Introduction à la théorie des graphes

Simon de Givry et Thomas Schiex

Table des matières

1 Introduction, notations, définitions **7**
- 1.1 Graphes orientés et non orientés . . . 7
- 1.2 Chemins, circuits, chaînes et cycles . . . 9
- 1.3 Forte connexité, connexité . . . 11
- 1.4 Arbres, arborescences . . . 12

2 Représentation machine **14**
- 2.1 Listes d'adjacence . . . 14
- 2.2 Matrice d'adjacence . . . 15

3 Parcours des graphes **17**
- 3.1 Parcours des arborescences en profondeur . . . 17
- 3.2 Parcours des graphes en profondeur d'abord . . . 18
  - 3.2.1 Application, le tri topologique . . . 19
- 3.3 Parcours des graphes en largeur d'abord . . . 20

4 Arbres couvrants de poids minimum **23**

5 Plus courts chemin, plus long chemin et ordonnancement **27**
  - 5.0.1 Propriétés des plus courts chemins . . . 30
- 5.1 Algorithme de Bellman-Ford-Moore . . . 31
- 5.2 L'algorithme de Dijkstra . . . 32
- 5.3 Cas des graphes orientés sans circuits . . . 34
- 5.4 Problèmes de plus long chemin et ordonnancement . . . 35
- 5.5 Distance minimum entre toutes les paires . . . 37

6 Autres problèmes bien résolus **38**
- 6.1 Couplages . . . 38
- 6.2 Graphe de transport et flot . . . 41
  - 6.2.1 Coupe d'un réseau . . . 42
  - 6.2.2 Flot maximum . . . 42
  - 6.2.3 Améliorations . . . 46
  - 6.2.4 Théorème de Ford-Fulkerson . . . 47
  - 6.2.5 Fermeture de bénéfice maximal . . . 47
- 6.3 Flot de coût minimum . . . 49
  - 6.3.1 Graphe des capacités résiduelles . . . 49
  - 6.3.2 Algorithme de Busacker et Gowen . . . 50

6.3.3 Chargement de bateaux 53
6.4 Problèmes NP-complets sur les graphes 53

Ce cours s'appuie sur un grand nombre de ressources et avant tout sur l'excellent livre "Introduction à l'algorithmique" de Cormen, Leiserson, Rivest et Stein. Pour l'essentiel, nous utiliserons les notations de cet ouvrage. N'hésitez pas à vous le procurer.

Nous sommes aussi inspirés des livres "Network Flows" de Ahuja, Magnati et Orlin et de "Digraphs, Theory, Algorithms and Applications" de Bang-Jensen et Gutin. Enfin, certaines preuves sont extraites du cours de Master de R. Giroudeau et H. Dicky du LIRMM (Montpellier). Qu'ils soient tous remerciés ici et ne soient pas tenus pour responsable des erreurs que nous avons pu y introduire.

Il est souvent naturel d'abstraire un problème ou un objet en traçant sur une feuille des points, représentant des objets élémentaires du problème, et en les reliant par des traits ou des flèches mettant en valeur une relation entre points. La théorie des graphes s'intéresse à ces représentations, les graphes, et aux traitements que l'on peut leur appliquer pour résoudre des problèmes sur ces graphes. Le but de ce cours est de vous faire connaître les notions et algorithmes fondamentaux développés en théorie des graphes et d'illustrer les types de problèmes qu'ils permettent de résoudre. On se limitera ici à la présentation d'algorithmes efficaces destinés à résoudre des problèmes polynomiaux.

La première utilisation des graphes est de modéliser une situation donnée afin de mieux la comprendre, l'analyser. On pourra aussi ainsi profiter du large corpus de connaissances et d'algorithmes disponibles pour le traitement des graphes.

Un exemple classique de problème sur les graphes est le problème de la ville de Koenigsberg (plus récemment Kaliningrad), traité par Euler (1736). Cette ville touristique est traversée par la Pregel qui coule de part et d'autre de l'île de Kneiphof. Son plan schématique est indiqué en figure 1. Elle disposait de sept ponts et un circuit de visite permettant de parcourir ces 7 ponts, sans jamais passer plusieurs fois par le même pont, était très désirable.

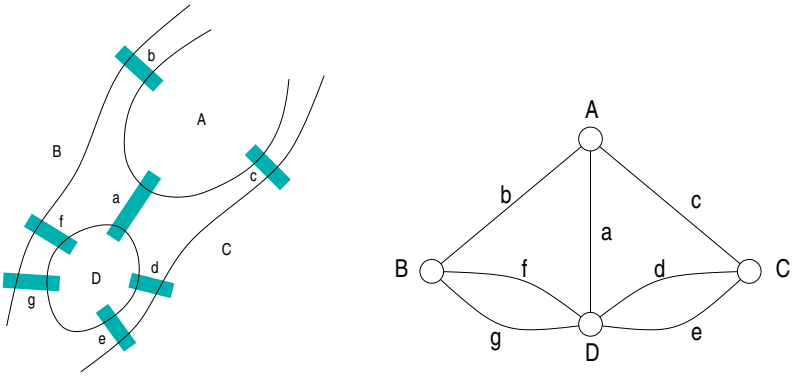

FIGURE 1 – Un graphe orienté, G_1Un classique : les ponts de Koenigsberg

Ce problème peut simplement se modéliser sous la forme de graphe. On associe à chaque région géographique un point (ou sommet) et chaque pont est représenté par un trait (une arête) qui relie deux régions, donc deux sommets. Le graphe obtenu est visible dans la figure 1, à droite. Le problème est alors d'identifier un chemin dans le graphe qui passe une fois et une seule par chaque arête. Il s'agit d'un problème maintenant résolu, celui de la recherche d'un circuit eulérien et dont on sait qu'il n'existe que si chaque sommet est connecté à un nombre pair d'autres sommets. Ce n'est pas le cas ici et donc on sait qu'il n'existe pas de solution au problème de la ville de Koenigsberg.

Un problème plus proche de nous est présenté dans [AHU87]. On considère le carrefour routier de la figure 2 (toute ressemblance avec le carrefour du bar des avions est fortuite, il s'agirait d'un carrefour situé près de l'université de

Princeton). Les voies C et E sont à sens unique, les autres sont à double sens. Suite à un nombre important d'accidents, on souhaite installer des feux de circulation à ce carrefour. Ces feux de circulations suivent des phases successives, chaque phase autorisant certains changement de voies. Il y a 13 changements de voie possibles à ce carrefour. Par exemple, AB consiste à passer de la voie A à la voie B. L'utilisation simultanée de AB et EC est possible alors que celle de AD et EB ne l'est pas car les flots de communication se croisent. Il faut bien sûr absolument éviter que deux changements de voies incompatibles apparaissent dans la même phase.

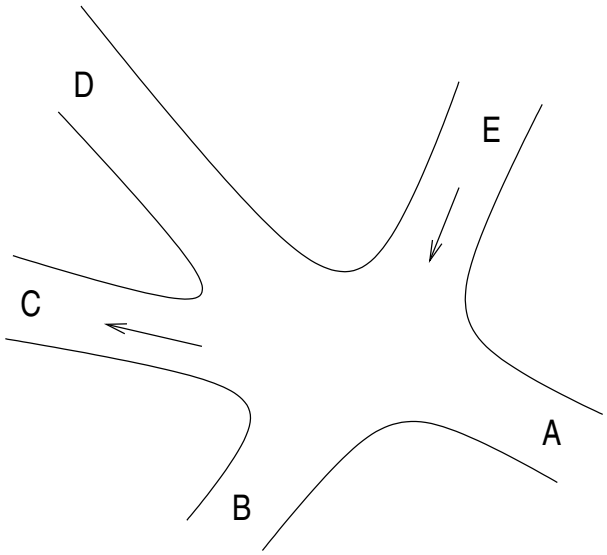

FIGURE 2 – Un carrefour à Princeton

Afin de maximiser l'efficacité des feux, on cherche à trouver une série de phases qui soit la plus courte possible et qui couvre chaque changement de voies une fois. On peut modéliser ce problème sous forme de graphe : chaque changement de voie possible est représenté par un sommet et une arête relie deux sommets "incompatibles". La coloration d'un sommet dans une couleur donnée correspondra à l'affectation d'une phase au changement de voie correspondant. Le problème est alors de trouver un coloriage des sommets tel qu'une arête relie toujours deux sommets de couleurs différentes (deux voies de communications incompatibles sont dans deux phases distinctes) et qui utilise un nombre de couleurs minimum.

