
8          pour tout sommet s_i ∈ succ(s_FirstOut) faire
9              si s_i est blanc alors
10                 Ajouter s_i dans la file f et colorier s_i en gris
11                 π[s_i] ← s_FirstOut
12         Enlever s_FirstOut de f et colorier s_FirstOut en noir
13     retourne π
```

Complexité. Soient n et p le nombre de sommets et arcs de g, respectivement. Chaque sommet accessible depuis s_0 est mis au plus une fois dans la file f. En effet, seuls les sommets blancs entrent dans la file, et un sommet blanc est colorié en gris quand il entre dans la file, puis en noir quand il en sort, et ne pourra jamais redevenir blanc. À chaque passage dans la boucle lignes 6 à 12, il y a exactement un sommet qui est enlevé de la file. Cette boucle sera donc exécutée au plus n fois. À chaque fois qu'un sommet est enlevé de la file, la boucle lignes 8 à 11 parcourt tous les successeurs du sommet enlevé, de sorte que les lignes 9 à 11 seront exécutées au plus une fois pour chaque arc. Par conséquent, la complexité de l'algorithme 2 est $\mathcal{O}(n+p)$ (sous réserve d'une implémentation par listes d'adjacence).

▶ Utilisation de BFS pour rechercher des plus courts chemins

Considérons deux sommets s_0 et s_i tels qu'il existe au moins un chemin de s_0 jusque s_i. Le **plus court chemin** de s_0 jusque s_i est le chemin comportant le moins d'arcs. La **distance** de s_0 jusque s_i, notée $\delta(s_0, s_i)$, est le nombre d'arcs de ce plus court chemin.
Dans le cas de graphes non orientés, on pourra vérifier à titre d'exercice que cette distance vérifie bien les trois propriétés qui en font une métrique, à savoir, séparation, symétrie et inégalité triangulaire. En revanche, dans le cas de graphes orientés la symétrie n'est pas toujours vérifiée (autrement dit, on peut avoir $\delta(s_0, s_i) \neq \delta(s_i, s_0)$) de sorte

que le terme distance est un abus de langage dans ce cas. Notons que nous verrons au chapitre 5 une définition plus générale de cette notion de plus court chemin dans le cas de graphes pondérés.

Algorithme pour calculer $\delta(s_0, s_i)$. L'algorithme 2 peut être facilement adapté pour calculer $\delta(s_0, s_i)$ pour chaque sommet s_i accessible depuis s_0. Nous introduisons pour cela un tableau d. Nous initialisons $d[s_0]$ à 0 au début de l'algorithme, et nous ajoutons l'instruction $d[s_i] \leftarrow d[s_{FirstOut}] + 1$ au moment où $s_{FirstOut}$ fait entrer un sommet s_i dans la file f, c'est-à-dire ligne 10 de l'algorithme 2.

Preuve de correction. Montrons qu'à la fin de l'exécution de l'algorithme 2, $d[s_i] = \delta(s_0, s_i)$ pour tout sommet noir s_i. Pour cela, montrons que les trois propriétés suivantes sont des invariants qui sont satisfaits à chaque passage à la ligne 6 de l'algorithme 2.

1. Aucun successeur d'un sommet noir n'est blanc.
2. Pour tout sommet s_i qui est gris ou noir, $\delta(s_0, s_i) = d[s_i]$.
3. Notons $s_{FirstOut}$ le prochain sommet à sortir de f. Il y a deux possibilités :
   — soit tous les sommets de f ont la même valeur de d que $s_{FirstOut}$;
   — soit f contient 1 ou plusieurs sommets dont la valeur de d est égale à $d[s_{FirstOut}]$, suivis par 1 ou plusieurs sommets dont la valeur de d est égale à $d[s_{FirstOut}] + 1$.

Montrons tout d'abord que ces trois invariants sont vérifiés au premier passage à la ligne 6. En effet, à ce moment la file f contient un seul sommet (s_0) qui est gris et tous les autres sommets sont blancs. Les invariants (1) et (3) sont donc naturellement vérifiés. Pour l'invariant (2), nous avons $d[s_0] = 0 = \delta(s_0, s_0)$.

Supposons maintenant que l'invariant est vérifié au $k^{\text{ème}}$ passage et montrons qu'il sera vérifié au $k + 1^{\text{ème}}$ passage. À l'exécution des lignes 6 à 12, le sommet $s_{FirstOut}$ passe de gris à noir, sort de f et fait entrer dans f tous ses successeurs s_i qui sont blancs, en les coloriant en gris. Montrons que les trois invariants restent vrais :

1. Le seul sommet à devenir noir est $s_{FirstOut}$ et tous ses successeurs qui étaient blancs sont coloriés en gris à la fin de l'exécution de la boucle lignes 8 à 11. Par conséquent, aucun successeur d'un sommet noir n'est blanc.
2. Pour chaque successeur s_i de $s_{FirstOut}$ qui est encore blanc, l'algorithme affecte $d[s_i]$ à $d[s_{FirstOut}]+1$. Pour montrer que l'invariant (2) reste vrai, il faut montrer que $d[s_i] = \delta(s_0, s_i) = \delta(s_0, s_{FirstOut}) + 1$. Imaginons qu'il existe un chemin de s_0 à s_i dont la longueur soit inférieure à $\delta(s_0, s_{FirstOut}) + 1$. Comme s_0 est noir et s_i est blanc et que l'invariant (1) nous dit qu'aucun successeur d'un sommet noir n'est blanc, ce chemin passe nécessairement par un sommet gris. Or, l'invariant (3) nous dit que $s_{FirstOut}$ a la plus petite valeur de d parmi tous les sommets gris de la file. Par conséquent, aucun chemin allant de s_0 à s_i ne peut avoir une longueur inférieure à $\delta(s_0, s_{FirstOut}) + 1$ et donc $\delta(s_0, s_i) = \delta(s_0, s_{FirstOut}) + 1$.
3. À la fin de chaque itération, $s_{FirstOut}$ est enlevé de f et tous ses successeurs blancs ont été ajoutés à la fin de f avec une valeur de d égale à $d[s_{FirstOut}] + 1$. Comme $s_{FirstOut}$ était le sommet de f ayant la plus petite valeur de d, et que tous les autres sommets de f ont une valeur de d inférieure ou égale à $d[s_{FirstOut}] + 1$, l'invariant (3) reste vérifié.

Affichage du plus court chemin. Pour afficher les sommets du plus court chemin de s_0 jusqu'à un sommet s_k, il suffit de remonter dans l'arborescence π de s_k jusqu'à la racine s_0, comme décrit dans l'algorithme 3.

Algorithme 3 : Affichage du plus court chemin de s_0 jusque s_j à partir d'une arborescence π de s_0

```
1  Procédure plusCourtChemin(s_0, s_j, π)
       Entrée        : 2 sommets s_0 et s_j, et une arborescence π
       Précondition  : π = arborescence couvrante retournée par l'algorithme 2 appelé à partir de s_0
       Postcondition : Affiche un plus court chemin pour aller de s_0 jusque s_j
2      si s_0 = s_j alors afficher(s_0);
3      sinon si π[s_j] = null alors afficher("Il n'y a pas de chemin de ",s_0," jusque ",s_j);
4      sinon
5          plusCourtChemin(s_0, π[s_j], π)
6          afficher(" suivi de ",s_j)
```

4.2 Parcours en profondeur (Depth First Search = DFS)

Le parcours en profondeur est obtenu en gérant la liste d'attente au coloriage en noir comme une pile (LIFO = Last In First Out). Autrement dit, l'algorithme considère à chaque fois le dernier sommet gris entré dans la pile, et introduit devant lui tous ses successeurs blancs. Ce principe est décrit dans l'algorithme 4.

Algorithme 4 : Parcours en profondeur d'un graphe

```
1  Fonction DFS(g, s_0)
       Entrée        : Un graphe g et un sommet s_0 de g
       Postcondition : Retourne une arborescence π d'un parcours en profondeur de g à partir de s_0
       Déclaration   : Une pile (LIFO) p initialisée à vide
2      pour tout sommet s_i ∈ S faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5      Empiler s_0 dans p et colorier s_0 en gris
6      tant que la pile p n'est pas vide faire
7          Soit s_i le dernier sommet entré dans p (au sommet de p)
8          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
9              Empiler s_j dans p et colorier s_j en gris
10             π[s_j] ← s_i
11         sinon
12             Dépiler s_i de p et colorier s_i en noir
13     retourne π
```

Complexité : Chaque sommet accessible depuis s_0 est mis, puis enlevé, exactement une fois dans la pile, comme dans BFS, et à chaque passage dans la boucle lignes 6 à 12, soit un sommet est empilé (si s_i a encore un successeur blanc), soit un sommet est dépilé (si s_i n'a plus de successeur blanc). Par conséquent, l'algorithme passera au plus $2n$ fois dans la boucle lignes 6 à 12. À chaque passage, il faut parcourir la liste des successeurs de s_i pour chercher un successeur blanc. Si nous utilisons un itérateur qui mémorise pour chaque sommet de la pile le dernier successeur de ce sommet qui était blanc, alors la complexité de DFS est $\mathcal{O}(n+p)$ (sous réserve d'une implémentation par listes d'adjacence).

▶ Version récursive de DFS.

Cet algorithme peut s'écrire récursivement sans utiliser de pile explicite, comme décrit dans l'algorithme 5. La complexité de cet algorithme est la même que sa version itérative. Dans ce cas, l'arborescence π ainsi que la structure de donnée mémorisant la couleur des sommets sont des variables globales (ou des paramètres en entrée/sortie de DFSrec) et il faut les initialiser auparavant : les éléments de π à $null$ et la couleur des sommets à blanc. Notons que π correspond aux enchaînements d'appels récursifs : $\pi[s_j] = s_i$ ssi DFSrec(g, s_j) a été appelé par DFSrec(g, s_i).

Algorithme 5 : Parcours en profondeur récursif d'un graphe

```
1  Procédure DFSrec(g, s_0)
       Entrée       : Un graphe g et un sommet s_0 de g
       Précondition : s_0 est blanc
2      début
3          Colorier s_0 en gris
4          pour tout s_j ∈ succ(s_0) faire
5              si s_j est blanc alors
6                  π[s_j] ← s_0
7                  DFSrec(g, s_j)
8          Colorier s_0 en noir
```

▶ Recherche de circuits

Lors du parcours en profondeur d'un graphe avec l'algorithme 5, si un successeur s_j du sommet s_0 est déjà gris, cela implique qu'il existe un chemin permettant d'aller de s_j jusque s_0, et donc qu'il existe un circuit. Ainsi, un algorithme pour détecter si un graphe contient un circuit peut être obtenu en rajoutant dans l'algorithme 5 une instruction entre les lignes 4 et 5 testant si s_j est gris.

▶ Calcul d'une forêt en profondeur

Un parcours en profondeur à partir d'un sommet s_0 donné permet de découvrir l'ensemble des sommets accessibles depuis s_0 et de construire une arborescence ayant pour racine s_0. À l'issue de ce parcours, les sommets qui sont encore blancs ne sont pas accessibles depuis s_0. Nous pouvons alors recommencer un nouveau parcours à partir d'un sommet blanc, afin de découvrir de nouveaux sommets, et répéter cela jusqu'à ce que tous les sommets soient noirs. Nous construisons ainsi une forêt contenant une arborescence différente pour chaque parcours en profondeur. Ce principe est décrit dans l'algorithme 6.

Algorithme 6 : Calcul d'une forêt en profondeur

```
1  Fonction ForetDFS(g)
       Entrée       : Un graphe g
       Postcondition : Retourne une forêt en profondeur π
2      pour tout sommet s_i ∈ S faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5      pour chaque sommet s_i de g faire
6          si s_i est blanc alors DFSrec(g, s_i);
7      retourne π
```

▶ Recherche des composantes connexes d'un graphe non orienté

La forêt en profondeur retournée par l'algorithme 6 peut être utilisée pour rechercher les composantes connexes d'un graphe non orienté : deux sommets appartiennent à une même composante connexe si et seulement s'ils appartiennent à la même arborescence de la forêt. Par exemple, l'algorithme 7 décrit une fonction qui détermine si deux sommets donnés appartiennent à la même composante connexe d'un graphe en comparant les racines des arborescences contenant les sommets. La complexité en temps de la recherche des racines est linéaire par rapport à la profondeur des arborescences contenant les deux sommets. Dans le pire des cas, π contient une seule arborescence telle que chaque sommet a exactement un fils. Dans ce cas, la profondeur de l'arborescence sera égale au nombre de sommets du graphe. Par conséquent, la complexité en temps de *mêmeCC* est la même que celle de *ForetDFS*, *i.e.*, $\mathcal{O}(n+p)$ si le graphe a n sommets et p arcs.

Algorithme 7 : Détermine si deux sommets appartiennent à une même composante connexe d'un graphe

1 **Fonction** *mêmeCC*(s_i, s_j, g)
  Entrée **:** deux sommets s_i et s_j, et un graphe g
  Postcondition : Retourne vrai si s_i et s_j appartiennent à une même composante connexe de g, faux sinon
2 $\pi \leftarrow$ ForetDFS(g)
3 **tant que** $\pi[s_i] \neq null$ **faire** $s_i \leftarrow \pi[s_i]$;
4 **tant que** $\pi[s_j] \neq null$ **faire** $s_j \leftarrow \pi[s_j]$;
5 **retourne** $s_i = s_j$

▶ Tri topologique des sommets d'un DAG

Un **DAG (Directed Acyclic Graph)** est un graphe orienté sans circuit. Le **tri topologique** d'un DAG consiste à définir un ordre total des sommets tel que pour tout arc (s_i, s_j), s_i soit plus petit que s_j. Évidemment, si le graphe comporte des circuits, cet ordre n'existe pas. Le tri topologique d'un DAG peut être vu comme un alignement de ses sommets le long d'une ligne horizontale de telle sorte que tous les arcs du graphe soient orientés de gauche à droite.
Pour effectuer un tri topologique, nous allons numéroter les sommets du graphe en fonction de leur ordre de coloriage en noir. Nous utilisons un compteur *cpt*, initialisé à 0 au début de l'algorithme 6. À chaque fois qu'un sommet s_0 est colorié en noir (ligne 7 de l'algorithme 5), nous mémorisons la valeur du compteur et incrémentons le compteur :

$$\begin{aligned} num[s_0] &\leftarrow cpt \\ cpt &\leftarrow cpt + 1 \end{aligned}$$

Nous appellerons ForetDFSnum l'algorithme appliquant ce principe et retournant le tableau num.

Théorème : Après l'exécution de ForetDFSnum sur un DAG $G = (S, A)$, pour tout arc $(s_i, s_j) \in A$, nous avons $num[s_j] < num[s_i]$.

Preuve : À l'appel de DFSrec(g, s_i),
- si s_j est noir alors $num[s_j] < cpt < num[s_i]$
- si s_j est blanc alors $cpt < num[s_j] < num[s_i]$
- s_j ne peut pas être gris car cela impliquerait l'existence d'un circuit.

Par conséquent, pour obtenir un tri topologique des sommets d'un graphe, il suffit de trier les sommets par ordre de valeur de num décroissante. En pratique, il n'est pas nécessaire de trier les sommets : il suffit de ranger les sommets dans un tableau au fur et à mesure de leur coloriage en noir, ce qui permet de retrouver l'ordre topologique sans avoir

à faire de tri. Par conséquent, la complexité d'un tri topologique est la même que celle d'un parcours en profondeur, soit $\mathcal{O}(n+p)$ si le graphe a n sommets et p arcs.

D'une façon plus générale, les DAG sont utilisés dans de nombreuses applications pour représenter des précédences entre événements : les sommets représentent les évènements, et les arcs les relations de précédence. Dans ce cas, un tri topologique permet de trier les évènements de telle sorte qu'un évènement n'apparait qu'après tous les évènements qui doivent le précéder. Nous y reviendrons au chapitre 5.

▶ Recherche des composantes fortement connexes d'un graphe orienté

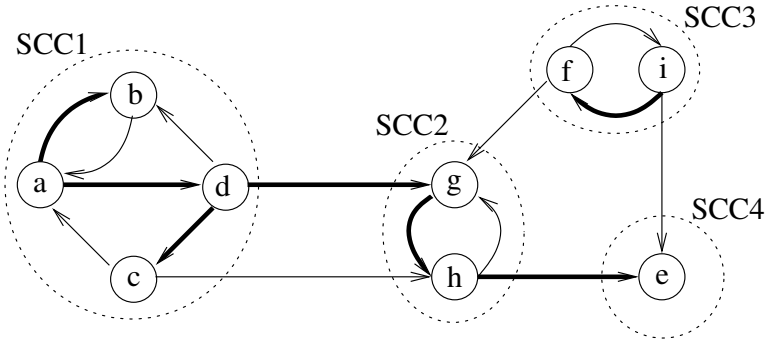
Rappelons qu'un graphe orienté est fortement connexe si pour tout couple de sommets (s_i, s_j), il existe un chemin de s_i vers s_j et un chemin de s_j vers s_i. Cette notion est utilisée pour vérifier l'accessibilité dans des réseaux. Par exemple, si le graphe des rues d'une ville n'est pas fortement connexe, cela implique qu'il n'est pas possible d'aller de n'importe quel point de la ville vers n'importe quel autre point, ce qui peut être fâcheux. Une composante fortement connexe (Strongly Connected Component = SCC) d'un graphe est un sous-graphe fortement connexe maximal. Une façon naïve de déterminer les différentes SCC d'un graphe consiste à faire un parcours (en largeur ou en profondeur) à partir de chacun des sommets du graphe afin de déterminer, pour chaque couple de sommets (s_i, s_j), s'il existe un chemin de s_i vers s_j et un chemin de s_j vers s_i. La complexité de cet algorithme naïf est $\mathcal{O}(n(n+p))$ si le graphe comporte n sommets et p arcs.

Kosaraju a proposé un algorithme plus efficace pour résoudre ce problème. L'idée est de faire un premier parcours en profondeur afin de déterminer un ordre total sur les sommets, à savoir l'ordre inverse de coloriage au noir, puis d'inverser le sens de tous les arcs, et enfin de faire un deuxième parcours (en largeur ou en profondeur) en utilisant l'ordre total pour choisir le prochain sommet de départ. Ce principe est détaillé dans l'algorithme 8.

Algorithme 8 : Recherche des SCC d'un graphe

```
1  Fonction SCC(g)
       Entrée       : Un graphe orienté g = (S, A)
       Postcondition : Retourne l'ensemble des composantes fortement connexes de g
2      SCC ← ∅
3      num ← ForetDFSnum(g)
4      Construire le graphe g^t = (S, A^t) tel que A^t = {(s_i, s_j) | (s_j, s_i) ∈ A}
5      Colorier tous les sommets de g^t en blanc
6      pour chaque sommet s_i pris par ordre de num décroissant faire
7          si s_i est blanc alors
8              B ← {s_j ∈ S | s_j est blanc}
9              DFSrec(g^t, s_i)
10             Ajouter à SCC l'ensemble {s_j ∈ B | s_j est noir}
11     retourne SCC
```

Exemple. Considérons le graphe suivant, comportant 9 sommets nommés de a à i.



Ce graphe comporte 4 SCC qui sont représentées par des ellipses en pointillés. Une forêt calculée par ForetDFSNum(g) est représentée par les arcs en gras. Le tableau π correspondant est :

$-$	a	d	a	h	i	d	g	$-$
a	b	c	d	e	f	g	h	i

et le tableau num donnant l'ordre de passage en noir est :

7	1	5	6	2	9	4	3	8
a	b	c	d	e	f	g	h	i

Ainsi, la boucle lignes 6 à 10 de l'algorithme 8 considèrera les sommets dans l'ordre suivant : $f, i, a, d, c, g, h, e, b$. L'appel à DFSrec$(g^t, f)$ permettra de colorier en noir les sommets f et i de SCC3. L'appel à DFSrec(g^t, a) permettra de colorier en noir les sommets a, b, c et d de SCC1. L'appel à DFSrec(g^t, g) permettra de colorier en noir les sommets g et h de SCC2. L'appel à DFSrec(g^t, e) permettra de colorier en noir le sommet e de SCC4.

Correction de SCC. Pour nous convaincre de la correction de SCC, nous introduisons le graphe des composantes fortement connexes $g^{scc} = (S^{scc}, A^{scc})$ tel que

- S^{scc} est une partition de S telle que chaque sommet de S^{scc} contient tous les sommets de S appartenant à une SCC différente ;
- les arcs de g^{scc} traduisent l'existence de chemins entre les sommets des SCC :

$$A^{scc} = \{(scc_i, scc_j) \in S^{scc} \times S^{scc} \mid \exists s_i \in scc_i, \exists s_j \in scc_j, s_i \rightsquigarrow s_j\}$$

Ce graphe des SCC est un DAG. En effet, s'il existait un circuit alors, tous les sommets des SCC du circuit devraient appartenir à une même SCC. Sur notre exemple, le DAG des SCC comporte 4 sommets et 4 arcs :

$$\begin{aligned} S^{scc} &= \{scc_1 = \{a, b, c, d\}, scc_2 = \{g, h\}, scc_3 = \{f, i\}, scc_4 = \{e\}\} \\ A^{scc} &= \{(scc_1, scc_2), (scc_2, scc_4), (scc_3, scc_2), (scc_3, scc_4)\} \end{aligned}$$

Montrons maintenant qu'après l'exécution de la ligne 3 de l'algorithme 8, pour tout arc $(scc_i, scc_j) \in A^{scc}$, la plus grande valeur de num de l'ensemble des sommets de scc_i est supérieure à la plus grande valeur de num de l'ensemble des sommets de scc_j. Autrement dit,

$$max\{num[s_i] \mid s_i \in scc_i\} > max\{num[s_j] \mid s_j \in scc_j\}$$

Soit s_0 le premier sommet de $scc_i \cup scc_j$ qui est colorié en gris.

- Cas 1 : $s_0 \in scc_i$. Quand s_0 est découvert et colorié en gris, tous les autres sommets de $scc_i \cup scc_j$ sont blancs et sont accessibles à partir de s_0. Dans ce cas, le dernier sommet de $scc_i \cup scc_j$ à être colorié en noir sera s_0 de sorte que $num[s_0] = max\{num[s_i] \mid s_i \in scc_i\} > max\{num[s_j] \mid s_j \in scc_j\}$.
  C'est le cas par exemple pour l'arc (scc_1, scc_2) : le premier sommet à être découvert est a, qui a bien la plus grande valeur de num parmi tous les sommets de $scc_1 \cup scc_2$.

— Cas 2 : $s_0 \in scc_j$. Comme il n'existe pas d'arc allant d'un sommet de scc_j vers un sommet de scc_i, tous les sommets de scc_j seront coloriés en noir avant que le premier sommet de scc_i ne soit découvert et, par conséquent, toutes les valeurs de num des sommets de scc_j seront inférieures à toutes les valeurs de num des sommets de scc_i.
C'est le cas par exemple pour l'arc (scc_3, scc_4) : le premier sommet à être découvert est e, qui a une valeur de num égale à 2, et tous les sommets de scc_3 ont une valeur de num supérieure à 2.

Notons finalement que le graphe transposé g^t a les mêmes SCC que g. Par conséquent, nous pouvons rechercher les SCC dans g^t. En triant les sommets par ordre de num décroissant, nous sommes sûrs que pour tout arc $(scc_i, scc_j) \in A^{scc}$, un sommet de scc_i sera rencontré avant un sommet de scc_j. Le parcours en profondeur à partir de ce premier sommet s_i de scc_i (ligne 9 de l'algorithme 8) permettra de découvrir tous les sommets de scc_i et uniquement les sommets de scc_i. En effet, à l'appel de DFSrec(g^t, s_i) :

— tous les sommets appartenant à une SCC scc_j telle qu'il existe dans g^t un chemin de s_i vers un sommet de scc_j seront noirs (car au moins un sommet de scc_j aura été rencontré avant s_i dans la boucle lignes 6 à 9) ;
— tous les sommets appartenant à scc_i seront blancs car tous les appels précédents à DFSrec seront partis de sommets pour lesquels il n'existe pas de chemin jusque s_i.

.

Complexité de SCC. La construction du graphe transposé g^t est en $\mathcal{O}(n+p)$, et chaque parcours de graphe est également en $\mathcal{O}(n+p)$. Pour éviter d'avoir à trier les sommets par ordre de num décroissant, nous pouvons ranger les sommets dans une liste au fur et à mesure de leur coloriage en noir. Par conséquent, la complexité de SCC est $\mathcal{O}(n+p)$.

CHAPITRE 5
PLUS COURTS CHEMINS

Nous avons vu au chapitre précédent un algorithme permettant de trouver le plus court chemin en nombre d'arcs entre deux sommets. Dans de nombreuses applications, les arcs sont valués (par exemple, des distances, des durées, des coûts ou encore des probabilités sont associés aux arcs) et l'objectif n'est pas de trouver le chemin empruntant le moins d'arcs, mais le chemin optimisant une fonction dépendant des valeurs des arcs empruntés (par exemple, minimisant la somme des durées ou maximisant le produit des probabilités).
Nous étudions dans ce chapitre des algorithmes permettant de résoudre ces problèmes. De façon implicite, nous considérons dans ce chapitre des graphes orientés. Cependant tous les algorithmes peuvent être étendus de façon immédiate aux graphes non orientés.

5.1 Définitions

Soit $G = (S, A)$ un graphe orienté valué tel que la fonction $cout : A \to \mathbb{R}$ associe à chaque arc (s_i, s_j) de A un coût réel $cout(s_i, s_j)$. Le coût d'un chemin $p = \langle s_0, s_1, s_2, \ldots, s_k \rangle$ est défini par la somme des coûts de ses arcs, c'est-à-dire,

$$cout(p) = \sum_{i=1}^{k} cout(s_{i-1}, s_i)$$

Le coût d'un plus court chemin entre deux sommets s_i et s_j est noté $\delta(s_i, s_j)$ et est défini par

$$\begin{array}{lcll} \delta(s_i, s_j) & = & +\infty & \text{si } \nexists \text{ de chemin entre } s_i \text{ et } s_j \\ \delta(s_i, s_j) & = & \min\{cout(p)/p = \text{chemin de } s_i \text{ a } s_j\} & \text{sinon} \end{array}$$

Considérons par exemple le graphe de la figure 5.1(a). Dans ce graphe, $\delta(a, b) = 3$, $\delta(a, e) = 5$, $\delta(a, c) = 9$ et $\delta(a, d) = 11$.

Conditions d'existence d'un plus court chemin : s'il existe un chemin entre deux sommets u et v contenant un circuit de coût négatif, alors $\delta(u, v) = -\infty$, et il n'existe pas de plus court chemin entre u et v. Un circuit négatif est appelé un **circuit absorbant**.
Considérons par exemple le graphe de la figure 5.1(b). Dans ce graphe, $\delta(s, a) = 3$, $\delta(s, c) = 5$, $\delta(s, b) = -1$ et $\delta(s, d) = 11$. Le chemin $\langle s, e, f, e, f, g \rangle$ contient le circuit $\langle e, f, e \rangle$ de coût négatif -3. Autrement dit, à chaque fois que nous passons par ce circuit, le coût total du chemin est diminué de 3. Par conséquent, $\delta(s, g) = -\infty$ et il n'existe pas de plus court chemin entre s et g.

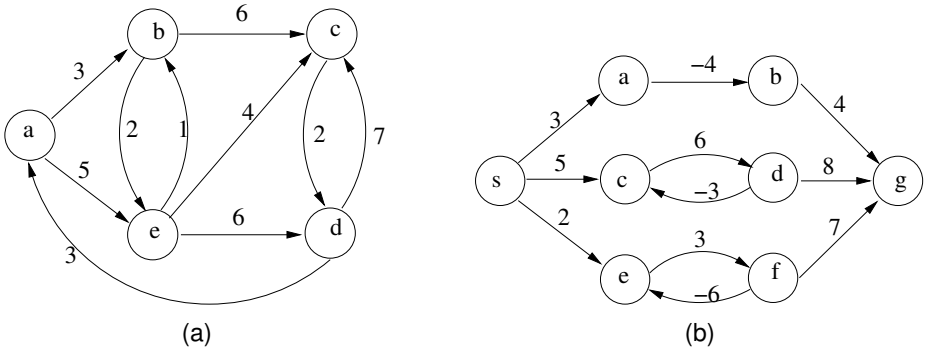


FIGURE 5.1 – Exemples de graphes

Définition du problème des plus courts chemins à origine unique : étant donné un graphe orienté $G = (S, A)$ ne comportant pas de circuit absorbant, une fonction $cout : A \to R$ et un sommet origine $s_0 \in S$, il s'agit de calculer pour chaque sommet $s_j \in S$ le coût $\delta(s_0, s_j)$ du plus court chemin de s_0 à s_j.

Quelques variantes du problème.

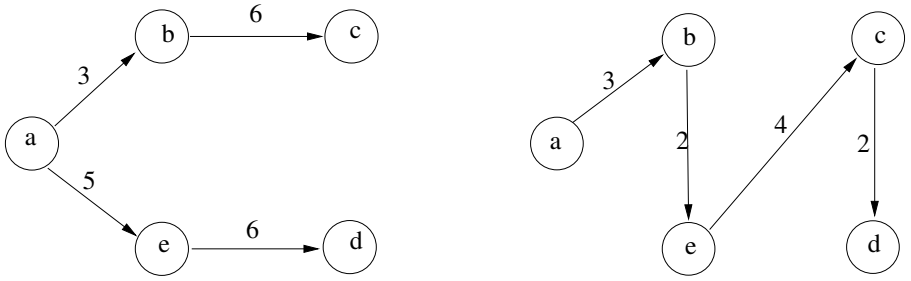
- Pour calculer le plus court chemin allant d'un sommet s_0 vers un autre sommet s_i (la destination est unique), il suffit d'utiliser la résolution du problème précédent, qui calcule tous les plus courts chemins partant de s_0. Dans ce cas, il est également possible d'utiliser l'algorithme A* que vous étudierez dans le cours d'Intelligence Artificielle. Cet algorithme est généralement plus efficace dès lors qu'il est possible de calculer efficacement une borne minimale de la longueur d'un plus court chemin entre deux points (par exemple, la distance euclidienne).
- Pour calculer tous les plus courts chemins entre tous les couples de sommets possibles, nous pourrions utiliser la résolution du problème à origine unique en résolvant le problème pour chaque sommet du graphe, ce qui rajoute un facteur n à la complexité de l'algorithme. Toutefois, il existe un algorithme plus efficace dans ce cas : l'algorithme de Floyd-Warshall, qui utilise un principe de programmation dynamique pour éviter de recalculer plusieurs fois des chemins. Cet algorithme ne sera pas étudié dans le contexte de ce cours, mais vous pouvez vous référer, par exemple, au livre de Cormen, Leiserson et Rivest pour plus de détails.
- Pour calculer non pas des plus courts chemins, mais des plus longs chemins, ou encore si la fonction définissant le coût d'un chemin est différente (s'il s'agit, par exemple, d'un produit, un max ou un min des coûts des arcs du chemin), il faudra adapter les algorithmes que nous allons étudier. Ces conditions d'adaptation seront discutées après chaque présentation d'algorithme.