Ce problème est encore une fois un problème classique sur les graphes, dit problème de coloriage des graphe. Le nombre minimum de couleur nécessaire au coloriage est d'ailleurs un paramètre fondamental du graphe appelé nombre chromatique du graphe. On sait malheureusement que ce problème du calcul du nombre chromatique est "NP-difficile" (notion analysée plus en détail dans le cours d'optimisation combinatoire), ce qui implique qu'aucun algorithme polynomial n'est disponible à l'heure actuelle pour le résoudre. La modélisation apporte ici, en dehors d'une formalisation du problème, un résultat théorique

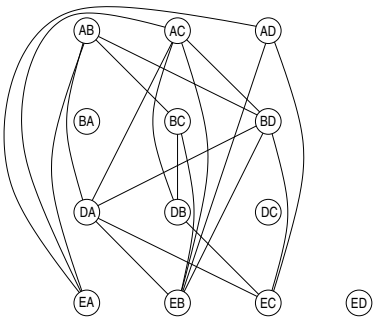

FIGURE 3 – Le graphe d'incompatibilité

fort sur sa difficulté. Heureusement, la taille limité du graphe n'empêchera pas ici la résolution du problème par exploration de toutes les combinaisons possibles.

Chapitre 1

Introduction, notations, définitions

La théorie des graphes est riche en vocabulaires et définitions. Voici un court extrait des définitions essentielles. Les anglophones et les francophones utilisent un vocabulaire qui n'est pas toujours simplement équivalent.

1.1 Graphes orientés et non orientés

Définition 1 (Graphe orienté) *un graphe orienté (directed graph ou digraph)* $G = (S, A)$ *est défini par :*

- *un ensemble* S *fini de sommets (vertices, un sommet : a vertex)*
- *un ensemble* A *d'arcs (edges), chaque arc ayant une origine (source) et un but (target) dans* S*. Un arc* $a = (s, t)$ *d'origine* s*, de but* t *est généralement noté* $\overset{a}{s \to t}$*.* t *est un successeur de* s*,* s *un prédécesseur de* t*.*
  Un arc $s \to s$ *est une boucle. On appelle* $|S|$ *l'ordre du graphe. Pour tout sommet* $u \in S$*, on note* $\Gamma^+(u)$ *(resp.* $\Gamma^-(u)$*) l'ensemble des sommets* v *tels que* (u, v) *(resp.* (v, u)*) soit un arc de* A*.*

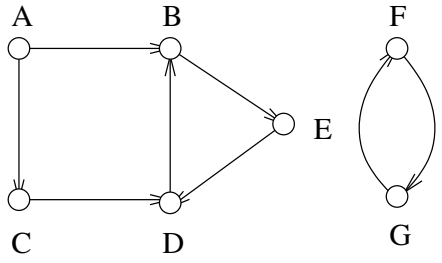

FIGURE 1.1 – Un graphe orienté, G_1

Les graphes orientés sont utilisés en particulier pour modéliser tous les problèmes faisant apparaître des relations de précédence (ordre partiel).

Définition 2 (Graphe non orienté) *un graphe non orienté* $G = (S, A)$ *est défini par :*

- *un ensemble* S *fini de sommets (vertices, un sommet : a vertex)*
- *un ensemble* A *d'arêtes (edges), chaque arête connectant deux sommets (pas nécessairement différents). Une arête* $a = (x, y)$ *est généralement notée* $x - y$. *On dit que* x *et* y *sont adjacents.*
  Pour tout sommet $u \in S$, *on note* $\Gamma(u)$ *l'ensemble des sommets adjacents à* u.

On perd de la notion d'orientation des arcs. Tout graphe orienté admet un graphe non orienté sous-jacent (obtenu par perte de l'orientation). De même, on associe fréquemment à un graphe non orienté un graphe orienté dans lequel chaque arête (x, y) est remplacé par deux arcs (x, y) et (y, x).

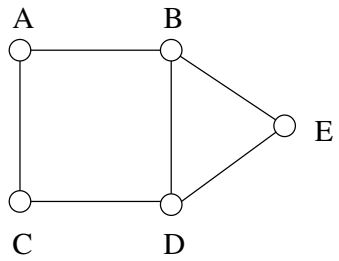

FIGURE 1.2 – Un graphe non orienté, G_2

Définition 3 (Graphe simple) *Un graphe est dit simple s'il existe au plus un arc (arête) d'origine et de buts donnés et s'il est sans boucle.* A *défini dans ce cas une relation binaire sur* S, *définie comme une partie de* $S \times S$. *Dans ce cas, on note un arc* $s \to t$ *ou* (s, t).

Le vocabulaire des relations s'applique : un graphe orienté est antisymétrique si $s \to t \in A \Rightarrow t \to s \notin A$. Un graphe non orienté simple défini une relation symétrique.

Dans la suite, sauf exception, les graphes considérés seront simples. Un arc/arête est alors entièrement caractérisé par ses extrémités.

Définition 4 (degrés) *Dans un graphe orienté, le degré sortant* $d^+(s) = |\Gamma^+(s)|$ *(resp. entrant* $d^-(s) = \Gamma^-(s)|$*) ou demi-degré extérieur (resp. intérieur) d'un sommet* s *est le nombre de sommets qui sont successeurs (resp. prédécesseurs) de* s. *Le degré est la somme des degrés rentrants et sortants.*

Dans un graphe non orienté, le degré $d(s) = |\Gamma(s)|$ *de* s *est le nombre de sommets adjacents à* s

Définition 5 (Sous-graphe) *Soit un graphe* $G = (S, A)$, *et* $S' \subset S$, *le sous-graphe induit par* S' *est le graphe* (S', A') *où* $A' = \{(u, v) \in A / u \in S' \wedge v \in$

$S'\}$. *On ne considère donc que les arcs/arêtes ayant leurs extrémités dans l'ensemble de sommets considérés.*

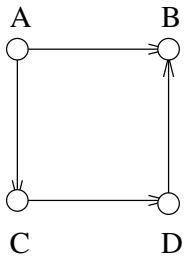

FIGURE 1.3 – Le sous-graphe de G_1 induit par $\{A, B, C, D\}$

Définition 6 (Graphe partiel) *Soit un graphe* $G = (S, A)$*, et* $A' \subset A$*, le graphe* (S, A') *est un graphe partiel de* G*.*

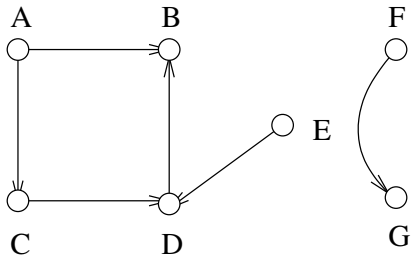

FIGURE 1.4 – Un graphe partiel de G_1

Définition 7 (Graphe complet) *Un graphe est complet si pour toute paire de sommets* (u, v) *il existe au moins un arc (arête) entre* u *et* v*.*

Si le sous-graphe induit par un ensemble de sommets $S' \subset S$ *est complet on dit que* S' *forme une clique. Un graphe simple complet d'ordre* n *est noté* K_n*.*

Définition 8 (Graphe biparti) *Un graphe non orienté* $G = (S, A)$ *est biparti si* $S = S_1 \cup S_2$*, s'il n'existe pas d'arête entre les sommets de* S_1 *et s'il n'existe pas d'arête entre les sommets de* S_2*.*

1.2 Chemins, circuits, chaînes et cycles

Encore une fois, l'école française distingue les notions liées aux graphes orientés de celles liées aux graphes non orientés.

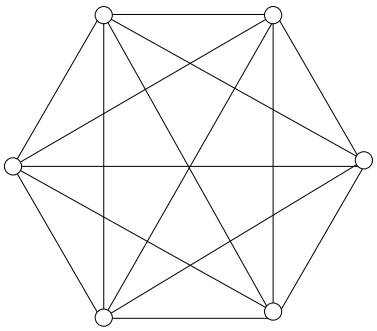

FIGURE 1.5 – Le graphe simple complet non orienté K_6

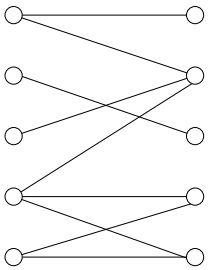

FIGURE 1.6 – Un graphe biparti

Définition 9 (Chemins,circuits) *Soit un graphe orienté* $G = (S, A)$*, un chemin de longueur* k *d'un sommet* u *appelé extrémité initiale vers un sommet* u' *appelé extrémité finale/terminale est une séquence* $\langle v_0, \ldots, v_k \rangle$ *telle que* $v_0 = u, v_k = u'$ *et* $\forall i, 0 \leq i \leq k-1, v_i \rightarrow v_{i+1} \in A$*.*

S'il existe un chemin de u *vers* u'*, on dit que* u' *est accessible à partir de* u*.*

Un chemin est dit simple s'il ne passe pas deux fois par le même arc. Il est dit élémentaire s'il ne passe pas deux fois par le même sommet (tous les sommets sont de degré 2 au plus dans le sous-graphe partiel défini par le chemin).