Arborescence des plus courts chemins. Nous allons en fait calculer non seulement les coûts des plus courts chemins, mais aussi les sommets présents sur ces plus courts chemins. La représentation utilisée pour représenter ces plus courts chemins est la même que celle utilisée pour les arborescences calculées lors d'un parcours en largeur ou en profondeur d'un graphe. Cette arborescence est mémorisée dans un tableau π tel que

- $\pi[s_0] = null$
- $\pi[s_j] = s_i$ si $s_i \to s_j$ est un arc de l'arborescence.

Pour connaitre le plus court chemin entre s_0 et un sommet s_k donné, il faudra alors "remonter" de s_k jusque s_0 en utilisant π, comme décrit dans l'algorithme 3 introduit au chapitre 4.

Considérons par exemple le graphe de la figure 5.1(a). Ce graphe possède plusieurs arborescences des plus courts chemins dont l'origine est a :



La première de ces 2 arborescences est représentée par le tableau π tel que $\pi[a] = null$, $\pi[b] = a$, $\pi[c] = b$, $\pi[d] = e$ et $\pi[e] = a$.

Propriété d'optimalité des sous-chemins. Les différents algorithmes que nous allons utiliser supposent que tout sous-chemin d'un plus court chemin est également être un plus court chemin. Cette propriété peut être facilement démontrée : imaginons qu'un plus court chemin $p = i \rightsquigarrow j \rightsquigarrow k \rightsquigarrow l$ allant de i à l en passant par j et k soit tel que le sous-chemin allant de j à k ne soit pas un plus court chemin allant de j à k ; dans ce cas, p n'est pas le plus court chemin de i à l puisque nous pouvons le raccourcir en remplaçant le sous-chemin de j à k par le plus court chemin allant de j à k.

Pour tout sommet $s_i \neq s_0$, nous avons $\delta(s_0, s_i) = \delta(s_0, \pi[s_i]) + cout(\pi[s_i], s_i)$.

Notons que cette propriété peut ne plus être vérifiée dès lors que des contraintes sont ajoutées. Imaginons par exemple que nous cherchions le plus court chemin empruntant au plus x arcs. Dans ce cas, il est possible qu'un plus court chemin $p = i \rightsquigarrow j \rightsquigarrow k$ allant de i à k en passant par j soit tel qu'il existe un chemin p' plus court pour aller de i à j que celui emprunté par p car ce chemin p' emprunte plus d'arcs que p entre i et j de sorte qu'il ne peut pas être complété par la partie de p allant de j à k sans violer la contrainte imposant de ne pas passer par plus de x arcs.

5.2 Principe commun aux algorithmes de recherche de plus courts chemins

Nous allons étudier trois algorithmes qui permettent de résoudre des problèmes de recherche de plus courts chemins à origine unique :

- un algorithme (Dijkstra) qui peut être utilisé dès lors que tous les coûts sont positifs ou nuls ;
- un algorithme qui peut être utilisé dès lors que le graphe ne comporte pas de circuit ;
- un algorithme (Ford-Bellman) qui peut être utilisé pour n'importe quel graphe ne comportant pas de circuit absorbant.

Les trois algorithmes procèdent de la même façon, en associant à chaque sommet s_i une borne supérieure $d[s_i]$ du coût du plus court chemin de s_0 jusque s_i de sorte que, à tout moment, $d[s_i] \geq \delta(s_0, s_i)$. Au départ, ces bornes sont initialisées à $d[s_i] = +\infty$ pour tous les sommets s_i sauf pour le sommet initial s_0 pour lequel $d[s_0]$ est initialisé à $\delta(s_0, s_0) = 0$. Ensuite, les algorithmes vont grignoter progressivement ces bornes jusqu'à ce qu'elles ne puissent plus être diminuées et que $d[s_i] = \delta(s_0, s_i)$ pour chaque sommet s_i. Pour grignoter les valeurs de d, les algorithmes vont itérativement examiner chaque arc $s_i \rightarrow s_j$ du graphe, et regarder s'il est possible de diminuer la valeur de $d[s_j]$ en passant par s_i. Cette opération de grignotage est appelée "relâchement de l'arc (s_i, s_j)", et est décrite dans

l'algorithme 9.

Algorithme 9 : Relâchement de l'arc (s_i, s_j)

```
1 Procédure relâcher((s_i, s_j), π, d)
    Entrée          : Un arc (s_i, s_j)
    Entrée/Sortie   : Les tableaux d et π
    Précondition    : d[s_i] ≥ δ(s_0, s_i) et d[s_j] ≥ δ(s_0, s_j)
    Postcondition   : δ(s_0, s_j) ≤ d[s_j] ≤ d[s_i] + cout(s_i, s_j) et π[s_j] = s_i si la borne d[s_j] a été diminuée
2   début
3     si d[s_j] > d[s_i] + cout(s_i, s_j) alors
4       d[s_j] ← d[s_i] + cout(s_i, s_j)
5       π[s_j] ← s_i
```

La différence entre les trois algorithmes réside dans la stratégie utilisée pour décider de l'ordre de relâchement des arcs. Les deux premiers algorithmes exploitent des cas particuliers pour ne relâcher chaque arc qu'une seule fois : le premier exploite le fait que tous les coûts sont positifs, et le second exploite le fait que le graphe est un DAG. Le troisième algorithme n'exploite pas de cas particulier et va donc relâcher chaque arc plusieurs fois.

5.3 Algorithme de Dijkstra

L'algorithme de Dijkstra permet de calculer les plus courts chemins dans le cas où tous les coûts sont positifs, et peut être vu comme une généralisation du parcours en largeur au cas où les arcs ont des coûts. L'algorithme colorie les sommets selon le même principe que BFS :

- un sommet s_i est blanc s'il n'a pas encore été découvert ($d[s_i] = +\infty$) ;
- il est gris s'il a été découvert et sa borne peut encore diminuer ($\delta(s_0, s_i) \leq d[s_i] < +\infty$) ;
- il est noir si sa borne ne peut plus diminuer ($d[s_i] = \delta(s_0, s_i)$) et tous les arcs partant de lui ont été relâchés.

À chaque itération, l'algorithme choisit un sommet gris, relâche tous les arcs partant de ce sommet et le colorie en noir. Il utilise une stratégie dite **gloutonne** pour choisir ce sommet gris. Le principe des stratégies gloutonnes est d'utiliser un critère simple pour prendre une décision à chaque itération, décision qui n'est plus remise en cause ultérieurement. Ici, le critère utilisé est la borne d : l'algorithme choisit le sommet gris s_i ayant la plus petite valeur de d. Ce principe est décrit dans l'algorithme 10.

Correction de l'algorithme de Dijkstra : Pour démontrer la correction de cet algorithme, nous allons montrer que la propriété suivante est un invariant qui est vérifié à chaque passage à la ligne 8 : pour tout sommet s_j,

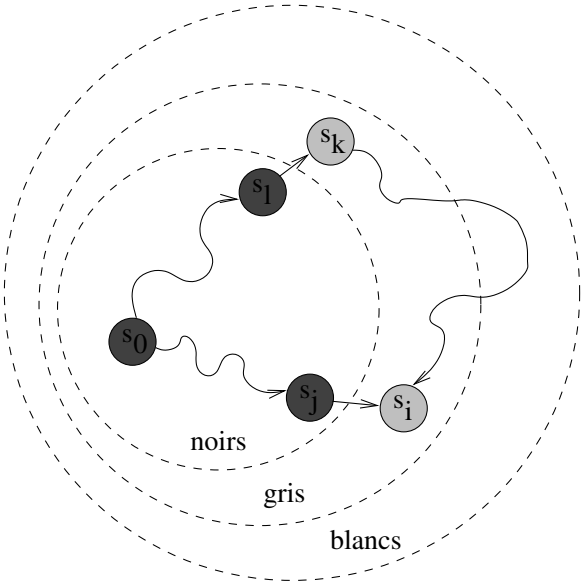
- si s_j est gris alors $d[s_j]$ = longueur du plus court chemin de s_0 vers s_j ne passant que par des sommets noirs (nous appellerons *chemin noir* un tel chemin) ;
- si s_j est noir alors $d[s_j] = \delta(s_0, s_j)$, et tous les successeurs de s_j sont gris ou noirs.

Au premier passage, tous les sommets sont blancs sauf s_0 qui est gris et $d[s_0] = 0 = \delta(s_0, s_0)$ de sorte que l'invariant est vérifié.

Supposons maintenant que l'invariant est vérifié au $k^{\text{ème}}$ passage et montrons qu'il sera vérifié au $k+1^{\text{ème}}$ passage. Lors du $k^{\text{ème}}$ passage dans la boucle lignes 8 à 14, le sommet s_i passe de gris à noir et il faut donc montrer que $d[s_i] = \delta(s_0, s_i)$. Comme ce sommet était gris, nous savons que $d[s_i]$ = longueur du plus court chemin noir de s_0 vers s_i. Soit $p_{noir} = s_0 \rightsquigarrow s_j \rightarrow s_i$ ce chemin, tel que tous les sommets de s_0 jusque s_j sont noirs. Tous les autres chemins allant de s_0 à s_i sont nécessairement de la forme $p' = s_0 \rightsquigarrow s_l \rightarrow s_k \rightsquigarrow s_i$ où tous les sommets de s_0 jusque s_l sont noirs, et s_k est le premier sommet non noir du chemin. s_k est nécessairement gris car aucun successeur d'un sommet noir n'est blanc. Comme s_i est le sommet gris pour lequel la borne d est minimale, nous savons que $d[s_k] \geq d[s_i]$. Comme les coûts de tous les arcs sont positifs ou nuls, le coût du sous-chemin $s_k \rightsquigarrow s_i$ est positif ou nul de sorte que le coût de p ne peut être inférieur à $d[s_i]$ et donc $d[s_i] = \delta(s_0, s_i)$.

Algorithme 10 : Recherche de plus courts chemins de Dijkstra

Fonction *Dijkstra*$(g, cout, s_0)$
  Entrée : Un graphe $g = (S, A)$, une fonction $cout : A \to \mathbb{R}$ et un sommet de départ $s_0 \in S$
  Précondition : Pour tout arc $(s_i, s_j) \in A, cout(s_i, s_j) \geq 0$
  Postcondition : Retourne une arborescence π des plus courts chemins partant de s_0 et un tableau d tel que $d[s_i] = \delta(s_0, s_i)$
  pour *chaque sommet* $s_i \in S$ **faire**
    $d[s_i] \leftarrow +\infty$
    $\pi[s_i] \leftarrow null$
    Colorier s_i en blanc
  $d[s_0] \leftarrow 0$
  Colorier s_0 en gris
  tant que *il existe un sommet gris* **faire**
    Soit s_i le sommet gris tel que $d[s_i]$ soit minimal
    pour *tout sommet* $s_j \in succ(s_i)$ **faire**
      si s_j *est blanc ou gris* **alors**
        relâcher$((s_i, s_j), \pi, d)$
        si s_j *est blanc* **alors** Colorier s_j en gris;
    Colorier s_i en noir
  retourne π *et* d



Il nous reste à montrer que, pour chaque sommet s_j gris, $d[s_j]$=longueur du plus court chemin noir de s_0 vers s_j. Soit s_j un sommet voisin du sommet s_i dont tous les arcs ont été relâchés.

— Si s_j était blanc au début de l'itération, alors cela implique qu'il existe un seul chemin noir $s_0 \rightsquigarrow s_i \to s_j$. Au moment où l'arc (s_i, s_j) est relâché, $d[s_j]$ est affecté à la longueur de ce chemin, c'est-à-dire, $d[s_i] + cout(s_i, s_j)$.

— Si s_j était gris au début de l'itération, alors cela implique qu'il existait déjà un chemin noir $s_0 \rightsquigarrow s_j$. Si ce chemin est plus court que le chemin passant par s_i alors $d[s_j]$ n'est pas modifié par le relâchement de l'arc

(s_i, s_j). Sinon, le relâchement de (s_i, s_j) met à jour $d[s_j]$ de sorte qu'il soit égal à la longueur du chemin $s_0 \rightsquigarrow s_i \rightarrow s_j$ qui devient le plus court chemin noir de s_0 jusque s_j.

Complexité : Soient n et p le nombre de sommets et arcs du graphe, respectivement. À chaque passage dans la boucle lignes 8 à 14, exactement un sommet est colorié en noir, et ne pourra plus jamais être recolorié en noir puisque seuls les sommets gris sont coloriés en noir. L'algorithme passera donc au plus n fois dans la boucle (il peut passer moins de n fois si certains sommets ne sont pas accessibles depuis s_0). À chaque passage dans la boucle, il faut chercher le sommet gris ayant la plus petite valeur de d, puis relâcher tous les arcs partant de ce sommet et arrivant sur un sommet non noir. Si la recherche du sommet ayant la plus petite valeur de d est faite linéairement, alors la complexité de Dijkstra est $\mathcal{O}(n^2)$. Cette complexité peut être améliorée en utilisant un *tas binaire* : comme vous l'avez vu au semestre 1, un tas binaire permet de trouver le plus petit élément d'un ensemble en temps constant ; en revanche, l'ajout, la suppression ou la modification d'un élément dans un tas binaire comportant n éléments est en $\mathcal{O}(log(n))$. Dans ce cas, la complexité de Dijkstra est $\mathcal{O}((n+p)log(n))$.

▶ Extension de Dijkstra au calcul de meilleurs chemins

L'algorithme 10 calcule des plus courts chemins dans le cas où la longueur d'un chemin est définie par la somme des coûts de ses arcs. Nous pouvons nous demander s'il est possible d'utiliser le même principe pour calculer d'autres "meilleurs chemins". Par exemple, si les coûts des arcs sont des valeurs réelles comprises entre 0 et 1, correspondant à la probabilité de sureté des arcs, est-il possible de chercher le chemin le plus sûr, c'est-à-dire, le chemin dont le produit des coûts des arcs soit maximal ?

Pour répondre à cette question, il faut réfléchir à la propriété de la fonction de coût qui nous permet de prouver la correction de l'algorithme de Dijkstra, à savoir que l'ajout d'un arc (s_i, s_j) derrière un chemin $s_0 \rightsquigarrow s_i$ ne peut que dégrader la qualité du chemin, autrement dit, si l'objectif est de calculer un plus court chemin, alors il faut que $cout(s_0 \rightsquigarrow s_i \rightarrow s_j) \geq cout(s_0 \rightsquigarrow s_i)$, tandis que si l'objectif est de calculer un plus long chemin, alors il faut que $cout(s_0 \rightsquigarrow s_i \rightarrow s_j) \leq cout(s_0 \rightsquigarrow s_i)$, pour tout chemin $s_0 \rightsquigarrow s_i$ et tout arc (s_i, s_j).

Nous pouvons vérifier que cette propriété est satisfaite dans le cas où les coûts des arcs représentent des probabilités comprises entre 0 et 1, et le coût d'un chemin est défini par le produit des arcs empruntés, de sorte que nous pourrons utiliser Dijkstra pour chercher des plus longs chemins. Il faudra toutefois adapter la procédure de relâchement, comme décrit dans l'algorithme 11. Il faudra également initialiser la borne d à la plus petite valeur possible, c'est-à-dire 0, sauf pour le sommet de départ s_0 pour lequel la valeur de d sera initialisée à 1.

Algorithme 11 : Relâchement de l'arc (s_i, s_j) pour chercher le chemin maximisant le produit des coûts de ses arcs

```
1  Procédure relâcher((s_i, s_j), π, d)
       Entrée          : Un arc (s_i, s_j)
       Entrée/Sortie   : Les tableaux d et π
       Précondition    : d[s_i] ≤ δ(s_0, s_i) et d[s_j] ≤ δ(s_0, s_j)
       Postcondition   : δ(s_0, s_j) ≥ d[s_j] ≥ d[s_i] * cout(s_i, s_j) et π[s_j] = s_i si la borne d[s_j] a été augmentée
2      début
3          si d[s_j] < d[s_i] * cout(s_i, s_j) alors
4              d[s_j] ← d[s_i] * cout(s_i, s_j)
5              π[s_j] ← s_i
```

5.4 Recherche de plus courts chemins dans un DAG

Rappelons qu'un DAG est un graphe orienté sans circuit. Cette propriété peut être exploitée pour calculer les plus courts chemins en ne relâchant chaque arc qu'une seule fois. Contrairement à l'algorithme de Dijkstra, nous n'avons plus de précondition sur les coûts des arcs qui pourront être positifs ou négatifs. Comme pour Dijkstra, l'idée est de relâcher les arcs partant d'un sommet s_i dès lors que $d[s_i] = \delta(s_0, s_i)$. Pour cela, il suffit de ne relâcher les arcs partant d'un sommet que si tous les arcs se trouvant sur un chemin entre s_0 et s_i ont déjà été relâchés. Nous avons vu en 4.2 que nous pouvons utiliser un parcours en profondeur pour trier les sommets d'un DAG de telle sorte que pour tout arc (s_i, s_j), le sommet s_i apparaisse avant le sommet s_j. Si nous considérons les sommets selon un tel ordre, et si nous relâchons à chaque fois les arcs partant du sommet considéré, alors nous obtiendrons les plus courts chemins, comme décrit dans l'algorithme 12. Cet algorithme suppose que le sommet de départ s_0 ne possède pas de prédécesseur. En effet, si s_0 possède des prédécesseurs, alors il n'existe pas de chemin allant de s_0 jusque ces prédécesseurs (car le graphe n'a pas de circuit) de sorte que cela n'a pas de sens de considérer ces sommets au moment de calculer les plus courts chemins partant de s_0. Cet algorithme suppose également que le sommet s_0 est le seul sommet ne possédant pas de prédécesseur, pour les mêmes raisons (s'il existe un autre sommet que s_0 n'ayant pas de prédécesseur, alors ce sommet n'est pas accessible depuis s_0).

Algorithme 12 : Recherche de plus courts chemins dans un DAG

1 **Fonction** *topoDAG*$(g, cout, s_0)$
  Entrée : Un graphe $g = (S, A)$, une fonction $cout : A \to \mathbb{R}$ et un sommet de départ $s_0 \in S$
  Précondition : g est un DAG et s_0 est le seul sommet de g ne possédant pas de prédécesseur
  Postcondition : Retourne une arborescence π des plus courts chemins partant de s_0 et un tableau d tel que $d[s_i] = \delta(s_0, s_i)$
2 **pour** *chaque sommet* $s_i \in S$ **faire**
3 $d[s_i] \leftarrow +\infty$
4 $\pi[s_i] \leftarrow null$
5 $d[s_0] \leftarrow 0$
6 Trier topologiquement les sommets de g à l'aide d'un parcours en profondeur
7 **pour** *chaque sommet* s_i *pris selon l'ordre topologique* **faire**
8 **pour** *chaque sommet* $s_j \in succ(s_i)$ **faire** relâcher$((s_i, s_j), \pi, d)$;
9 **retourne** π *et* d

Complexité. Soient n et p le nombre de sommets et arcs du graphe, respectivement. La complexité du tri topologique est $\mathcal{O}(n + p)$. L'algorithme passera exactement n fois dans la boucle lignes 7 à 8, et chaque arc sera relâché très exactement une fois. Donc, la complexité de l'algorithme 12 est $\mathcal{O}(n + p)$.

▶ Extension au calcul de meilleurs chemins : application à la planification de projets

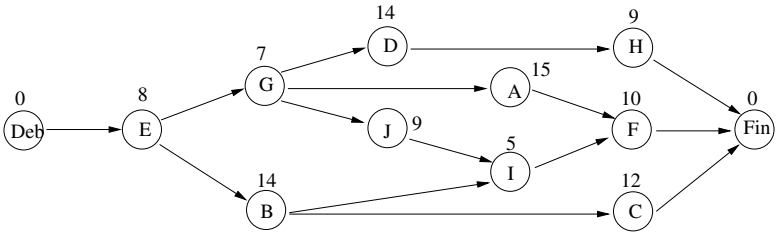
La seule condition d'application de l'algorithme 12 est que le graphe ne doit pas avoir de circuits. Nous pouvons donc facilement étendre ce principe à la recherche de plus longs chemins. Nous allons illustrer cela sur une application à la planification de projets.
Un projet est généralement décomposé en tâches, chaque tâche ayant une durée. La planification de projets a pour objectif de déterminer pour chaque tâche sa date de début de réalisation. Cette date doit être fixée en tenant compte d'un certain nombre de contraintes :

— des contraintes de précédence, imposant que certaines tâches ne peuvent commencer que si d'autres tâches sont terminées ;

— des contraintes de localisation temporelle, imposant qu'une tâche soit réalisée à l'intérieur d'une période de temps donnée ;
— des contraintes d'exclusion, interdisant que certaines tâches soient exécutées en même temps ;
— des contraintes cumulatives, traduisant le fait que certaines ressources sont limitées de sorte qu'à tout moment la quantité de ces ressources demandée par les tâches en cours de réalisation soit inférieure à la quantité disponible ;
— etc.

Quand les seules contraintes posées sont des contraintes de précédence, nous pouvons utiliser l'algorithme 12 pour planifier le projet[1]. Dans ce cas, nous pouvons définir le graphe associant un sommet à chaque tâche et un arc à chaque contrainte de précédence entre tâches. Deux sommets supplémentaires s_{deb} et s_{fin} sont ajoutés pour représenter le début et la fin du projet, comme illustré ci-dessous.

Tâche	Durée	Contraintes
A	15	G achevée
B	14	E achevée
C	12	B achevée
D	14	G achevée
E	8	
F	10	A et I achevées
G	7	E achevée
H	9	D achevée
I	5	B et J achevées
J	9	G achevée



Les durées des tâches sont associées aux sommets, et les durées des deux sommets supplémentaires représentant le début et la fin du projet sont nulles.

La **date au plus tôt** d'une tâche est le temps minimum qui sépare le début de cette tâche du début du projet, et correspond à la longueur du plus long chemin allant de s_{deb} jusqu'au sommet associé à cette tâche, la longueur d'un chemin étant définie comme la somme des durées associées aux sommets de ce chemin à l'exclusion du dernier sommet. Etant donné que le graphe est acyclique (sans quoi le problème est insoluble), nous pouvons adapter l'algorithme 12 pour calculer ces plus longs chemins, comme décrit dans l'algorithme 13.

Algorithme 13 : Calcul de dates au plus tôt

1 **Fonction** *datesAuPlusTot*$(g, duree, s_{deb}, s_{fin})$
  Entrée : Un graphe $g = (S, A)$, une fonction $duree : S \to \mathbb{R}^+$, et deux sommets $s_{deb} \in S$ et $s_{fin} \in S$
  Précondition : g est un DAG, s_{deb} est le seul sommet de g n'ayant pas de prédécesseur et s_{fin} le seul n'ayant pas de successeur
  Postcondition : Retourne un tableau d tel que $d[s_i] =$ date au plus tôt de la tâche associée à s_i
2 **pour** *chaque sommet* $s_i \in S$ **faire** $d[s_i] \leftarrow 0$;
3 Trier topologiquement les sommets de g à l'aide d'un parcours en profondeur
4 **pour** *chaque sommet* s_i *pris selon l'ordre topologique* **faire**
5 **pour** *chaque sommet* $s_j \in succ(s_i)$ **faire**
6 **si** $d[s_i] + duree(s_i) > d[s_j]$ **alors** $d[s_j] \leftarrow d[s_i] + duree(s_i)$;
7 **retourne** d

1. Les contraintes d'exclusion et les contraintes cumulatives (généralement appelées contraintes disjonctives) rendent le problème $\mathcal{NP}$-difficile (cf chapitre 7).

Sur l'exemple précédent, un ordre topologique est : Deb, E, G, D, B, A, J, C, I, H, F, Fin
et les dates au plus tôt sont :

tâche	Deb	E	G	D	B	A	J	C	I	H	F	Fin
date au plus tôt	0	0	8	15	8	15	15	22	24	29	30	40

Nous pouvons ensuite calculer la **date au plus tard** de chaque tâche i, c'est-à-dire le temps maximum qui peut séparer le début de l'exécution de i du début du projet, sans augmenter la durée totale du projet. Il suffit pour cela de calculer la longueur du plus long chemin entre i et le sommet supplémentaire s_{fin} associé à la fin du projet, et retrancher cette longueur de la date de fin du projet. Pour calculer les longueurs de tous les chemins partant de chaque tâche jusque s_{fin}, nous pouvons inverser le sens de tous les arcs et appliquer l'algorithme 13.

Sur l'exemple précédent, les dates au plus tard sont :

tâche	Deb	E	G	D	B	A	J	C	I	H	F	Fin
date au plus tard	0	0	8	17	11	15	16	28	25	31	30	40

En retranchant la date au plus tot de la date au plus tard d'une tâche, nous obtenons la **marge totale** associée à cette tâche, c'est-à-dire le battement maximum dont nous disposons, en plus de la durée propre de la tâche, pour fixer sa réalisation, sans pour autant perturber la date finale de fin du projet.

Sur l'exemple précédent, les marges totales sont :

tâche	Deb	E	G	D	B	A	J	C	I	H	F	Fin
Marge totale	0	0	0	2	3	0	1	6	1	2	0	0

Ainsi, la tâche H a une marge totale de 2, ce qui signifie que le début de son exécution peut être reculé de 2 unités de temps après avoir fini l'exécution de la tâche précédente D (ou encore, que la durée de réalisation de H peut être augmentée de 2 unités de temps), sans pour autant retarder la date de fin du projet.

Notons que la marge totale suppose que tout ce qui a précédé la tâche i se soit toujours accompli le plus tôt possible, tandis que tout ce qui suit sera accompli le plus tard possible. Une conséquence de cela est que les "marges totales se partagent", ce qui signifie qu'elles ne peuvent être utilisées qu'une seule fois par toutes les tâches d'un chemin non ramifié. Sur notre exemple, les tâches D et H ont toutes les deux une marge totale de 2. Néanmoins, si le début de D est retardé de 2 unités de temps, alors il n'y aura plus aucune marge pour l'exécution de la tâche suivante H.

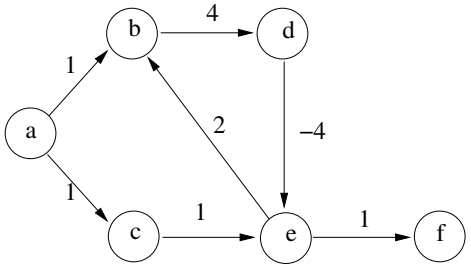
Certaines tâches ne disposent d'aucune marge pour leur réalisation. Le moindre retard dans la réalisation d'une telle tâche entrainera nécessairement un retard global du projet. C'est pourquoi de telles tâches sont dites **tâches critiques**.

Sur l'exemple précédent, les tâches E, G, A et F sont critiques. La réalisation de ces tâches devra être particulièrement surveillée car le moindre retard se transmettra à l'ensemble du projet.

D'une façon plus générale, un **chemin critique** est un chemin dans le graphe allant du sommet de début s_{deb} au sommet de fin s_{fin} et ne passant que par des tâches critiques. Tout projet possèdera au moins un chemin critique, correspondant au plus long chemin entre les sommets de début et de fin du projet. Certains projets peuvent avoir plusieurs chemins critiques, s'il y a plusieurs plus longs chemins.

5.5 Algorithme de Bellman-Ford

L'algorithme que nous venons de voir ne peut être utilisé que si le graphe ne comporte pas de circuit. Quand le graphe comporte des circuits, nous pouvons utiliser l'algorithme de Dijkstra, mais il faut dans ce cas que la fonction coût soit telle que l'ajout d'un arc à la fin d'un chemin ne puisse que dégrader le coût du chemin. Considérons par exemple le graphe suivant :



Les coûts de certains arcs sont négatifs. Si le coût d'un chemin est défini par la somme des coûts de ses arcs, alors l'ajout d'un arc de coût négatif à la fin d'un chemin diminue le coût du chemin. Si nous exécutons l'algorithme de Dijkstra sur ce graphe, à partir du sommet a, l'algorithme va trouver le chemin $< a, c, e, f >$ pour aller de a à f (car il relâchera les arcs partant de e avant de relâcher les arcs partant de d), alors qu'il existe un chemin plus court ($< a, b, d, e, f >$). Nous ne pouvons pas non plus appliquer TopoDAG car le graphe comporte un circuit.
L'algorithme de Bellman-Ford permet de trouver les plus courts chemins à origine unique dans le cas où le graphe contient des arcs dont le coût est négatif, sous réserve que le graphe ne contienne pas de circuit absorbant (dans ce cas, l'algorithme de Bellman-Ford va détecter l'existence de circuits absorbants).
L'algorithme fonctionne selon le même principe que les deux algorithmes précédents, en grignotant les bornes d par des relâchement d'arcs. Cependant, chaque arc va être relâché plusieurs fois : à chaque itération, tous les arcs sont relâchés. Ce principe est décrit dans l'algorithme 14.

Algorithme 14 : Recherche de plus courts chemins par Bellman-Ford

1 **Fonction** *Bellman-Ford*($g, cout, s_0$)
  Entrée : Un graphe $g = (S, A)$, une fonction $cout : A \to \mathbb{R}$ et un sommet de départ $s_0 \in S$
  Postcondition : Retourne une arborescence π des plus courts chemins partant de s_0 et un tableau d tel que $d[s_i] = \delta(s_0, s_i)$. Affiche un message si g contient un circuit absorbant
2 **pour** *chaque sommet* $s_i \in S$ **faire**
3 $d[s_i] \leftarrow +\infty$
4 $\pi[s_i] \leftarrow null$
5 $d[s_0] \leftarrow 0$
6 **pour** k *variant de* 1 *à* $\mid S \mid -1$ **faire**
7 **pour** *chaque arc* $(s_i, s_j) \in A$ **faire** relâcher($(s_i, s_j), \pi, d$);
8 **si** $\exists$ *un arc* $(s_i, s_j) \in A$ *tel que* $d[s_j] > d[s_i] + cout(s_i, s_j)$ **alors**
9 afficher("Le graphe contient un circuit absorbant")
10 **retourne** π *et* d

Correction de l'algorithme. Pour prouver la correction de l'algorithme, nous allons montrer que la propriété suivante est vérifiée à chaque passage à la ligne 6 : pour tout sommet s_i, $d[s_i]$ = longueur du plus court chemin de s_0 jusque s_i comportant au plus k arcs.
Au premier passage, $k = 0$ et $d[s_0] = 0 = \delta(s_0, s_0)$. À chaque passage, tous les arcs sont relâchés. Soit $p = s_0 \rightsquigarrow s_i$ un plus court chemin comportant k arcs, et soit (s_j, s_i) le dernier arc de ce chemin, de sorte que $p = s_0 \rightsquigarrow s_j \to s_i$. Le chemin $s_0 \rightsquigarrow s_j$ comporte $k - 1$ arcs de sorte qu'après $k - 1$ itérations, $d[s_j] = \delta(s_0, s_j)$. À l'itération suivante, l'arc (s_j, s_i) est relâché et donc $d[s_i] = \delta(s_0, s_i)$.