Un circuit est un chemin dont l'origine et le but sont confondus. Il est dit élémentaire si tous ses sommets sont distincts (à l'exception du premier et du dernier, de façon équivalente : tous les sommets sont de degré 2 exactement dans le sous-graphe partiel défini par le circuit).

Un DAG (Directed Acyclic Graph) ou graphe orienté acircuitique est un graphe orienté sans circuit (GOSC). Le sous-graphe de G_1 induit par $\{A, B, C, D\}$ est un GOSC.

Les équivalents non orientés des chemins et des circuits sont les chaînes et les cycles. Les anglophones utilisent path et cycle dans les 2 cas. Le même type de confusion, généralement peu génératrice d'ambiguïté, devient fréquente en français (mais est parfois mal supportée).

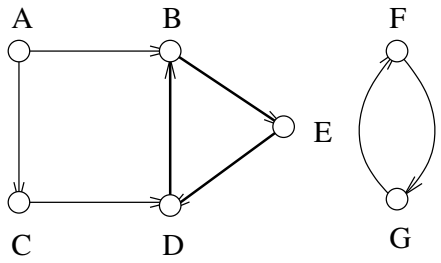

FIGURE 1.7 – Un circuit dans G_1

Définition 10 (Chaînes,cycles) *Soit un graphe non orienté* $G = (S, A)$*, une chaîne de longueur* k *entre un sommet* u *et un sommet* u' *appelé but est une séquence* $\langle v_0, \ldots, v_k \rangle$ *telle que* $v_0 = u, v_k = u'$ *et* $\forall i, 0 \leq i \leq k-1, (v_i, v_{i+1}) \in A$*.*

Une chaîne dont les deux extrémités sont confondues est appelé un cycle.

Définition 11 (Fermeture transitive) *La fermeture transitive d'un graphe (simple)* $G = (S, A)$ *est le graphe* $G' = (S, A')$ *tel que :*

$$(i, j) \in A' \Leftrightarrow \textit{il existe un chemin/chaîne de } i \textit{ à } j$$

1.3 Forte connexité, connexité

Définition 12 *Un graphe orienté* $G = (S, A)$ *est fortement connexe si pour tous sommets* $s, t \in S$*, il existe un chemin allant de* s *à* t*.*

La composante fortement connexe d'un sommet $s \in S$ *est le sous-graphe induit par l'ensemble des sommets* $t \in S$ *tels qu'il existe une chemin de* s *à* t *et de* t *à* s *(classe d'équivalence de la relation d'accessibilité mutuelle).*

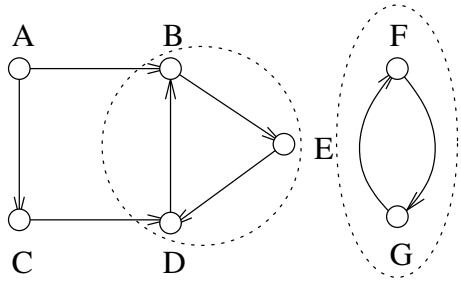

FIGURE 1.8 – Les composantes fortement connexes de G_1

Définition 13 (Connexité) *Un graphe (non orienté)* $G = (S, A)$ *est connexe s'il existe une chaîne entre chaque paire de sommets* $s, t \in S$.

La composante connexe d'un sommet $s \in S$ *est le sous graphe induit par tous les sommets reliés à* s *par une chaîne au moins (classe d'équivalence de la relation d'existence d'une chaîne).*

Le graphe G_2 n'a qu'une composante connexe : tous les sommets sont accessibles entre eux.

1.4 Arbres, arborescences

Définition 14 (Arbre) *Un graphe non orienté et sans cycle (acyclique) est aussi appelé une forêt. Chaque composante connexe d'une forêt est appelé un arbre.*

Un arbre non trivial (qui contient plus d'un sommet) contient toujours au moins 2 sommets de degré 1. En effet, comme il est sans cycle, il contient une plus longue chaîne élémentaire dont les deux extrémités ont forcément un degré de 1. Tout arbre de taille n peut donc être obtenu est greffant un sommet pendant à un arbre de $n-1$ sommets. L'inverse est évidemment vrai (on ne peut créer de cycle ou de non connexité ainsi).

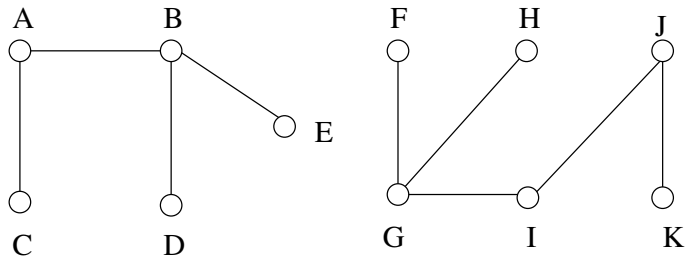

FIGURE 1.9 – Une petite forêt

Théorème 1 *Soit un graphe* $G = (S, A)$. *Les propositions suivantes sont équivalentes :*

1. G *est un arbre*
2. G *est connexe, a* n *sommets et* $m = n - 1$ *arêtes.*
3. G *est sans cycle a* n *sommets et* $n - 1$ *arêtes.*
4. G *est sans boucle et il existe une et une seule chaîne entre chaque paire de sommets.*
5. G *est connexe et la suppression d'une arête quelconque de* G *entraîne la non-connexité.*

6. G *est sans cycle et l'adjonction d'une arête quelconque crée un et une seul cycle.*

Définition 15 (Arborescence) *Un arbre enraciné ou arborescence est un graphe orienté possédant un sommet privilégié appelé racine et tel qu'il existe un chemin et un seul de la racine à tout autre sommet.*

On peut obtenir une arborescence à partir de tout arbre et d'un sommet quelconque de l'arbre en orientant les arêtes depuis la racine vers les autres sommets.

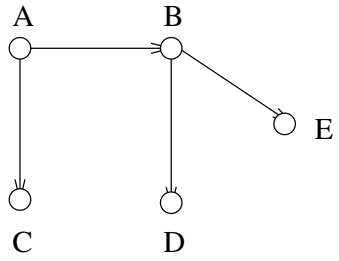

FIGURE 1.10 – Arborescence obtenue à partir de la composante connexe $\{A, B, C, D, E\}$ de la forêt précédente, en isolant A comme racine

En pratique, on confond souvent les deux termes (arbres/arborescences). Sur les arborescences, on emploie un vocabulaire spécifique : nœud, branche, père, fils, frère, ancêtre, descendant, feuille, profondeur...

Une arborescence est ordonnée si les ensemble des fils de chaque sommet sont ordonnés.

Chapitre 2

Représentation machine

2.1 Listes d'adjacence

Adaptée aux graphes peu denses. Étant donné un graphe $G = (S, A)$, on utilise :

- un tableau Adj de $|S|$ listes, une pour chaque sommet.
- pour chaque sommet u, la liste $Adj[u]$ contient les (ou des références aux) sommets v adjacents à u (tels qu'il existe un arc/arête $(u, v) \in A$).

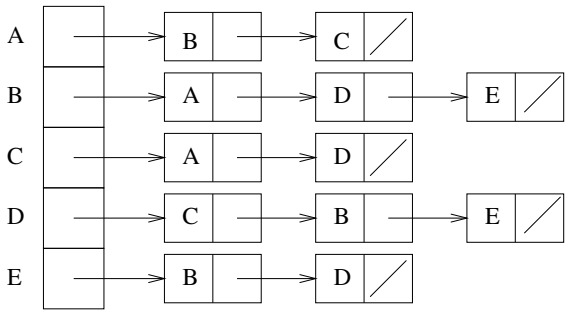

FIGURE 2.1 – La structure de liste d'adjacence du graphe G_2

Pour un graphe orienté, la somme des longueurs des listes est $|A|$ chaque arc étant représenté exactement une fois dans la liste du sommet origine. Pour un graphe non orienté, elle est de $2|A|$ car chaque arête est représentée une fois dans chacun des 2 sommets de l'arête. Gros avantage, la taille mémoire de la structure est en $O(S + A)$. Elle s'adapte aussi au cas où le graphe est étiqueté en stockant l'étiquette dans les listes d'adjacences. Faiblesse essentielle : lenteur de la résolution du problème de l'existence d'une arête entre deux sommets.

Dans un compromis temps/mémoire où le temps prime, on peut utiliser des représentations redondantes (en utilisant aussi une matrice d'adjacence, voir ci-dessous).