Si le graphe ne comporte pas de circuit absorbant, un plus court chemin est nécessairement élémentaire et a au plus $|S| - 1$ arcs. Par conséquent, après $|S| - 1$ itérations l'algorithme aura trouvé tous les plus courts chemins partant de s_0. Si le graphe contient un circuit absorbant, il y aura encore au moins un arc (s_i, s_j) pour lequel un relâchement

permettrait de diminuer la valeur de $d[s_j]$. L'algorithme utilise cette propriété pour détecter la présence de circuits absorbants.

Complexité. Si le graphe comporte n sommets et p arcs, chaque arc sera relâché n fois, de sorte que l'algorithme effectuera np appels à la procédure de relâchement. Par conséquent, la complexité est $\mathcal{O}(np)$. En pratique, l'algorithme peut être arrêté dès lors qu'aucune valeur de d n'a été modifiée pendant une itération complète. Il est également possible de mémoriser à chaque itération l'ensemble des sommets pour lesquels la valeur de d a changé, afin de ne relâcher lors de l'itération suivante que les arcs partant de ces sommets. Ces deux améliorations ne changent pas la complexité de l'algorithme, mais peuvent l'accélérer en pratique.

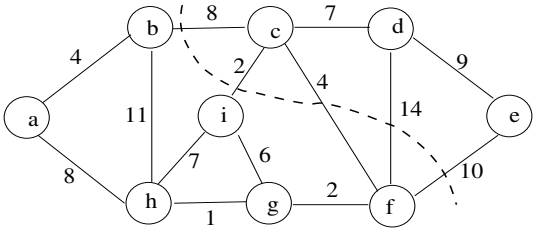
Adaptation de Bellman-Ford. L'algorithme peut être facilement adapté pour calculer d'autres "meilleurs" chemins, qu'il s'agisse de plus courts ou de plus longs chemins, et pour différentes fonctions définissant le coût d'un chemin (par exemple, la somme, le produit ou encore le min ou le max des coûts des arcs du chemin). Il faudra changer l'initialisation de la borne d et adapter la procédure de relâchement.
Rappelons qu'une précondition forte à l'utilisation de tous les algorithmes étudiés dans ce chapitre est que tout sous-chemin d'un plus court chemin doit également être un plus court chemin. Cette propriété peut ne plus être vérifiée si des contraintes sont ajoutées. Supposons par exemple qu'à chaque arc (s_i, s_j) soient associés un coût $cout(i, j)$ et une durée $duree(i, j)$, et que nous recherchions le chemin de s_0 à s_k minimisant la somme des coûts des arcs et tel que la somme des durées des arcs soit inférieure ou égale à une borne donnée. Dans ce cas, le problème devient $\mathcal{NP}$-difficile (cf chapitre 7) et les algorithmes étudiés dans ce chapitre ne peuvent plus être utilisés.

CHAPITRE 6
ARBRES COUVRANTS MINIMAUX

Un arbre couvrant minimal (Minimal Spanning Tree / MST) d'un graphe non orienté $G = (S, A)$ est un graphe partiel $G' = (S, A')$ de G tel que G' est un arbre couvrant (autrement dit, G' est connexe et sans cycle), et la somme des coûts des arêtes de A' est minimale.

6.1 Principe générique

Dans ce chapitre, nous allons étudier deux algorithmes permettant de calculer des MST. Les deux algorithmes fonctionnent selon un principe glouton décrit dans l'algorithme 15 : l'idée est de sélectionner, à chaque itération, une arête de coût minimal traversant une coupure.
Une **coupure** d'un graphe $G = (S, A)$ est une partition de l'ensemble des sommets en deux parties $(P, S \setminus P)$. Une arête (s_i, s_j) **traverse** une coupure $(P, S \setminus P)$ si chaque extrémité de l'arête appartient à une partie différente, *i.e.*, $s_i \in P$ et $s_j \in S \setminus P$, ou $s_j \in P$ et $s_i \in S \setminus P$. Une coupure **respecte** un ensemble d'arêtes E si aucune arête de E n'est traversée par la coupure.
Considérons par exemple le graphe suivant :



La coupure $(\{a, b, f, g, h, i\}, \{c, d, e\})$ est représentée en pointillés et traverse les arêtes $\{b, c\}$, $\{i, c\}$, $\{c, f\}$, $\{d, f\}$ et $\{e, f\}$. L'arête de coût minimal traversant cette coupure est $\{i, c\}$.

Correction de l'algorithme générique. Pour montrer que l'algorithme est correct, nous allons montrer qu'à chaque passage à la ligne 3, il existe un MST dont les arêtes sont un sur-ensemble de E. Au premier passage, E est vide et l'invariant est donc vérifié. Supposons qu'il soit vérifié au $k^{\text{ème}}$ passage lorsque E contient $k - 1$ arêtes, et notons (S, E') un MST contenant E. Montrons que l'invariant est vérifié au $k + 1^{\text{ème}}$ passage, après avoir ajouté dans E une arête $\{s_i, s_j\}$ de coût minimal traversant une coupure $(P, S \setminus P)$ respectant E. Nous avons deux cas possibles : soit $\{s_i, s_j\} \in E'$, et dans ce cas l'invariant reste trivialement vérifié ; soit $\{s_i, s_j\} \notin E'$, et dans ce cas nous devons montrer que l'invariant reste vérifié, c'est-à-dire qu'il existe un autre MST (S, E'') contenant $\{s_i, s_j\}$. Comme (S, E')

Algorithme 15 : Principe glouton générique pour calculer un MST

1 **Fonction** *MSTgénérique*$(g, cout)$
  Entrée : Un graphe $g = (S, A)$ et une fonction $cout : A \to \mathbb{R}$
  Postcondition : Retourne un ensemble d'arêtes $E \subseteq A$ tel que (S, E) est un MST de g
2 $E \leftarrow \emptyset$
3 **tant que** $\mid E \mid < \mid S \mid -1$ **faire**
4 Soit $(P, S \setminus P)$ une coupure (quelconque) qui respecte E
5 Ajouter dans E une arête de coût minimal traversant la coupure $(P, S \setminus P)$
6 **retourne** E

est un arbre couvrant, le graphe $(S, E' \cup \{(s_i, s_j)\})$ contient un et un seul cycle, noté c, et ce cycle contient l'arête $\{s_i, s_j\}$. Comme $\{s_i, s_j\}$ traverse la coupure $(P, S \setminus P)$, il existe nécessairement une autre arête $\{s_k, s_l\}$ de c qui traverse également la coupure. Définissons $E'' = \{\{s_i, s_j\}\} \cup E' \setminus \{\{s_k, s_l\}\}$. Comme $\{s_i, s_j\}$ est une arête de coût minimal, parmi l'ensemble des arêtes traversant la coupure, nous avons $cout(s_i, s_j) \leq cout(s_k, s_l)$, et par conséquent (S, E'') est un MST.

Mise en œuvre. Toute la difficulté réside dans la recherche efficace, à chaque itération, d'une arête de coût minimal traversant une coupure respectant E. Nous allons voir deux façons différentes de faire cela : dans l'algorithme de Kruskal, l'ensemble E forme une forêt et l'arête sélectionnée est une arête de coût minimal permettant de connecter deux arbres différents ; dans l'algorithme de Prim, l'ensemble E est un arbre et l'arête sélectionnée est une arête de coût minimal connectant un sommet de l'arbre à un sommet n'appartenant pas à l'arbre.

6.2 Algorithme de Kruskal

L'algorithme de Kruskal, décrit dans l'algorithme 16 est un cas particulier de l'algorithme 15 où le graphe partiel (S, E) est une forêt, dont chaque composante connexe est un arbre. Initialement, E est vide de sorte que la forêt comporte un arbre par sommet, chaque arbre étant réduit à sa racine. L'algorithme sélectionne ensuite itérativement une arête de coût minimal parmi l'ensemble des arêtes permettant de connecter deux arbres différents. Pour cela, les arêtes du graphe sont triées par ordre de coût croissant et les arêtes sont considérées une par une selon cet ordre : si les deux sommets de l'arête courante $\{s_i, s_j\}$ appartiennent à deux arbres différents, alors elle est ajoutée à E et les deux arbres sont connectés. Notons que la coupure n'est pas maintenue explicitement dans cet algorithme, mais elle est implicitement définie par les deux arbres. Soit S_1 l'ensemble des sommets du premier arbre. La coupure $(S_1, S \setminus S_1)$ respecte E et l'arête $\{s_i, s_j\}$ est bien une arête de coût minimal traversant $(S_1, S \setminus S_1)$.
La difficulté majeure consiste à déterminer efficacement si s_i et s_j appartiennent à deux arbres différents. Nous utilisons pour cela une structure de données initialement introduite pour représenter une partition d'un ensemble : une partition est représentée par une forêt dont les sommets correspondent aux éléments de l'ensemble, chaque arbre de la forêt correspondant à un sous-ensemble différent. Cette forêt est représentée par un vecteur π tel que si $\pi[s_i] = null$ alors s_i est la racine d'un arbre, et si $\pi[s_i] = s_j \neq null$, alors s_j est le père de s_i dans un arbre. Pour savoir si deux sommets s_i et s_j appartiennent à un même sous-ensemble, il suffit de remonter dans le vecteur π de s_i jusqu'à la racine r_i de l'arbre contenant s_i, puis de s_j jusqu'à la racine r_j de l'arbre contenant s_j, et enfin de comparer r_i et r_j. Cette opération de recherche de racine est effectuée par la fonction *racine*. La complexité de cette fonction dépend de la profondeur de l'arbre. Afin de limiter cette profondeur, les sommets se trouvant sur le chemin entre l'élément de départ s et la racine r de l'arbre contenant s sont directement rattachés sous r (ligne 18). Comme la profondeur augmente (potentiellement) lorsque deux arbres sont fusionnés, nous pouvons également limiter l'augmentation de la profondeur en rattachant l'arbre le moins profond sous l'arbre le plus profond, lignes 12 à 14. Cette profondeur n'est pas calculée de façon exacte, mais une borne supérieure p est maintenue incrémentalement : cette borne est incrémentée

Algorithme 16 : Calcul d'un MST par Kruskal

```
1  Fonction Kruskal(g, cout)
       Entrée      : Un graphe g = (S, A) et une fonction cout : A → ℝ
       Postcondition : Retourne un ensemble d'arêtes E ⊆ A tel que (S, E) est un MST de g
       Déclaration : Une arborescence π
                     Un vecteur p tel que p[r_i] est une borne supérieure de la profondeur de l'arbre de racine r_i
2      pour chaque sommet s_i ∈ S faire
3          π[s_i] ← null
4          p[s_i] ← 0
5      E ← ∅
6      Trier les arêtes de A par ordre de coût croissant
7      pour chaque arête {s_i, s_j} prise par ordre de coût croissant et tant que |E| < |S| − 1 faire
8          r_i ←racine(π, s_i)
9          r_j ←racine(π, s_j)
10         si r_i ≠ r_j alors
11             Ajouter (s_i, s_j) dans E
12             si p[r_i] < p[r_j] alors π[r_i] ← r_j;
13             sinon si p[r_j] < p[r_i] alors π[r_j] ← r_i;
14             sinon π[r_j] ← r_i ; p[r_i] ← p[r_i] + 1;
15     retourne E
16 Fonction racine(π, s)
       Entrée/Sortie : Une forêt π
       Entrée        : Un sommet s
       Postcondition : Retourne la racine r de l'arborescence contenant s et la met à jour de sorte que tous les
                       sommets se trouvant entre s et r soient directement rattachés sous r
17     si π[s] = null alors retourne s;
18     π[s] ← racine(π, π[s])
19     retourne π[s]
```

à chaque fois que deux arbres de même profondeur sont fusionnés, ligne 14 (il s'agit d'une borne et non d'une valeur exacte car la fonction *racine* peut réduire la profondeur des arbres en rattachant certains sommets directement sous la racine).

Complexité : Soient n et p le nombre de sommets et arêtes, respectivement. Le tri des arêtes peut être fait en $\mathcal{O}(p \log p)$, à l'aide d'un *quicksort*, *mergesort* ou *heapsort*, par exemple. Si les coûts ont des valeurs entières comprises dans un intervalle $[min, max]$, il est possible d'utiliser un tri par dénombrement en $\mathcal{O}(p + k)$ où $k = max - min$[1]. La boucle lignes 7 à 14 est exécutée p fois dans le pire des cas. À chaque fois, il faut remonter des sommets s_i et s_j jusqu'aux racines des arbres correspondants. En rattachant directement les sommets visités sous la racine (dans la fonction *racine*), et en rattachant l'arbre le moins profond sous l'arbre le plus profond, cette opération peut être faite en $\mathcal{O}(\log p)$ (voir le livre de Cormen, Leiserson et Rivest pour plus de détails sur la gestion de π). Par conséquent, la complexité de l'algorithme de Kruskal est $\mathcal{O}(p \log p)$.

1. Le tri par dénombrement fait un premier parcours du tableau à trier pour compter le nombre d'occurrences de chaque valeur, puis un deuxième parcours pour remplir le tableau avec les valeurs triées, en utilisant les nombres d'occurrences ; voir le livre de Cormen, Leiserson et Rivest pour plus de détails.

6.3 Algorithme de Prim

L'algorithme de Prim, décrit dans l'algorithme 17 est également un cas particulier de l'algorithme 15.

Algorithme 17 : Calcul d'un MST par Prim

1 **Fonction** *Prim*$(g, cout)$
  Entrée : Un graphe $g = (S, A)$ et une fonction $cout : A \to \mathbb{R}$
  Postcondition : Retourne un ensemble d'arêtes $E \subseteq A$ tel que (S, E) est un MST de g
2 Soit s_0 un sommet de S choisi arbitrairement
3 $P \leftarrow \{s_0\}$
4 $E \leftarrow \emptyset$
5 **pour** *chaque sommet* $s_i \in S$ **faire**
6 **si** $s_i \in adj(s_0)$ **alors** $\pi[s_i] \leftarrow s_0$; $c[s_i] \leftarrow cout(s_0, s_i)$;
7 **sinon** $\pi[s_i] \leftarrow null$; $c[s_i] \leftarrow \infty$;
8 **tant que** $P \neq S$ **faire**
9 Ajouter dans P le sommet $s_i \in S \setminus P$ ayant la plus petite valeur de c
10 Ajouter $\{s_i, \pi[s_i]\}$ à E
11 **pour** *chaque sommet* $s_j \in adj(s_i)$ **faire**
12 **si** $s_j \notin P$ **et** $cout(s_i, s_j) < c[s_j]$ **alors**
13 $\pi[s_j] \leftarrow s_i$
14 $c[s_j] \leftarrow cout(s_i, s_j)$
15 **retourne** E

Cette fois ci, E est un ensemble d'arêtes connexes formant un arbre dont la racine est un sommet s_0 choisi arbitrairement au début de l'algorithme. La coupure considérée à chaque itération est $(P, S \setminus P)$ où P est l'ensemble des sommets qui appartiennent à l'arbre de racine s_0 et contenant les arêtes de E. Cette coupure respecte les arêtes de E puisque tous les sommets appartenant à une arête de E appartiennent à P. Pour trouver efficacement l'arête de coût minimal traversant la coupure, l'algorithme maintient deux tableaux c et π tels que, pour chaque sommet $s_i \in P$ pour lequel il existe une arête $\{s_i, s_j\}$ traversant la coupure,

— $\{s_i, \pi[s_i]\}$ est la plus petite arête partant de s_i et traversant la coupure $(P, S \setminus P)$;
— $c[s_i] = cout(s_i, \pi[s_i])$.