2.2 Matrice d'adjacence

Aussi appelé matrice d'incidence sommets-sommets. Pour les graphes simples, l'ensemble des arcs définit une relation binaire entre les sommets que l'on peut représenter par sa matrice associée

Définition 16 (Matrice d'adjacence) *La matrice d'adjacence $\mathcal{A}$ d'un graphe $G = (S, A)$ d'ordre n est une matrice $n \times n$ à coefficients dans $\{0, 1\}$ où chaque ligne et chaque colonne correspond à un sommet de G et où :*

$$\mathcal{A}_{ij} = 1 \Leftrightarrow (i, j) \in A$$

Dans le cas non orienté, la matrice d'adjacence est symétrique (une demi-matrice suffit).

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

TABLE 2.1 – La matrice d'adjacence du sous-graphe de G_1 induit par $\{A, B, C, D, E\}$

Cette forme convient bien aux graphes denses (sinon, la matrice est creuse). Elle permet d'utiliser tout l'attirail du calcul matriciel, mais le coût de base d'accès à toutes les arêtes est en $\theta(|S|^2)$ et non en $\theta(A)$.

Si l'on a affaire à un graphe étiqueté (ou pondéré), chaque arête est porteuse d'une étiquette que l'on peut utiliser dans la matrice en lieu et place de 1. Il faut pouvoir distinguer le cas 0.

Calculs matriciels

On rappelle que le produit de deux matrices $n \times n$ naïf est en $O(n^3)$. Pas d'améliorations importantes (eg. algorithme de Sollen en $O(n^{\log_2(7)})$).

Pour manipuler les matrices, on utilise ici l'algèbre de Boole (multiplication $= \wedge$, addition $= \vee$). D'autres algèbres permettent de résoudre d'autres problèmes sur les graphes.

Si M est la matrice d'adjacence d'un graphe G, la matrice M^p est telle que $M^p_{ij} = 1 \Leftrightarrow$ il existe un chemin de p arcs allant i à j. L'algorithme récursif suivant permet ce calcul en $O(\log_2(p)S^3)$.

S'il existe un chemin entre deux sommets de G, il existe un chemin élémentaire en ces sommets, de longueur $< n$. La matrice de la fermeture transitive du graphe est donc la somme (le ou logique) $M + M^2 + \cdots + M^{|S|-1}$. Une première

Algorithme 1 : Entrées : M, matrice d'adjacence, p puissance demandée

```
Procedure Calcul de M^p
    si p = 1 alors
        retourner M;
    sinon
        J ← M^⌊p/2⌋ (appel récursif);
        K ← J × J;
        si p est pair alors
            retourner K;
        sinon
            retourner K × M;
```

Algorithme 2 : Entrée : M, matrice d'adjacence

```
Procedure FermetureTransitive(M)
    pour i allant de 1 à n faire
        pour j allant de 1 à n faire
            si M_ji = 1 alors
                pour k allant de 1 à n faire
                    si M_ik = 1 alors M_jk ← 1
```

version purement matricielle de cet algorithme (ce calcul peut se réaliser plus simplement en utilisant l'algorithme de Warshall) est :

La complexité est en $O(|S|^3)$ et même en $\theta(|S|^3)$ et beaucoup de calculs inutiles sont réalisés si la matrice est creuse (le graphe peu dense). On peut dans ce cas utiliser une version plus proche de l'esprit des graphes.

```
Procedure FermetureTransitive (G)
    pour chaque sommet s de G faire
        pour chaque arc s1 → s de G faire
            pour chaque arc s → s2 de G faire
                Ajouter à G l'arc s1 → s2;
```

Preuve de correction : par récurrence sur le nombre de passage dans la boucle extérieure. L'hypothèse est que si k sommets ont déjà été traités, alors un arc $s_1 \to s_2$ a été rajouté dans le graphe ssi il existe un chemin de s_1 à s_2 dont les sommets intermédiaires sont parmi ces k sommets. Il suffit alors de remarquer que s'il existe un chemin de s_1 à s_2 passant par un sommet s alors il existe un chemin passant une seule fois par s (on prend un raccourci), donc tout chemin de s_1 à s_2 utilisant les $k+1$ premiers sommets est formé par la concaténation de chemins allant de s_1 à s, puis de s à s_2 et n'utilisant que les k premiers sommets comme sommets intermédiaires.

Chapitre 3

Parcours des graphes

On distingue deux grandes méthodes de parcours de graphes : le parcours en largeur et en profondeur. Du fait de son importance et de sa simplicité, nous considérons d'abord le parcours en profondeur, en commençant par le parcours des arborescences, plus simple.

3.1 Parcours des arborescences en profondeur

La stratégie de parcours d'un arbre en profondeur consiste, comme son nom l'indique, à s'enfoncer dans l'arbre à chaque fois que possible. Le premier fils est visité immédiatement à partir du sommet le plus récemment visité récursivement. Quand tous les fils d'un sommet ont été explorés, on effectue un retour en arrière pour explorer les fils suivants du dernier sommet qui n'est pas encore complètement visité (tous ses fils n'ont pas encore été visités). Le processus continue tant qu'il reste des sommets non complètement visités.

Cette exploration permet d'associer à chaque sommet une date de découverte (moment où l'on rencontre un sommet pour la première fois) et une date de fermeture (moment où tous les fils ont été visités et on l'on effectue un retour en arrière). Pour chaque sommet u, on notera $d[u]$ et $f[u]$ ces deux dates, calculées si nécessaire par l'algorithme ci-dessous (et que l'on utilisera par la suite).

Algorithme 3 : On appelle la fonction avec u égal à la racine de l'arborescence. t est une variable globale initialisée à 0

```
Procedure DFS-Tree(u)
    d[u] ← (t ← (t + 1));
    pour chaque sommet v, fils de u faire
        DFS-Tree(v);
    f[u] ← (t ← (t + 1));
```

3.2 Parcours des graphes en profondeur d'abord

Le parcours en profondeur s'adapte aux graphes quelconques en parcourant un graphe partiel qui est un arborescence (ou une forêt) : à la découverte d'un sommet s, les fils de ce sommet dans l'arborescence seront les sommets accessibles depuis s et non encore visités. Cet ensemble d'arborescences est appelé forêt en profondeur d'abord du graphe. Comme sa connaissance peut-être utile, on peut la mémoriser en se souvenant, pour chaque sommet s du graphe exploré son père dans cette arborescence. On stockera ainsi le père du sommet u dans la variable $\pi[u]$.

Pour pouvoir distinguer les sommets déjà explorés (découverts), complètement explorés et non encore explorés, on associe à chaque sommet une couleur : b ou blanc (non exploré), g ou gris (découvert) et n ou noir (complètement exploré). Cette couleur est stockée dans $couleur[u]$. On notera que la distinction de gris et noir n'est pas indispensable au fonctionnement de l'algorithme.

L'algorithme ressemble fortement au précédent mais est adapté à un graphe quelconque :

Algorithme 4 : On appelle la fonction sur le graphe $G = (S, A)$ (orienté ou non)

```
Procedure DFS(G)
    pour chaque sommet u ∈ S faire
        couleur[u] ← b;
        π[u] ← NIL;
    t ← 0;
    pour chaque sommet u ∈ S faire
        si couleur[u] = b alors DFS-Visit(u)

    DFS-Visit(u);
    couleur[u] ← g;
    d[u] ← (t ← (t + 1));
    pour chaque sommet v ∈ Adj[u] faire
        si couleur[v] = b alors
            π[v] ← u;
            DFS-Visit(v);
    couleur[u] ← n;
    f[u] ← (t ← (t + 1));
```

Quel est le temps de calcul de cet algorithme ? L'initialisation dans DFS parcourt chaque sommet donc en $\theta(|S|)$. La procédure DFS-Visit est appelée une fois pour chaque sommet puisqu'elle n'est invoquée que sur les sommets blancs et que la première chose qu'elle fait est de les colorier en gris. Durant l'exécution de DFS-Visit une boucle s'effectue sur les sommets adjacents. Comme la somme des degrés (sortants) de tous les sommets est en $\theta(|A|)$, le coût total de DFS-Visit est en $\theta(|A|)$. Globalement, DFS est donc en $\theta(|S|+|A|)$ et est donc linéaire

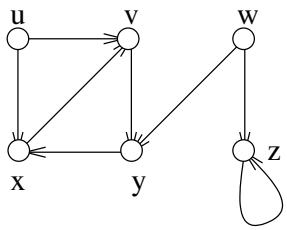

FIGURE 3.1 – Parcourir ce graphe en profondeur d'abord

dans la taille du graphe (sous forme de listes d'adjacence).