Les éléments du tableau c sont initialisés à ∞, sauf pour les sommets s_i adjacents à s_0 pour lesquels $c[s_i]$ est initialisé à $cout(s_0, s_i)$. À chaque fois qu'un sommet s_i est ajouté à P, chaque arête connectant s_i à un sommet s_j de $S \setminus P$ est utilisée pour mettre à jour $c[s_j]$ et $\pi[s_j]$ dans le cas où le coût de $\{s_i, s_j\}$ est inférieur à la valeur courante de $c[s_j]$.

Complexité : Soient n et p le nombre de sommets et arêtes, respectivement. L'algorithme passe $n - 1$ fois dans la boucle lignes 8 à 14 (initialement $P = \{s_0\}$, un sommet est ajouté à P à chaque passage, et l'itération s'arrête lorsque $P = S$). À chaque passage, il faut chercher le sommet de $S \setminus P$ ayant la plus petite valeur de c puis parcourir toutes les arêtes adjacentes à ce sommet. Si les sommets de $S \setminus P$ sont mémorisés dans un tableau ou une liste, la complexité est $\mathcal{O}(n^2)$. Cette complexité peut être améliorée en utilisant une file de priorité, implémentée par un tas binaire, pour gérer l'ensemble $S \setminus P$ (cf cours du premier semestre). Dans ce cas, la recherche du plus petit élément de $S \setminus P$ est faite en temps constant. En revanche, à chaque fois qu'une valeur du tableau c est diminuée, la mise à jour du tas est en $\mathcal{O}(\log n)$. Comme il y a au plus p mises à jour de c (une par arête), la complexité de Prim devient $\mathcal{O}(p \log n)$.

CHAPITRE 7

QUELQUES PROBLÈMES $\mathcal{NP}$-DIFFICILES SUR LES GRAPHES

Les algorithmes étudiés dans les chapitres précédents ont tous des complexités polynomiales qui sont bornées par $\mathcal{O}(n^k)$ où n dépend du graphe en entrée (son nombre de sommets et/ou son nombre d'arcs) et k est une constante indépendante du graphe en entrée. Les problèmes résolus par ces algorithmes sont donc considérés comme "faciles" dans le sens où ils peuvent être résolus en un temps "raisonnable", même pour de "gros" graphes.
Dans ce chapitre, nous allons aborder des problèmes plus difficiles, pour lesquels nous ne connaissons pas d'algorithme efficace, et il est conjecturé qu'il n'en existe pas. Nous allons tout d'abord introduire quelques éléments de complexité, permettant de définir la classe de ces problèmes. Nous étudierons ensuite quelques uns de ces problèmes.

7.1 Classes de complexité

Jusqu'ici la notion de complexité a été associée aux algorithmes. La complexité d'un algorithme donne un ordre de grandeur du nombre d'instructions qui seront effectuées lors de son exécution. Nous allons maintenant nous intéresser à la complexité des problèmes.
Un **problème** est spécifié par un quadruplet P = *(Entrées, Sorties, Préconditions, Postcondition)* tel que *Entrées* est la liste des paramètres en entrée, *Sorties* est la liste des paramètres en sortie, *Préconditions* est un ensemble (éventuellement vide) de conditions sur les paramètres en entrée, et *Postcondition* est une relation entre les valeurs des paramètres en entrée et en sortie.
Une **instance** d'un problème est une valuation des paramètres en entrée satisfaisant les préconditions.
Une **solution** d'une instance est une valuation des paramètres en sortie satisfaisant la postrelation pour la valuation des paramètres en entrée de l'instance.

Exemple. Le problème de la recherche d'un élément dans un tableau trié peut être spécifié de la façon suivante :
- Entrées : un tableau tab comportant n entiers indicés de 0 à $n-1$ et une valeur entière e
- Sortie : un entier i
- Précondition : les éléments de tab sont triés par ordre croissant
- Postrelation : si $\forall j \in [0, n-1], tab[j] \neq e$ alors $i = n$ sinon $i \in [0, n-1]$ et $tab[i] = e$

Ce problème admet une infinité d'instances. Par exemple :
- Instance 1 : $e = 8$ et $tab =$ | 4 | 4 | 7 | 8 | 10 | 11 | 12 |
- Instance 2 : $e = 9$ et $tab =$ | 4 | 4 | 7 | 8 | 10 | 11 | 12 |

Les solutions de ces deux instances sont $i = 3$ et $i = 7$, respectivement.
Un **algorithme** pour un problème P est une séquence d'instructions élémentaires permettant de calculer les valeurs des paramètres en sortie à partir des valeurs des paramètres en entrée, pour toute instance de P.

La notion d'instruction élémentaire dépend de la machine utilisée pour exécuter l'algorithme : pour être élémentaire, une instruction doit pouvoir être exécutée en temps constant par la machine.
La **complexité d'un problème** P est la complexité du meilleur algorithme pour P.
Chaque algorithme résolvant P fournit une borne supérieure de la complexité de P. Nous pouvons par ailleurs trouver des bornes inférieures en analysant le problème. Si la plus grande borne inférieure est égale à la plus petite borne supérieure, alors la complexité de P est connue ; sinon la complexité est ouverte.
Notons que la complexité d'un algorithme est généralement fonction de la taille des paramètres en entrée du problème. Quand il s'agit d'un problème dans un graphe, par exemple, la complexité pourra être fonction du nombre de sommets, du nombre d'arcs, ou encore du degré maximal des sommets.
Les travaux théoriques ont permis d'identifier différentes classes de problèmes en fonction de leur complexité. Il existe en fait un très grand nombre de classes différentes [1], et nous nous limiterons dans ce chapitre aux classes les plus importantes.
Ces classes de complexité ont été définies pour des problèmes particuliers, appelés **problèmes de décision**. Pour ces problèmes, la sortie et la postrelation sont remplacées par une question dont la réponse est binaire. Plus précisément, un **problème de décision** est spécifié par un triplet $P =$ *(Entrées, Préconditions, Question)* tel que *Entrées* est la liste des paramètres en entrée, *Préconditions* est un ensemble (éventuellement vide) de conditions sur les paramètres en entrée, et *Question* est une question portant sur les entrées dont la réponse est binaire (oui ou non).

Exemple. Le problème de décision associé à la recherche d'un élément dans un tableau trié peut être spécifié de la façon suivante :

- Entrées : un tableau tab comportant n entiers indicés de 0 à $n-1$ et une valeur entière e
- Précondition : les éléments de tab sont triés par ordre croissant
- Question : Existe-t-il un indice $j \in [0, n-1]$ tel que $tab[j] = e$?

▶ La classe $\mathcal{P}$

La classe $\mathcal{P}$ regroupe l'ensemble des problèmes de complexité polynomiale, c'est-à-dire, l'ensemble des problèmes pour lesquels il existe un algorithme dont la complexité est bornée par $\mathcal{O}(n^k)$ où n est la taille des paramètres en entrée et k est une constante indépendante de la taille des paramètres en entrée. Cette classe regroupe donc l'ensemble des problèmes qui peuvent être résolus en un temps "acceptable", même si cette notion d'acceptabilité du temps d'exécution dépend évidemment de l'application (et de la valeur de la constante k !).
Nous avons vu dans les chapitres précédents un certain nombre de problèmes appartenant à cette classe $\mathcal{P}$: problèmes de parcours, de recherche de plus courts chemins, et d'arbres couvrants minimaux. Il en existe évidemment beaucoup d'autres.

▶ La classe $\mathcal{NP}$

La classe $\mathcal{NP}$ regroupe l'ensemble des problèmes pour lesquels il existe un algorithme Polynomial pour une machine de Turing Non déterministe. Sans entrer dans les détails, une machine de Turing non déterministe peut être vue comme une machine capable d'exécuter en parallèle un nombre fini d'alternatives. Intuitivement, cela signifie que la résolution des problèmes de $\mathcal{NP}$ peut nécessiter l'examen d'un grand nombre (éventuellement exponentiel) de combinaisons, mais que l'examen de chaque combinaison peut être fait en temps polynomial. Autrement dit, un problème de décision appartient à la classe $\mathcal{NP}$ si pour toute instance dont la réponse est positive, il existe un certificat (souvent appelé "solution") permettant de vérifier en temps polynomial que la réponse est effectivement positive.
Ainsi, s'il peut être long de résoudre les instances d'un problème de $\mathcal{NP}$, il est très rapide de vérifier qu'une solution (c'est-à-dire, un certificat permettant de répondre "oui" à la question) est correcte.

1. Le « zoo des complexité », que l'on peut consulter sur `https://complexityzoo.uwaterloo.ca/Complexity_Zoo`, recense près de 500 classes de complexités différentes.

Considérons, par exemple, le problème de la satisfiabilité d'une formule booléenne (SAT) :

— Entrées : une formule booléenne F définie sur un ensemble X de n variables booléennes

— Question : Existe-t-il une valuation $V : X \to \{vrai, faux\}$ qui satisfait F

Ce problème appartient à la classe $\mathcal{NP}$ car il existe un algorithme polynomial pour vérifier si une valuation donnée $\in \{vrai, faux\}^n$ permet de répondre oui à la question : il suffit de décider si la valuation satisfait la formule F, ce qui peut être fait en temps linéaire par rapport à la taille de la formule.
Considérons par exemple l'instance suivante de SAT :

$$F = (a \vee \neg b) \wedge (\neg a \vee c \vee \neg d) \wedge (\neg b \vee \neg c \vee \neg e) \wedge (a \vee b \vee d \vee e)$$

définie sur l'ensemble de variables $X = \{a, b, c, d, e\}$. La réponse à la question pour cette instance est oui. Un certificat est

$$V = \{a = vrai, b = faux, c = vrai, d = vrai, e = faux\}$$

Nous pouvons vérifier que pour chaque clause de F, il existe au moins un littéral qui est satisfait par V.

▶ Problèmes $\mathcal{NP}$-complets

Les relations d'inclusion entre les classes $\mathcal{P}$ et $\mathcal{NP}$ sont à l'origine d'une très célèbre conjecture que l'on peut résumer par « $\mathcal{P} \neq \mathcal{NP}$ ». En effet, si la classe $\mathcal{P}$ est manifestement inclue dans la classe $\mathcal{NP}$, la relation inverse n'a jamais été ni montrée ni infirmée.
Cependant, certains problèmes de $\mathcal{NP}$ apparaissent plus difficiles à résoudre dans le sens où personne ne trouve d'algorithme polynomial pour les résoudre avec une machine déterministe. Les problèmes les plus difficiles de $\mathcal{NP}$ définissent la classe des problèmes $\mathcal{NP}$-complets ou, autrement dit, un problème de $\mathcal{NP}$ est $\mathcal{NP}$-complet s'il est au moins aussi difficile à résoudre que n'importe quel autre problème de $\mathcal{NP}$. Le premier problème qui a été montré $\mathcal{NP}$-complet est le problème SAT décrit précédemment. L'idée de la démonstration, proposée en 1971 par Cook, est de montrer que pour toute instance I d'un problème résolu par une machine de Turing non déterministe, il est possible de construire une formule booléenne F telle que F est satisfiable ssi la réponse à la question posée par I est positive.
Pour montrer qu'un nouveau problème X appartenant à la classe $\mathcal{NP}$ est $\mathcal{NP}$-complet, il faut montrer que tout algorithme permettant de résoudre X peut également être utilisé pour résoudre un autre problème $\mathcal{NP}$-complet X' (X' peut être le problème SAT ou un autre problème dont la $\mathcal{NP}$-complétude a été démontrée). Plus précisément, il s'agit de donner un algorithme permettant de transformer n'importe quelle instance I' de X' en une instance I de X telle que la réponse à la question posée par I soit égale à la réponse à la question posée par I'. De plus, la complexité de cet algorithme de transformation doit être polynomiale par rapport à la taille des paramètres en entrée de I'.
Cette équivalence par transformation polynomiale entre problèmes $\mathcal{NP}$-complets implique une propriété fort intéressante : si quelqu'un trouvait un jour un algorithme polynomial pour un de ces problèmes (n'importe lequel), il pourrait en déduire des algorithmes polynomiaux pour tous les autres problèmes, et il pourrait alors conclure que $\mathcal{P} = \mathcal{NP}$. La question de savoir si un tel algorithme existe a été posée en 1971 par Stephen Cook et n'a toujours pas reçu de réponse. La réponse à cette question fait l'objet d'un prix d'un million de dollars par l'institut de mathématiques Clay.

▶ Au delà de la classe $\mathcal{NP}$

Les **problèmes $\mathcal{NP}$-difficiles** sont les problèmes qui sont au moins aussi difficiles que tous les problèmes de la classe $\mathcal{NP}$, c'est-à-dire, les problèmes pour lesquels il existe une procédure de transformation polynomiale depuis un problème dont on sait qu'il est $\mathcal{NP}$-complet. Si la classe des problèmes $\mathcal{NP}$-difficiles est un sur-ensemble de la classe des problèmes $\mathcal{NP}$-complets, certains problèmes $\mathcal{NP}$-difficiles ne sont pas $\mathcal{NP}$-complets. En effet, pour qu'un problème $\mathcal{NP}$-difficile soit $\mathcal{NP}$-complet, il doit appartenir à la classe $\mathcal{NP}$, et donc il doit exister un algorithme polynomial permettant de vérifier qu'une combinaison candidate est effectivement une solution. Certains problèmes $\mathcal{NP}$-difficiles n'appartiennent pas à la classe $\mathcal{NP}$ et ne sont donc pas $\mathcal{NP}$-complets. C'est le cas, par exemple, du problème suivant :

— Entrées : un ensemble S de n entiers et 2 entiers k et l
— Question : Existe-t-il k sous-ensembles distincts de S dont la somme des éléments soit supérieure à l ?

En effet, la taille des données en entrée d'une instance de ce problème est en $\mathcal{O}(n)$, tandis que le nombre de sous-ensembles distincts de S est égal à 2^n de sorte que k peut être égal à 2^n. Dans ce cas, vérifier qu'une combinaison donnée est effectivement solution ne pourra pas toujours être fait en temps polynomial par rapport n.
Enfin, il existe des problèmes qui ne peuvent être résolus par une machine de Turing (déterministe ou non). Ces problèmes sont dits **indécidables**. Comme une machine de Turing a le même pouvoir d'expression qu'un ordinateur, ces problèmes indécidables ne peuvent être résolus par un ordinateur ou, autrement dit, il n'existe pas d'algorithme pour les résoudre. Le premier problème qui a été montré indécidable est le problème de l'arrêt de la machine de Turing qui consiste à décider si l'exécution d'un programme donné se termine en un temps fini : s'il existe certains programmes pour lesquels il est facile de déterminer s'ils terminent ou non, il en existe d'autres pour lesquels la seule façon de décider de leur terminaison est de les exécuter... mais malheureusement, si l'exécution du programme ne se termine pas, on n'aura pas la réponse au problème de l'arrêt en un temps fini !

▶ Problèmes d'optimisation

Le but d'un problème d'optimisation est de trouver une solution maximisant (resp. minimisant) une fonction objectif donnée. À chaque problème d'optimisation on peut associer un problème de décision dont le but est de déterminer s'il existe une solution pour laquelle la fonction objectif soit supérieure (resp. inférieure) ou égale à une valeur donnée. La complexité d'un problème d'optimisation est liée à celle du problème de décision qui lui est associé. En particulier, si le problème de décision est $\mathcal{NP}$-complet, alors le problème d'optimisation est dit $\mathcal{NP}$-difficile.

7.2 Recherche de cliques

Etant donné un graphe non orienté $G = (S, A)$, une clique est un sous-ensemble de sommets $S' \subseteq S$ qui sont tous connectés 2 à 2 par des arêtes de sorte que

$$\forall (i, j) \in S' \times S', i \neq j \Rightarrow \{i, j\} \in A$$

Autrement dit, une clique est un sous-graphe complet.