Le parcours effectué par l'algorithme DFS a de nombreuses propriétés que nous ne détaillerons pas ici. Il permet en particulier de classer les arcs du graphe parcouru en 4 catégories (si le graphe est non orienté, seules les deux premières catégories existent) :

- arcs d'arbre : ce sont ceux qui sont suivis par DFS. Un arc $u \to v$ est un arc d'arbre si v a été découvert à partir de u. v était blanc à sa découverte.
- arcs arrières : ce sont les arcs qui connectent un sommet u à un ancêtre dans l'arborescence de profondeur d'abord. v était gris à sa découverte. Un circuit est détecté dans le graphe.
- arcs avants : ce sont les arcs qui connecte un sommet u à un descendant de u mais qui n'ont pas été suivies par DFS. v était noir à sa découverte.
- arcs traversiers : tous les autres arcs. v était noir à sa découverte.

Exercice : appliquer à la main DFS sur le graphe orienté de la figure 3.1, à partir de u, les sommets étant considérés dans l'ordre u, v, w, x, y, z. On visualisera la forêt d'exploration en profondeur d'abord et on notera pour chaque sommet sa date de découverte et de fermeture, pour chaque arc sa nature (arbre, arrière, avant, traversier).

3.2.1 Application, le tri topologique

Nous citons une application de la recherche en profondeur d'abord, mais il en existe d'autres : recherche de composantes connexes, de composantes fortement connexes... On se reportera à [CLR90, GM79, AHU87] par exemple pour plus de détails.

Un tri topologique d'un GOSC G consiste à déterminer un ordre des sommets tel que si un arc $u \to v$ existe dans G, alors v est placé après u dans l'ordre des sommets. Nous verrons un utilisation de cette technique pour calculer les plus courts chemin dans un GOSC de façon très efficace. Plus généralement, si l'on considère que les sommets d'un GOSC représente des tâches et que chaque arc indique une dépendance, le tri topologique indique un ordre des tâches qui respecte les dépendances.

L'algorithme reste de la même complexité, i.e. en $\theta(|S| + |A|)$.

En pratique, l'algorithme DFS permet même de détecter si un graphe orienté est sans circuit car on montre qu'un graphe orienté est sans circuit si et

Procedure *Tri-Topologique(*G*)*
  Appeler DFS(G) pour calculer les f[u];
  à chaque fois qu'un sommet est fermé, l'insérer en tête d'une liste L;
  retourner L;

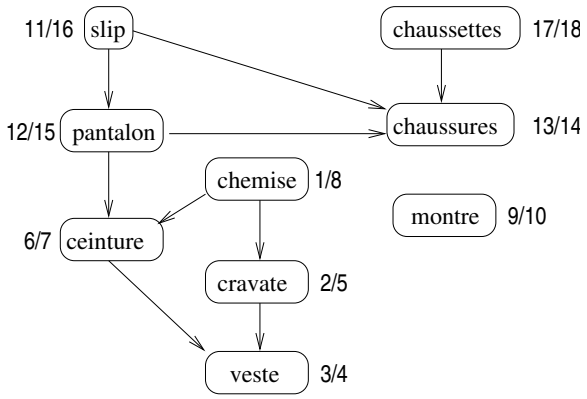

FIGURE 3.2 – Un problème d'ordonnancement élémentaire

seulement si un parcours en profondeur d'abord ne détecte pas d'arcs arrières. Il permet très simplement d'identifier les composantes connexes d'un graphe non orienté : chaque appel non récursif à DFS-Visit parcourt une composante connexe distincte.

Exercice : appliquer l'algorithme Tri-Topologique au graphe de la figure 3.2 (dépendances entre les tâches élémentaires d'habillage).

3.3 Parcours des graphes en largeur d'abord

Dans le cas des arborescences, le parcours en largeur consiste à visiter les nœuds "de gauche à droite et de haut en bas". Elle s'effectue avec une simple file (FIFO).

Nous définissons immédiatement la version étendue aux graphes. La file est appelée Q dans la fonction BFS. Comme dans la recherche en profondeur, les sommets inexplorés sont blancs, les sommets en cours d'explorations sont gris et les sommets fermés en noir. Le niveau de découverte d'un sommet est stocké dans les variables d[u] où u est un sommet. L'exploration en largeur d'abord d'un graphe définit une arborescence sur ce graphe appelé arborescence en largeur d'abord. Le parcours mémorise cette arborescence en associant une variable π[u] à chaque sommet qui contiendra *in fine* le père du sommet u dans cette arborescence.

Analysons rapidement la complexité de cet algorithme : une fois l'initialisation effectuée, aucun sommet n'est jamais blanchi. De ce fait, comme seuls

Algorithme 5 : On appelle la fonction sur le graphe $G = (S, A)$, à partir d'un sommet s

```
Procedure BFS(G, s)
    pour chaque sommet u ∈ S − {s} faire
        couleur[u] ← b;
        d[u] ← +∞;
        π[u] ← NIL;
    couleur[s] ← g;
    d[s] ← 0;
    π[s] ← NIL;
    Q ← {s};
    tant que Q ≠ ∅ faire
        u ← tete(Q);
        pour chaque sommet v ∈ Adj[u] faire
            si couleur[v] = b alors
                π[v] ← u;
                couleur[v] ← g;
                d[v] ← d[u] + 1;
                Enfiler(Q, v);
        Défiler(Q);
        couleur[u] ← n;
```

les sommets blancs peuvent entrer dans la file et qu'ils sont aussitôt coloriés en gris, chaque sommet entre (et sort) une fois dans la file. Ces opérations étant en $O(1)$, on aboutit à un premier coût en $O(|S|)$. La liste d'adjacence de chaque sommet est parcourue une fois et une seule, quand le sommet est extrait de la file. La somme des longueurs des listes d'adjacences est en $\theta(|A|)$. L'initialisation étant en $O(|S|)$, l'algorithme est globalement en $O(|S| + |A|)$ et est donc linéaire dans la taille du graphe (sous forme de listes d'adjacence).

Exercice : appliquer l'algorithme BFS au graphe de la figure 3.3, en partant du sommet s et en détaillant les couleurs des sommets et les valeurs des variables $\pi[v]$ et $d[v]$ et Q à chaque étape.

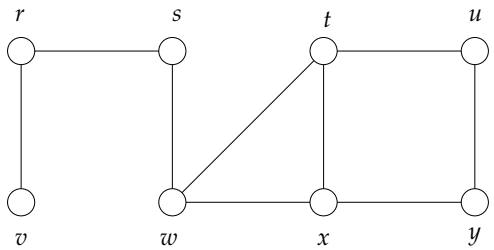

FIGURE 3.3 – Un graphe à parcourir en largeur

Le parcours en largeur d'abord offre la particularité intéressante d'identi-

fier le plus court chemin entre le sommet s et les autres sommets du graphe (si l'on compte chaque parcours d'arête ou d'arc pour une longueur unitaire). L'arborescence en largeur d'abord est alors une arborescence des plus courts chemin (cf. section 5).

Chapitre 4

Arbres couvrants de poids minimum

À partir de maintenant, nous allons travailler sur des graphes dits pondérés : chaque arc/arête est porteuse d'un « poids », habituellement un nombre entier ou réel quelconque. Pour tout arc ou arête (u, v) d'un graphe pondéré nous noterons $w(u, v)$ le poids qui lui est associé.

Dans la conception de circuits électroniques ou de circuits de communications (en avionique par exemple), il est souvent nécessaire de connecter plusieurs « composants » ensemble en utilisant le moins de fils possible. On peut parfois modéliser ce problème sous la forme d'un graphe connexe non orienté et pondéré dans lequel chaque sommet est un « composant » et chaque arête (u, v) entre deux composants est pondérée par le coût (la longueur) de l'établissement d'une connexion entre u et v. On cherche alors à déterminer un sous-ensemble de ces arêtes qui définisse un graphe partiel acyclique, connectant tous les sommets du graphe et dont le poids soit minimum (le poids d'un ensemble d'arête étant défini comme la somme des poids des arêtes de l'ensemble).

Ce problème consiste à exhiber un « arbre couvrant de poids minimum » (appelé *minimum spanning tree* par les anglophones). Il s'agit d'un problème

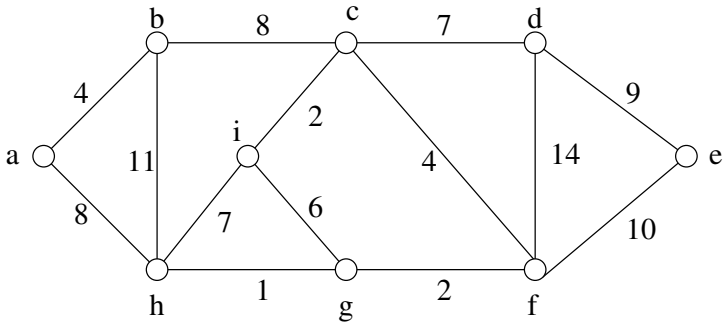

FIGURE 4.1 – Un graphe non orienté pondéré et un arbre couvrant de poids minimum

que l'on peut résoudre de façon optimale en temps polynomial et ce malgré le fait qu'il existe un nombre a priori exponentiel de solution potentielles.

Il existe deux familles d'algorithme pour résoudre ce problème : l'algorithme de Kruskal et l'algorithme de Prim [1]. En utilisant une simple structure de tas binaire (*heap*), ces algorithmes ont un temps d'exécution en $O(|A|.\log(|S|))$. En utilisant des tas de Fibonacci, L'algorithme de Prim peut être rendu plus efficace, en $O(|A| + |S|.\log(|S|))$ qui peut être significatif si $|S|$ est très faible devant $|A|$. Pour cette raison, nous présenterons uniquement ce dernier. Nous invitions le lecteur à se reporter à [CLR90] pour découvrir l'algorithme de Kruskal, très simple lui aussi.

L'algorithme de Prim est un exemple d'algorithme dit « glouton » dans le sens où lorsqu'il a un choix à faire, l'algorithme va simplement choisir le meilleur choix parmi ceux qui sont « possibles ». Pour la grande majorité des problèmes, les algorithmes gloutons ne garantissent pas de trouver l'optimum. Le problème de recherche d'un arbre couvrant de poids minimum est particulier de ce point de vue.

L'algorithme s'appuie implicitement sur la notion de cocycle :

Définition 17 *Soit un graphe* $G = (S, A)$ *non orienté, connexe et pondéré par la fonction de pondération* p *de* A *dans* $\mathbb{R}$. *Soit un ensemble de sommets* $T \subset S$ *quelconque. On appelle cocycle de* T, *et on note* $\omega(T)$, *l'ensemble des arêtes de* A *dont une des extrémités est dans* T *et l'autre dans* $S - T$.

Soit $C \subset S$ un ensemble de sommets non vide quelconque de G et $\omega(C)$ le cocycle de C. Soit (u, v) une arête de poids minimum dans $\omega(C)$. On va montrer qu'il existe nécessairement un arbre couvrant de poids minimum qui contient (u, v).

Soit T un arbre couvrant de poids minimum quelconque. Soit T contient (u, v) et la propriété est vérifiée. Sinon, (u, v) forme un cycle fermé par le chemin p de u à v dans T. Ce cycle permet de quitter C et d'y revenir et utilise donc une autre arête du cocycle. Soit (x, y) l'arête du cocycle qui permet de revenir dans C (voir Figure 4.2). Si l'on enlève (x, y) de T et qu'on la remplace par (u, v), on conserve toutes les propriétés d'un arbre couvrant et on ne peut que diminuer le poids de l'arbre car e est de poids minimum parmi les arêtes du cocycle.

Pour former le sous-ensemble des arêtes qui formera l'arbre couvrant, l'algorithme de Prim va faire grossir un arbre (au début formé d'un unique sommet) en rajoutant des arêtes à cet arbre. Pour cela, il va sélectionner un sommet qui n'est pas encore dans l'arbre, et qui est adjacent à un sommet de l'arbre courant (une arête du cocycle de l'ensemble de sommets déjà couverts). La gloutonnerie de l'algorithme à chaque étape consiste à choisir parmi tous les sommets possibles, le sommet qui va le moins faire augmenter le poids de l'arbre : celui qui est

1. Pour mémoire, on notera aussi l'existence d'un algorithme qui est en quelque sorte un hybride des deux précédents, l'algorithme de Sollin.

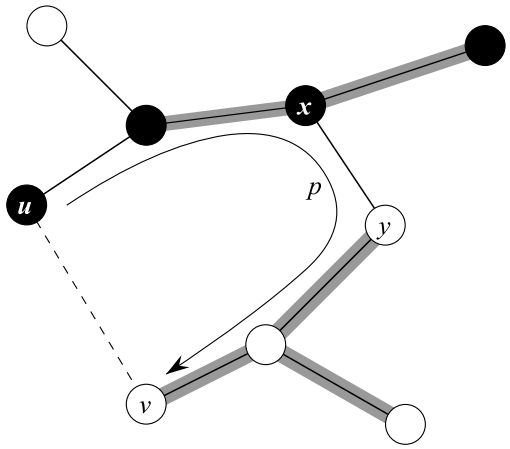

FIGURE 4.2 – Un graphe G, un ensemble de sommets C en noir avec (u, v) et (x, y) les arêtes du cocycle de C qui forme un cycle sur le chemin p

relié à l'arbre par une arête de poids minimum. Ce choix garantit l'obtention d'une solution optimale comme on vient de le voir.

L'algorithme utilise à cette fin une file de priorité Q de sommets qui peut être implémentée au moyen d'un tas binaire ou d'un tas de Fibonacci (plus complexe). On suppose que la fonction Extrait-Minimum extrait du tas le sommet de priorité la plus faible. La priorité est stockée dans $key[u]$. Cette file va permettre d'accéder rapidement a la meilleure façon d'étendre l'arbre que l'algorithme construit. Initialement, la racine de l'arbre a une priorité nulle et tous les autres sommets une priorité infinie. Lorsqu'un nouveau sommet u est couvert, on actualise la priorité des sommets adjacents à u avec la valeur de l'arête qui les relient à u.

Algorithme 6 : La fonction est appelée sur le graphe G, muni d'une pondération w et à partir d'un sommet r.

```
Procedure MST-Prim(G, w, r)
    Q ← ensemble des sommets du graphe;
    pour chaque u ∈ Q faire key[u] ← +∞;
    key[r] ← 0;
    π[r] = NIL;
    tant que Q ≠ ∅ faire
        u ← Extrait-Minimum(Q);
        pour chaque v ∈ Adj[u] faire
            si v ∈ Q et w(u, v) < key[v] alors
                π[v] ← u;
                key[v] ← w(u, v);
```

NB : l'algorithme permet aussi de construire un arbre couvrant de poids maximum. Il suffit de changer le signe des poids.

Exercice : appliquer l'algorithme MST-Prim au graphe de la figure 4.1. Détailler chaque étape.

Chapitre 5

Plus courts chemin, plus long chemin et ordonnancement

Un automobiliste souhaite trouver la route la plus courte de Toulouse à Lille. S'il dispose d'une carte routière donnant la distance entre chaque intersection adjacentes, comment déterminer la route la plus courte ?

Là encore, le nombre de chemins possibles entre deux villes, pourvu que le réseau routier soit un peu dense, devient rapidement trop important pour que l'on puisse envisager d'énumérer tous les chemins possibles, même si l'on évite d'examiner les chemins qui passe par Brest (qui est manifestement une mauvaise option) ou contiennent des cycles (et qui sont forcement plus long que le même chemin sans cycle). Nous allons voir qu'il est possible de résoudre ce problème efficacement.

Dans un problème de plus court chemin, on dispose d'un graphe (orienté), avec une pondération des arcs w qui fait correspondre à chaque arc un poids à valeurs réelles. On définit le poids (ou longueur) d'un chemin $c = \langle v_0, \ldots, v_k \rangle$ comme la somme des poids (longueurs) des arcs qui le constituent :

$$w(c) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

On définit alors la distance minimum entre deux sommets u et v par :

$$\delta(u,v) = \begin{cases} \min_c \text{ chemin de } u \text{ à } v\{w(c)\} & \text{si il existe un chemin de } u \text{ à } v \\ +\infty & \text{sinon} \end{cases}$$

Un chemin le plus court entre u et v est alors défini comme un chemin de u à v de poids $w(c) = \delta(u,v)$. Si dans sa version la plus classique les poids représentent des distances, rien ne limite l'utilisation de ces algorithmes à ce cas de figures. Il peut s'agir de coûts, de pénalités, ou de l'opposé du logarithme d'une probabilité.

Variantes Plusieurs variantes du problème existe selon que l'on cherche :

- un chemin le plus court entre 2 sommets u t v donnés,
- un chemin le plus court permettant de rejoindre le sommet v à partir du sommet u pour n'importe quel sommet u (on aura donc n plus courts chemins).
- un plus court chemin pour toute paire de sommets u et v (n^2 chemins au total).

De fait, le problème de calcul du plus court chemin entre 2 sommets u et v fixés n'est jamais abordé car on connaît pas d'algorithme qui le traite et qui soit asymptotiquement plus efficace que ceux permettant de traiter le second problème. On peut donc se limiter aux deux derniers.

De même, pour déterminer le plus court chemin entre toutes les paires de sommets, on peut résoudre le problème à une source pour chaque sommet du graphe [1]. Nous nous limiterons donc ici à ce type de problème appelé "problème de plus court chemin à source unique" (ou *single source shortest path*) :

Étant donné un graphe $G = (S, A)$, trouver le plus court chemin d'un sommet s donné (appelé sommet source) vers tout autre sommet du graphe.