Le problème de la clique est spécifié de la façon suivante :

— Entrées : un graphe non orienté $G = (S, A)$ et un entier positif k
— Question : G contient-il une clique de k sommets ?

▶ Complexité du problème de la clique

Théorème : Le problème de la clique est $\mathcal{NP}$-complet.

Démonstration : Le problème de la clique appartient à la classe $\mathcal{NP}$ car nous pouvons vérifier en temps polynomial qu'un sous-ensemble $S' \subseteq S$ de k sommets est bien une clique. Il suffit pour cela de vérifier que pour toute paire de sommets $\{s_i, s_j\} \subseteq S'$, il existe une arête (s_i, s_j) dans A.
Pour montrer que le problème de la clique est $\mathcal{NP}$-complet, nous allons décrire ici une procédure permettant de transformer une instance du problème SAT en une instance du problème de la clique telle que les deux instances admettent la même réponse. Soit F une formule booléenne sous forme normale conjonctive, de sorte que F est une conjonction de clauses, chaque clause étant une disjonction de littéraux, et chaque littéral étant soit une variable booléenne, soit la négation d'une variable booléenne. Nous construisons le graphe non orienté $G = (S, A)$ tel que :

— S associe un sommet à chaque littéral de chaque clause de F ; on note clause(u) et littéral(u) la clause et le littéral correspondant au sommet u ;
— A associe une arête à chaque couple de sommets (u, v) tel que

1. u et v correspondent à deux clauses différentes, c'est-à-dire, clause$(u) \neq$ clause(v) ;
2. les littéraux associés à u et v sont compatibles, c'est-à-dire, littéral$(u) \neq \neg$littéral(v).

Par exemple, la figure 7.1 décrit le graphe construit à partir d'une formule sous forme normale conjonctive.

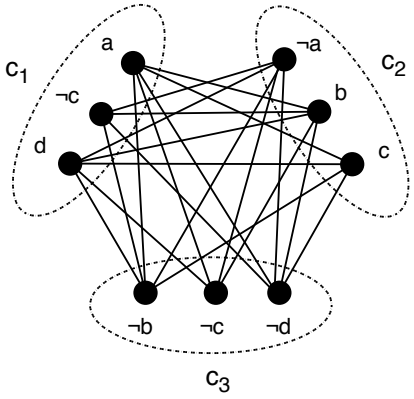


FIGURE 7.1 – Graphe généré à partir de la formule booléenne $F = (a \vee \neg c \vee d) \wedge (\neg a \vee b \vee c) \wedge (\neg b \vee \neg c \vee \neg d)$.

Montrons maintenant qu'il existe une valuation des variables satisfaisant F si et seulement si le graphe admet une clique de k sommets, où k est égal au nombre de clauses de F :

— S'il existe une valuation des variables satisfaisant les k clauses, cela implique qu'au moins un littéral de chaque clause est vrai. L'ensemble formé des sommets associés à l'un de ces littéraux par clause est une clique du graphe.

— Si le graphe a une clique d'ordre k, elle contient exactement un sommet parmi ceux représentant les littéraux de chaque clause (puisque deux littéraux dans une même clause ne peuvent être reliés par une arête). On peut définir une valuation des variables à partir des littéraux appartenant à la clique : pour chaque variable x, si la clique contient le littéral x alors x est valuée à vrai ; si la clique contient le littéral $\neg x$ alors x est valuée à faux ; si la clique ne contient ni x ni $\neg x$ alors x est valuée à vrai ou faux, arbitrairement (puisque x n'est pas utilisée dans la clique pour satisfaire une clause) ; la clique ne peut pas contenir à la fois x et $\neg x$ puisqu'il n'y a pas d'arête entre x et $\neg x$. Cette valuation satisfait bien chaque clause et est donc solution.

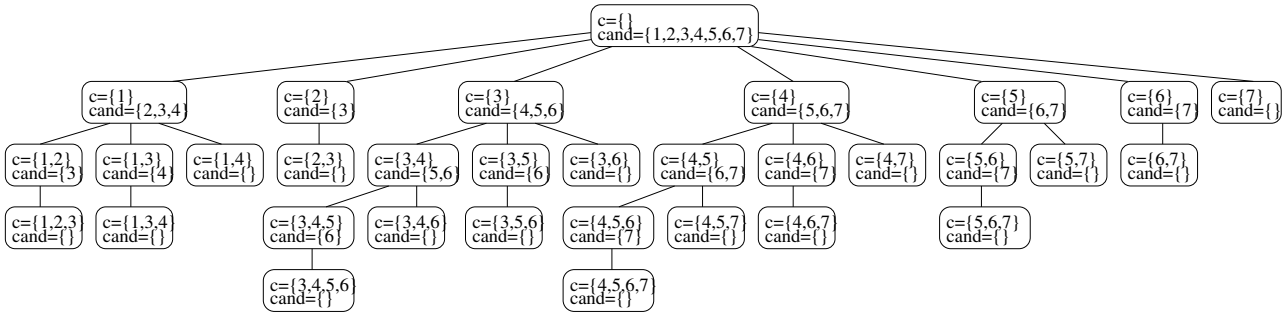
▶ Enumération de toutes les cliques d'un graphe

Les algorithmes d'énumération basés sur le principe du "retour-arrière" (*backtracking*) sont généralement très facilement décrits récursivement en se basant sur l'idée suivante : étant donnée une solution partielle, essayer récursivement toutes les extensions possibles permettant de compléter la solution partielle en une solution complète.
L'algorithme 18 applique ce principe pour rechercher exhaustivement toutes les cliques dans un graphe. Au premier appel, la clique c passée en paramètre doit être vide.
On dit généralement que la fonction enumClique construit un **arbre de recherche**. Il s'agit de l'arbre dont chaque nœud contient une clique (la racine contenant la clique vide) et dont les arcs correspondent aux enchaînements d'appels récursifs : le nœud contenant une clique c est le père d'un nœud contenant une clique c' si enumClique(g, c) a appelé enumClique(g, c'). À chaque nœud de l'arbre de recherche, l'algorithme calcule l'ensemble $cand$ de tous les sommets qui peuvent être ajoutés à la clique courante c puis il crée autant de fils qu'il y a de sommets dans $cand$.
L'algorithme construit l'arbre de recherche en profondeur d'abord, selon un principe de retour-arrière chronologique. Lorsque la clique courante c ne peut être augmentée, la fonction continue l'exploration en remettant en cause le dernier sommet ajouté. Ce principe est très simple à mettre en œuvre par une procédure récursive. Pour éviter d'énumérer plusieurs fois une même clique, l'ensemble $cand$ est limité aux sommets plus grands que les sommets de la clique

Algorithme 18 : Enumération des cliques d'un graphe

1 **Fonction** *enumClique*(g, c)
  Entrée : Un graphe $g = (S, A)$ et un ensemble de sommets $c \subseteq S$
  Précondition : c est une clique de g
  Postcondition : Affiche l'ensemble des cliques de g qui sont des sur-ensembles de c
2 afficher c
3 $cand \leftarrow \{s_i \in S \mid \forall s_j \in c, (s_i, s_j) \in A \wedge s_i > s_j\}$
3 **pour** *chaque sommet* $s_i \in cand$ **faire**
4 enumClique$(g, c \cup \{s_i\})$

(en considérant un ordre arbitraire donné sur les sommets). Pour une implémentation plus efficace, l'ensemble $cand$ est généralement maintenu incrémentalement, au lieu d'être recalculé à partir de rien à chaque appel de la fonction enumClique : initialement $cand$ contient tous les sommets ; à chaque fois qu'un sommet s est ajouté à la clique courante c, avant de rappeler récursivement enumClique, tous les sommets qui ne sont pas adjacents à s sont enlevés de $cand$.
Par exemple, l'arbre de recherche de la procédure enumClique pour le graphe de la figure 7.2 est :



La complexité de l'algorithme 18 peut être évaluée par rapport au nombre d'appels récursifs à la fonction enumClique. Montrons tout d'abord qu'à chaque appel récursif à enumClique, le paramètre c en entrée de l'algorithme est une clique :

— au premier appel, c est vide et est bien une clique ;
— si à l'appel de enumClique c est une clique, alors pour chaque appel récursif à enumClique de la ligne 3 de l'algorithme 18, le deuxième paramètre $c \cup \{s_i\}$ est une clique puisque $s_i \in cand$ et $cand$ ne contient que des sommets connectés à tous les sommets de c.

Montrons maintenant que si c' est une clique de g, alors il y aura exactement un appel à enumClique pour lequel le paramètre c en entrée sera égal à c'. Soit $n = |c'|$. Notons $s_1, s_2, \ldots s_n$ les sommets de c' triés par ordre croissant. Pour tout i compris entre 1 et $n-1$, l'appel à enumClique avec $c = \{s_1, \ldots, s_i\}$ engendrera un appel à enumClique avec $c = \{s_1, \ldots, s_i, s_{i+1}\}$ puisque pour tout sommet $s_j \in \{s_1, \ldots, s_i\}$, s_{i+1} est connecté à s_j et $s_{i+1} > s_j$. Par ailleurs, il ne peut y avoir plusieurs appels différents à enumClique avec une même clique c passée en paramètre du

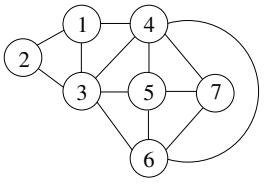


FIGURE 7.2 – Exemple de graphe

fait que l'ensemble $cand$ est limité aux sommets supérieurs à tous les sommets de c.
Ainsi, la fonction enumClique est appelée exactement une fois pour chaque clique de g. À chaque appel, il faut essentiellement construire l'ensemble $cand$ (ou le maintenir incrémentalement), ce qui peut être fait en temps linéaire par rapport au nombre de sommets. Si le graphe a n sommets, la complexité est $\mathcal{O}(nx)$, où x est le nombre de cliques de g. Il s'agit d'une complexité un peu particulière, qui dépend de la valeur retournée par l'algorithme. Nous dirons que l'algorithme a une complexité en temps polynomial incrémental (*incremental polynomial time*) car la complexité en temps entre deux calculs de cliques est polynomiale. Évidemment, comme le nombre de cliques d'un graphe peut être exponentiel, la complexité est exponentielle dans le pire des cas, et les temps d'exécution de l'algorithme 18 peuvent vite devenir rédhibitoires.

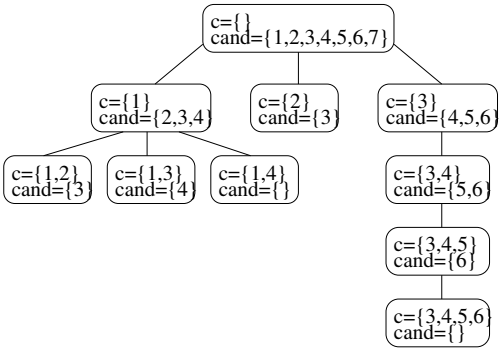
▶ Recherche d'une clique de taille donnée

L'algorithme 19 recherche une clique d'une taille k donnée. Comme pour l'algorithme 18, à chaque appel récursif, l'algorithme calcule l'ensemble $cand$ de tous les sommets qui peuvent être ajoutés à la clique courante c puis il crée autant de fils qu'il y a de sommets dans $cand$. Pour éviter de développer des branches de l'arbre ne pouvant contenir de nœuds correspondant à des cliques de taille k, les nœuds pour lesquels le nombre de candidats ajouté au nombre de sommets dans la clique courante c est inférieur à k ne sont pas développés. Cette étape d'évaluation est appelée *Bound*, et l'algorithme 19 est une instanciation du principe de recherche appelé *Branch & Bound*.

Algorithme 19 : Recherche d'une clique de k sommets

```
1 Fonction chercheClique(g, c, k)
      Entrée        : Un graphe g = (S, A), un ensemble de sommets c ⊆ S et un entier positif k
      Précondition  : c est une clique de taille inférieure ou égale à k
      Postcondition : Retourne vrai s'il existe une clique c' de g telle que |c'| = k et c ⊆ c' ; faux sinon
2     si |c| = k alors retourne vrai;
3     cand ← {s_i ∈ S | ∀s_j ∈ c, (s_i, s_j) ∈ A ∧ s_i > s_j}
4     si |c| + |cand| < k alors retourne faux;
5     pour chaque sommet s ∈ cand faire
6         si chercheClique(g, c ∪ {s}, k) alors retourne vrai;
7     retourne faux
```

Par exemple, l'arbre de recherche de la fonction chercheClique pour le graphe de la figure 7.2 pour $k = 4$ est :



Il est possible d'améliorer l'étape d'évaluation en raisonnant plus finement. Par exemple, nous pouvons ne garder dans l'ensemble $cand$ que les sommets connectés à au moins $k - |c|$ sommets de $cand$.
Considérons par exemple le nœud à la racine de l'arbre de recherche, où $c = \emptyset$ et $cand = \{1, 2, 3, 4, 5, 6, 7\}$. Les sommets 1 et 2 ont des degrés inférieurs à 4 et peuvent être éliminés de $cand$. Une fois que 1 et 2 ont été enlevés de $cand$, le sommet 3 n'est plus connecté qu'à 3 sommets de $cand$ (4, 5 et 6), de sorte que 3 peut être également supprimé de $cand$. Sur cet exemple, cette étape de raisonnement nous permet de supprimer tous les nœuds inutiles

de l'arbre de recherche. Malheureusement, il peut arriver, dans d'autres cas, que ce raisonnement ne permette pas de réduire la taille de l'arbre de recherche. Notons par ailleurs que cette opération de filtrage ne peut être faite en une seule passe : à chaque fois qu'un sommet est enlevé dans l'ensemble $cand$, il est nécessaire de vérifier que les sommets de $cand$ qui sont voisins du sommet enlevé sont encore connectés à au moins $k - |c|$ sommets de $cand$.

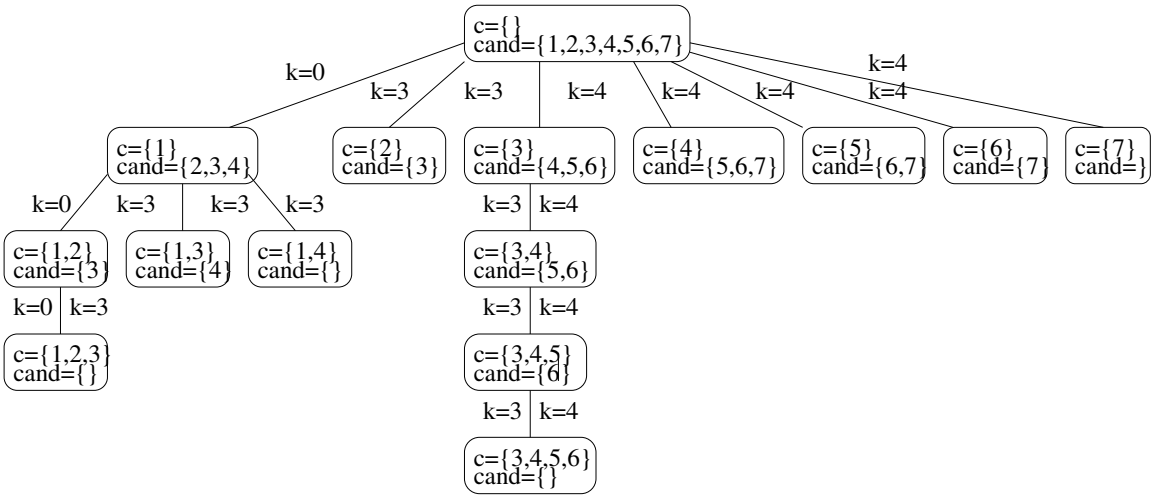
▶ Recherche d'une clique maximum

L'algorithme 19 peut être facilement adapté pour rechercher une clique de taille maximum dans un graphe. L'idée est de maintenir la taille k de la plus grande clique trouvée depuis le début de la recherche, et de chercher une clique de taille strictement supérieure à k, comme décrit dans l'algorithme 20.