Arcs de poids négatifs Un point important pour la résolution des problèmes de plus court chemin est l'existence de circuits ayant un poids négatif. Si ce circuit est atteignable depuis s, les chemins les plus courts ne sont pas définis. On peut en effet améliorer indéfiniment la qualité d'un chemin en faisant un tour de plus dans le circuit. Il existe des algorithmes de plus court chemin qui détectent ces circuits absorbants (tel que l'algorithme de Bellman-Ford-Moore que nous verrons plus loin) et des algorithmes spécialisés dans différents cas où l'on sait qu'il ne peut exister de circuits absorbants ; par exemple dans un graphe où les poids sont tous positifs ou nul (algorithme de Dijkstra) ou si le graphe n'a pas de circuit (algorithme de Bellmann-Kalaba).

Principe d'optimalité Un plus court chemin de longueur finie ne peut donc pas contenir de circuits de coût négatifs. Il ne peut pas non plus contenir de circuits de coûts positif car on supprimant le circuit du chemin, on obtiendrait un chemin plus court. On put donc faire l'hypothèse que les plus courts chemins ne contiennent pas de circuits et la longueur (en nombre d'arcs) d'un plus court chemin est donc toujours inférieure à $|S| - 1$.

La propriété essentielle qui permet de calculer efficacement les plus courts chemin est appelé propriété de Bellman. Elle est à la base de classes d'algorithmes tels que les algorithmes gloutons et surtout la programmation dynamique. Elle se résume par « tout sous-chemin d'un chemin optimal est aussi un chemin optimal ». Si ce n'était pas le cas, on pourrait remplacer la section correspondante par un sous-chemin optimal et améliorer ainsi la longueur totale du chemin. Pour connaître la longueur ℓ du plus court chemin de Toulouse

1. Il existe cependant des algorithmes spécialisés (algorithmes souvent inspirés du calcul matriciel proches de l'algorithme de Warshall décrit précédemment) qui peuvent être plus adaptés.

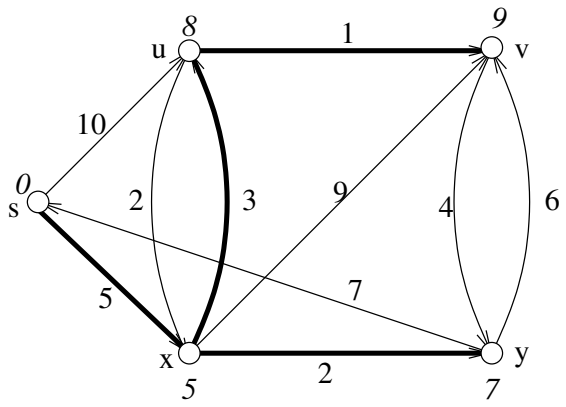

FIGURE 5.1 – Arbre des plus courts chemins, en italique les $\delta(s, u)$.

à Lille, il suffit par exemple de connaître la longueur d'un plus court chemin ℓ_i de Toulouse aux villes i reliées directement à Lille et de choisir d'aller vers la ville i^* qui minimise la somme $\ell_i + w(i, \text{Lille})$.

Représenter les plus courts chemins Pour mémoriser les plus courts chemin de s à tout autre sommet du graphe, on va calculer pour chaque sommet u un prédécesseur $\pi[u]$ dans le graphe par lequel il faut passer si l'on veut aller de s à u. Dans notre exemple, le prédécesseur de Lille est la ville i^*. S'il n'est pas possible d'atteindre u depuis s ce prédécesseur sera égal à NIL. Pour reconstruire le plus court chemin vers n'importe quel sommet u accessible depuis s, il suffira d'opérer en arrière en cherchant le prédécesseur de u, puis son prédécesseur... jusqu'à s. Le graphe G_π formé de s et des sommets du graphe initial qui ont un prédécesseur non égal à NIL et ayant pour arcs les $(\pi[v], v)$ forme ce que l'on appelle le graphe des plus courts chemin. On montre que le graphe ainsi construit est une arborescence de racine s tel que pour tout sommet u l'unique chemin de s à u dans cet arbre est un plus court chemin de s à u.

La figure 5.1 illustre un graphe avec un arbre des plus courts chemins en gras. Chaque sommet u est annoté avec $\delta(s, u)$.

Tous les algorithmes présentés gère un attribut $d[u]$ pour stocker une estimation du plus court chemin de s à u. À la fin de l'exécution, cette estimation sera exacte. La procédure Initialisation-PCC est utilisée pour initialiser ces structures. Initialement, les estimations des plus courts chemins sont donc telles que s est à distance 0 de s et tous les autres sommets sont à distance infinie. L'étape fondamentale des algorithme de plus court chemin consiste à améliorer ces estimations au moyen de la procédure Relaxation.

Elle constitue une application directe de la remarque sur la propriété de Bellman : si pour aller de s à v on connaissait un chemin de longueur $d[v]$ mais que le chemin de s à u plus la longueur de l'arc (u, v) est strictement plus court alors, il vaut mieux passer par u. On met à jour l'estimation de la distance et le prédécesseur en conséquence. On notera que si le graphe G_π des prédécesseurs

Algorithme 7 : La fonction est appelée avec le graphe G et le sommet source s

Procedure *Initialisation-PCC(*G, s*)*
  pour chaque *sommet* v *de* G **faire**
    $d[v] \leftarrow +\infty$;
    $\pi[v] \leftarrow$ NIL;
  $d[s] \leftarrow 0$;

Algorithme 8 : Amélioration éventuelle de l'estimation de la distance de s à v. Le sommet u est un sommet accessible depuis s et qui permet d'accéder à v. w est la pondération du graphe.

Procedure *Relaxation(*u, v, w*)*
  si $d[v] > d[u] + w(u, v)$ **alors**
    $d[v] \leftarrow d[u] + w(u, v)$;
    $\pi[v] \leftarrow u$;

formait une arborescence de racine s avant l'appel à Relaxation, c'est encore le cas après. On notera également que si $d[x]$ est une estimation pessimiste de la distance $\delta(s, x)$ avant l'application de Relaxation, c'est encore le cas après.

5.0.1 Propriétés des plus courts chemins

Les plus courts chemins vérifient toujours un certain nombre de propriétés élémentaires très utiles dans les preuves. On pourra démontrer ces propriétés en exercice.

1. Inégalité triangulaire :
   Pour tout arc $(u, v) \in A$, on a $\delta(s, v) \leq \delta(s, u) + w(u, v)$.
2. Propriété du majorant :
   On a toujours $\delta[v] \geq \delta(s, v)$ pour tous les sommets $v \in S$, et une fois que $d[v]$ a atteint la valeur $\delta(s, v)$, elle ne change plus.
3. Propriété aucun-chemin : S'il n'y a pas de chemin de s à v, alors on a toujours $d[v] = \delta(s, v) = \infty$.
4. Propriété de convergence : Si $s \rightsquigarrow u \rightarrow v$ est un plus court chemin dans G pour un certain $u, v \in S$ et si $d[u] = \delta(s, u)$ à un instant antérieur au relâchement de l'arc (u, v), alors $d[v] = \delta(s, v)$ en permanence après le relâchement.
5. Propriété de relâchement de chemin : Si $p = \langle v_0, v_1, \ldots, v_k \rangle$ est un plus court chemin de $s = v_0$ à v_k et si les arcs de p sont relâchés dans l'ordre (v_0, v_1), $(v_1, v_2), \ldots (v_{k-1}, v_k)$, alors $d[v_k] = \delta(s, v_k)$. Cette propriété est vraie indépendamment de toutes autres étapes de relâchement susceptibles de se produire, même si elles s'entremêlent avec des relâchements d'arcs de p.

6. Propriété de sous-graphe prédécesseur : Une fois que $d[v] = \delta(s, v)$ pour tout $v \in S$, le sous-graphe prédécesseur est une arborescence de plus courts chemins de racine s.

5.1 Algorithme de Bellman-Ford-Moore

Cet algorithme résout le problème de calcul des plus courts chemins à origine unique dans le cas général d'un graphe qui peut avoir des circuits et des poids négatifs (et éventuellement un circuit absorbant). Si un circuit absorbant atteignable depuis l'origine s existe, l'algorithme le détecte et retourne un booléen égal à *vrai*.