Algorithme 20 : Recherche d'une clique maximum

1 **Fonction** *chercheCliqueMax*(g, c, k)
  Entrée : Un graphe $g = (S, A)$, un ensemble de sommets $c \subseteq S$ et un entier positif k
  Précondition : c est une clique
  Postcondition : retourne max(k, k') où k' est la taille de la plus grande clique de g contenant c
2 $cand \leftarrow \{s_i \in S \mid \forall s_j \in c, (s_i, s_j) \in A \wedge s_i > s_j\}$
3 **si** $cand = \emptyset$ **alors retourne** *max*$(|c|, k)$;
4 **pour** *chaque sommet* $s \in cand$ **et tant que** $|c| + |cand| > k$ **faire**
5 $k \leftarrow$chercheCliqueMax$(g, c \cup \{s\}, k)$
6 **retourne** k

Par exemple, l'arbre de recherche de la fonction chercheCliqueMax pour le graphe de la figure 7.2 est :



▶ Heuristiques d'ordre

Pour tenter de trouver plus rapidement de grandes cliques, il est possible d'utiliser une heuristique d'ordre au moment de parcourir les sommets de l'ensemble $cand$. Il s'agit d'une heuristique dans le sens où la complexité théorique de l'algorithme n'est pas changée et reste exponentielle, même si la résolution est accélérée pour de très nombreuses instances. L'idée est de définir le poids d'un sommet $s \in cand$ par son degré dans le sous-graphe induit par $cand$, et d'ordonner les sommets par ordre de poids décroissant.

Considérons par exemple le graphe de la figure 7.2. Initialement, $cand$ contient tous les sommets de sorte qu'au premier appel de chercheCliqueMax les sommets sont choisis par ordre de degré décroissant. Le premier sommet choisi sera donc 3 ou 4. Supposons que ce soit 3 qui soit choisi en premier. L'ensemble $cand$ est réduit à $\{1, 2, 4, 5, 6\}$. Dans le sous-graphe induit par $cand$, le sommet ayant le plus fort degré est 4 de sorte que le prochain sommet à entrer

dans la clique courante est 4. L'ensemble $cand$ est réduit à $\{5, 6\}$. Ces deux sommets ont le même degré dans le sous-graphe induit par $cand$, et appartiennent tous les deux à une clique de taille 4. Ainsi, l'heuristique d'ordre permet de trouver la clique maximum en 5 appels récursifs, au lieu de 11 appels sans heuristique. En revanche, il faudra encore quelques appels récursifs pour prouver que cette clique est effectivement la plus grande
Sur cet exemple, l'heuristique d'ordre permet donc de trouver tout de suite la plus grande clique du graphe. Cependant, il est très facile de construire des exemples pour lesquels l'heuristique ne marchera pas aussi bien. Considérons par exemple le graphe de la figure 7.2, et supposons que nous ajoutons 4 nouveaux sommets au graphe, et que nous connectons chacun de ces nouveaux sommets au sommet 2. Dans ce cas, le sommet 2 devient le sommet ayant le plus grand degré et il sera donc choisi en premier par l'heuristique alors qu'il n'appartient pas à la plus grande clique du graphe.

▶ Algorithme glouton

Les algorithmes gloutons peuvent être utilisés pour construire des solutions sous-optimales à des problèmes d'optimisation $\mathcal{NP}$-difficiles. L'idée est de construire une solution itérativement, en partant d'une solution vide et en ajoutant à chaque itération un composant de solution. Le composant ajouté à chaque itération est choisi selon une heuristique gloutonne, c'est-à-dire qu'on choisit le composant maximisant (ou minimisant) une fonction heuristique donnée. Les algorithmes gloutons permettent généralement de construire rapidement (en temps polynomial) des solutions de "bonne qualité", mais sans garantie sur la qualité de la solution (sauf dans quelques cas bien identifiés comme, par exemple, les algorithmes de Dijkstra, Kruskal ou Prim, où la solution calculée de façon gloutonne est effectivement optimale).
L'algorithme 21 décrit un algorithme glouton permettant de calculer rapidement de grandes cliques, sans aucune garantie sur la taille de la clique. L'heuristique gloutonne utilisée consiste à choisir à chaque itération le sommet ayant le plus grand nombre de voisins dans l'ensemble $cand$ de tous les sommets pouvant être ajoutés à la clique c en cours de construction.

Algorithme 21 : Recherche gloutonne d'une clique

```
1 Fonction chercheCliqueGlouton(g)
      Entrée       : Un graphe g = (S, A)
      Postcondition : retourne une clique de g
2     cand ← S
3     c ← ∅
4     tant que cand ≠ ∅ faire
5         Soit s_i le sommet de cand maximisant |cand ∩ adj(s_i)|
6         c ← c ∪ {s_i}
7         cand ← cand ∩ adj(s_i)
8     retourne c
```

7.3 Coloriage de graphes

Le coloriage d'un graphe non orienté G consiste à attribuer une couleur à chaque sommet de G, de telle sorte qu'une même couleur ne soit pas attribuée à deux sommets adjacents (reliés par une arête). Plus précisément, étant donné un ensemble C de couleurs, le **coloriage** d'un graphe non orienté $G = (S, A)$ est une fonction $c : S \to C$ telle que $\forall (s_i, s_j) \in A, c(s_i) \neq c(s_j)$. Le **nombre chromatique** de G, noté $\chi(G)$, est la cardinalité du plus petit ensemble de couleurs C permettant de colorier G.
Le problème du k coloriage consiste à déterminer s'il est possible de colorier un graphe avec au plus k couleurs ou, autrement dit, si $\chi(g) \leq k$. Ce problème est spécifié de la façon suivante :

— Entrées : un graphe non orienté $G = (S, A)$ et un entier positif k
— Question : G peut-il être colorié avec k couleurs ?

Ce problème est $\mathcal{NP}$-complet, mais nous ne détaillerons pas la preuve ici.

Relation entre coloriage et cliques : Pour tout graphe G, $\chi(G)$ est supérieur ou égal au nombre de sommets de la clique maximum de G.
En effet, tous les sommets d'une clique sont connectés deux à deux de sorte qu'ils doivent tous avoir des couleurs différentes. Par conséquent, pour toute clique c, $\chi(g)$ est supérieur ou égal au nombre de sommets de c.

Théorème des 4 couleurs : Un graphe planaire est un graphe qui peut être dessiné sur un plan de telle sorte qu'aucune arête ne croise une autre arête (en dehors des extrémités des arêtes). Francis Guthrie a conjecturé en 1852 que le nombre chromatique d'un graphe planaire est inférieur ou égal à 4. Cette conjecture a passionné de nombreux mathématiciens pendant plus d'un siècle, et elle n'a été démontrée qu'en 1976.
Une conséquence immédiate de cette propriété est qu'il est toujours possible de colorier les pays d'une carte géographique avec seulement 4 couleurs de telle sorte que deux pays voisins soient coloriés avec des couleurs différentes.

▶ Algorithme de Brélaz

L'algorithme de Brélaz (également appelé DSATUR), décrit dans l'algorithme 22, est un algorithme glouton qui permet de calculer une borne supérieure $borne\chi$ de $\chi(G)$.

Algorithme 22 : Coloriage glouton d'un graphe

1 **Fonction** *brélaz*(g)
  Entrée : Un graphe $g = (S, A)$
  Postcondition : retourne une borne supérieure de $\chi(g)$
2 $borne\chi \leftarrow 0$
3 $N \leftarrow S$
4 **pour** *chaque sommet* $s_i \in S$ **faire**
5 $couleursVoisines[s_i] \leftarrow \emptyset$
6 $nbVoisinsNonColories[s_i] \leftarrow |adj(s_i)|$
7 **tant que** $N \neq \emptyset$ **faire**
8 Soit X l'ensemble des sommets $s_j \in N$ tels que $|couleursVoisines[s_i]|$ soit maximal
9 Choisir un sommet s_i de X tel que $nbVoisinsNonColories[s_i]$ soit maximal
10 Enlever s_i de N
11 **si** $|couleursVoisines[s_i]| = borne\chi$ **alors** $borne\chi \leftarrow borne\chi + 1$;
12 Soit k la plus petite valeur comprise entre 1 et $borne\chi$ telle que $k \notin couleursVoisines[s_i]$
13 $couleur[s_i] \leftarrow k$
14 **pour** *chaque sommet* $s_j \in adj(s_i) \cap N$ **faire**
15 $nbVoisinsNonColories[s_j] \leftarrow nbVoisinsNonColories[s_j] - 1$
16 $couleursVoisines[s_j] \leftarrow couleursVoisines[s_j] \cup \{couleur[s_i]\}$
17 **retourne** $borne\chi$

L'algorithme maintient un ensemble N de sommets qui n'ont pas été coloriés et, à chaque itération, sélectionne un sommet s_i de N et le colorie avec la plus petite couleur telle qu'aucun voisin de s_i ne soit déjà colorié avec cette couleur (les couleurs sont numérotées de 1 à $borne\chi$) ; si toutes les couleurs existantes sont utilisées par au moins un voisin de s_i, alors $borne\chi$ est incrémenté.
Une heuristique est utilisée pour choisir, à chaque itération, le prochain sommet s_i à être colorié : l'algorithme construit l'ensemble X des sommets ayant le plus de voisins coloriés avec des couleurs différentes (sommets les plus contraints,

ligne 8), puis sélectionne le sommet de X ayant le plus de voisins qui n'ont pas encore été coloriés (sommet le plus contraignant, ligne 9).

Complexité de l'algorithme de Brélaz. Soit $n = |S|$ et $p = |A|$. L'algorithme passe n fois dans la boucle lignes 7 à 16. À chaque passage dans la boucle, l'algorithme parcourt la liste des voisins du sommet colorié s_i. La complexité de l'algorithme est $\mathcal{O}(n + p)$ (en utilisant un tableau de booléens permettant de déterminer en temps constant si un sommet voisin d'un sommet donné utilise déjà une couleur donnée).

7.4 Le voyageur de commerce

Un **cycle hamiltonien** d'un graphe non orienté est un cycle passant par chacun de ses sommets exactement une fois. Le problème de décision consistant à déterminer si un graphe admet un cycle hamiltonien est $\mathcal{NP}$-complet, mais nous ne détaillerons pas la preuve ici.
Considérons maintenant un graphe non orienté $G = (S, A)$ muni d'une fonction $c : A \to \mathbb{R}^+$ associant un coût à chacune de ses arêtes, et définissons le coût d'un cycle par la somme des coûts de ses arêtes. Le problème du voyageur de commerce consiste à rechercher dans G un cycle hamiltonien de coût minimal. Quand le graphe est orienté, de sorte que le coût de l'arc (i, j) peut être différent du coût de l'arc (j, i), le problème est appelé : voyageur de commerce asymétrique.

Théorème : Le problème de décision associé au voyageur de commerce, visant à déterminer si un graphe donné contient un cycle hamiltonien de coût inférieur ou égal à une borne k donnée, est $\mathcal{NP}$-complet.

Démonstration. Le problème appartient à la classe $\mathcal{NP}$ car nous pouvons vérifier en temps polynomial qu'un cycle donné est hamiltonien et est de coût inférieur ou égal à k.
Pour montrer qu'il est $\mathcal{NP}$-complet, nous allons décrire une procédure permettant de transformer une instance du problème de recherche de cycle hamiltonien dans un graphe G en une instance du problème du voyageur de commerce. Nous définissons pour cela la fonction coût telle que toutes les arêtes de G ont un même coût égal à 1. Nous pouvons facilement vérifier qu'il existe une solution au problème du voyageur de commerce pour $k = |S|$ si et seulement si G admet un cycle hamiltonien.
Dans la suite, nous allons supposer que le graphe est complet. Si ce n'est pas le cas, il est toujours possible d'ajouter de nouvelles arêtes au graphe, jusqu'à ce qu'il devienne complet, et d'associer à chaque nouvelle arête un coût supérieur à la borne k.

▶ Enumération de toutes les permutations d'un ensemble

Une façon naïve de résoudre le problème du voyageur de commerce consiste à énumérer tous les cycles hamiltoniens du graphe pour rechercher celui ayant le plus petit coût. Dans le cas où le graphe est complet, nous devons pour cela énumérer toutes les permutations de l'ensemble S des sommets, comme décrit dans l'algorithme 23. Comme pour l'algorithme énumérant toutes les cliques d'un graphe, l'algorithme est récursif et construit implicitement un arbre de recherche énumérant toutes les permutations de l'ensemble S des sommets. Au premier appel, $cand$ contient l'ensemble S des sommets et la liste deb est vide.

▶ Résolution du voyageur de commerce par séparation et évaluation

L'algorithme 23 peut être facilement adapté pour résoudre le problème du voyageur de commerce selon un principe d'exploration de l'espace de recherche par séparation et évaluation (*branch and bound*). Pour cela, il faut maintenir le coût k du plus court chemin hamiltonien trouvé depuis le début de la recherche : avant le premier appel, k est initialisé

Algorithme 23 : Enumération de toutes les permutations d'un ensemble

1 **Procédure** *enum-cycles*$(cand, deb)$
  Entrée : Un ensemble de sommets $cand$, une liste ordonnée de sommets deb
  Postcondition : Affiche toutes les listes commençant par deb et se terminant par une permutation de $cand$
2 **si** $cand = \emptyset$ **alors** Afficher deb;
3 **pour** *chaque sommet* $s_j \in cand$ **faire**
4 enum-cycles$(cand \setminus \{s_j\}, deb. < s_j >)$

à ∞ ; à chaque fois qu'un chemin hamiltonien complet, a été construit (ligne 2), ce coût est mis à jour si la somme des coûts des arcs du chemin est inférieure à k.

Afin de limiter le nombre d'appels récursifs, une fonction d'évaluation (appelée *bound*) est utilisée avant chaque appel ligne 3. Cette fonction *bound* calcule une borne inférieure du coût du plus court chemin allant du dernier sommet de deb jusqu'au premier sommet de deb, et passant par chaque sommet de $cand$ exactement une fois. Si le coût des arcs de deb ajouté à cette borne est supérieur à k, alors nous pouvons en déduire qu'il n'existe pas de solution commençant par deb, et il n'est pas nécessaire d'appeler *enum-cycles* récursivement (nous disons dans ce cas que la branche est coupée). Quelques exemples de définitions possibles pour la fonction *bound* sont données ci-dessous.

- Retourner $(|cand| + 1) * minCost$, où $minCost$ est le plus petit coût associé à une arête du graphe. Cette définition a l'avantage d'être calculable en temps constant à chaque appel récursif (en supposant que $minCost$ est évalué une fois pour toute au début de la recherche).
- Retourner $\sum_{s_j \in cand} \min\{c(s_j, s_l), s_l \in \{s_1\} \cup cand\}$, où s_1 est le premier sommet de deb.
- Retourner le coût d'un MST du graphe induit par $cand \cup \{s_1, s_i\}$, où s_1 est le premier sommet de deb et s_i le dernier sommet de deb

Evidemment, plus la fonction *bound* retourne une valeur proche du coût réel du plus court chemin élémentaire reliant le dernier sommet de deb à son premier sommet en passant par chacun des sommets de $cand$, et moins l'algorithme fera d'appels récursifs. Cependant, calculer une meilleure borne est généralement plus coûteux en temps. Ainsi, il s'agit de trouver un compromis entre une fonction *bound* plus précise, mais plus coûteuse, et une fonction *bound* moins précise, mais moins coûteuse.