Procedure *Bellman-Ford(*G, w, s*)*
  Initialisation-PCC(G, s);
  pour chaque $i \leftarrow 1$ *à* $|S| - 1$ **faire**
    pour chaque $(u, v) \in A$ **faire**
      Relaxation(u, v, w);
  pour chaque $(u, v) \in A$ **faire**
    si $d[v] > d[u] + w(u, v)$ **alors**
      retourner *faux*;

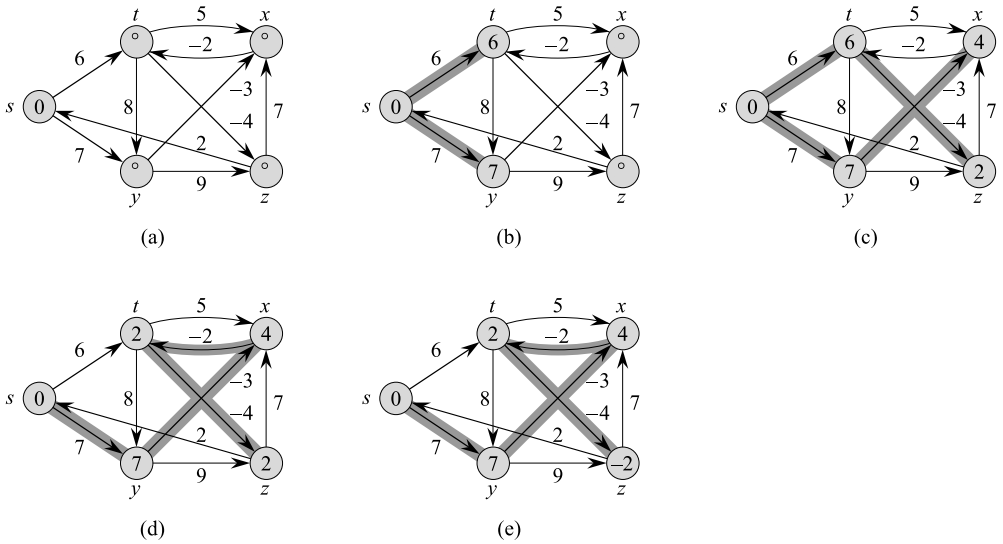

FIGURE 5.2 – Exemple : application de Bellman-Ford sur un graphe.

La complexité de l'algorithme est très facile à calculer. L'initialisation est en $\theta(|S|)$, chaque itération de la première boucle est en $\theta(|A|)$ et s'exécute $|S|-1$ fois. La dernière boucle est en $\theta(|A|)$. Au total, l'algorithme est en $O(|S||A|)$, potentiellement cubique en $|S|$.

Si le graphe ne contient pas de circuit absorbant accessible depuis s, l'algorithme est correct du fait que tout plus court chemin élémentaire contient au plus $|S|-1$ arcs et découle de la propriété de "relâchement de chemin". Si l'on considère un sommet v accessible depuis s et $p = \langle v_0, v_1, \ldots, v_k \rangle$, où $v_0 = s$ et $v_k = v$, un plus court chemin élémentaire de s à v. Le chemin p a au plus $|S|-1$ arcs. A chaque itération i, le couple (v_{i-1}, v_i) est relaxé et donc la propriété de relâchement de chemin s'applique et $d[v] = \delta(s, v)$. L'algorithme retourne de plus $vrai$ du fait de l'inégalité triangulaire.

Inversement, supposons que G contient un circuit absorbant $\langle v_0, v_1, \ldots, v_k \rangle$ accessible depuis $v_0 = v_k = s$ et que l'algorithme retourne $vrai$. Pour tout $i = 1, \ldots, k$, on a $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$. En sommant ces inégalités sur le circuit, on obtient :

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Comme on a un circuit, $\sum_{i=1}^{k} d[v_{i-1}] = \sum_{i=1}^{k} d[v_i]$ et donc $\sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0$ ce qui est en contradiction avec l'hypothèse.

5.2 L'algorithme de Dijkstra

Cet algorithme est adapté aux graphes orientés qui peuvent avoir des circuits mais dont les arêtes ont toutes un poids positif ou nul. On suppose donc ici que $w(u, v) \geq 0$ pour tout arc (u, v) du graphe G.

L'algorithme de Dijkstra maintient un ensemble de sommets V pour lesquels on sait que l'estimation $d[u]$ est devenue exacte i.e., telle que $d[u] = \delta(s, u)$. L'algorithme sélectionne alors le sommet $u \in S - V$ dont l'estimation $d[u]$ est la plus faible, insère u dans V et applique la procédure Relaxation à tous les arcs qui quittent u.

Pour un plus grande efficacité, un file de priorité Q est utilisée pour stocker les sommets de $S - V$, utilisant $d[u]$ comme priorité. On suppose que le graphe est représenté par ses listes d'adjacences.

Le fait que l'algorithme est correct découle du fait que lorsqu'un sommet u est inséré dans V, alors l'estimation pessimiste $d[u] = \delta(s, u)$.

Preuve : Supposons que ce ne soit pas le cas et soit u le premier sommet que l'algorithme insère dans V tel que $d[u] \neq \delta(s, u)$. On doit avoir $u \neq s$ car s est le premier sommet inséré dans V et $d[s] = 0 = \delta(s, s)$. Puisque $u \neq s$, alors quand u est inséré dans V, V est non vide. Il doit de plus y avoir un chemin de s à u car sinon $d[u] = \infty = \delta(s, u)$ (le fait que $d[x]$ est une estimation pessimiste de $\delta[s, x]$ est un invariant de Relaxation). Puisqu'un chemin existe, un plus court chemin existe. Soit un plus court chemin de s à u, y le premier sommet rencontré sur ce chemin qui n'est pas dans V et x son prédécesseur immédiat dans V.

Algorithme 9 : $G = (S, A)$ est le graphe traité, w sa pondération et s le sommet source.

Procedure *Dijkstra(*G, w, s*)*
- Initialisation-PCC(G, s);
- $V \leftarrow \varnothing$;
- $Q \leftarrow S$;
- **tant que** $Q \neq \varnothing$ **faire**
  - $u \leftarrow$ Extrait-Minimum(Q);
  - $V \leftarrow V \cup \{u\}$;
  - **pour chaque** *sommet* $v \in Adj[u]$ **faire**
    - Relaxation(u, v, w);

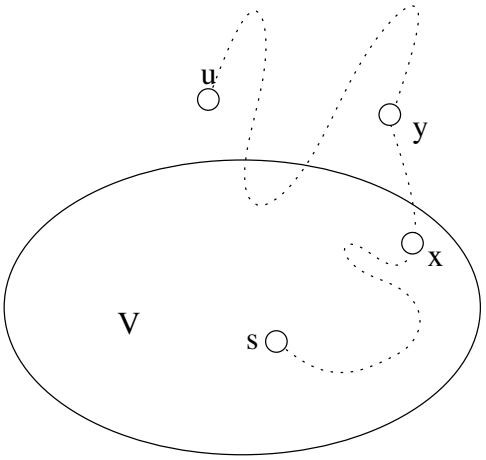

FIGURE 5.3 – Mise à jour des sommets de V

Rappelons que l'on a supposé que u est le premier sommet pour lequel $d[u] \neq \delta(s, u)$. Donc pour x, inséré avant, on a $d[x] = \delta(s, x)$. Comme le chemin de s à u passant par x, y est un plus court chemin, le plus court chemin de s à y aussi (propriété de Bellman sur un plus court chemin) et l'arête (x, y) ayant subi l'application de Relaxation après cette insertion, on a $d[y] = \delta(s, y)$

y apparaît donc avant u sur un plus court chemin de s à u et les poids étant positifs on a : $d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u]$ mais u et y sont tous deux dans $S - V$ et on a choisit u de d minimum donc $d[u] \leq d[y]$ et donc $d[y] = \delta(s, y) = \delta(s, u) = d[u]$ ce qui est impossible car on a supposé $d[u] \neq \delta(s, u)$.

Quelle est la complexité de l'algorithme ? Il y a $|S|$ appels à Extrait-Minimum. Si la file de priorité est simplement maintenue par un tableau, chaque Extrait-Minimum est en $O(|S|)$, ce qui donne un $O(|S|^2)$. Chaque sommet u de S est inséré dans V une fois donc chaque arc de $Adj[u]$ est examiné une fois. On a $|A|$ arcs dans les listes d'adjacence et Relaxation est en $O(1)$ donc on aboutit à un total en $O(|S|^2 + |A|) = O(|S|^2)$.

Si le graphe est peu dense, on peut améliorer ce résultat en utilisant un tas binaire pour représenter Q. L'algorithme résultant est habituellement ap-

pelé algorithme de Dijkstra modifié. Chaque Extrait-Minimum est en $O(\log(|S|))$ et il a $|S|$ appels à Extrait-Minimum. La construction initiale du tas est en $O(|S|)$. L'éventuelle abaissement d'une priorité par Relaxation est accompli en $O(\log(|S|))$ et se fait au plus $O(|A|)$ fois. Au total, on aboutit à une complexité en $O((|S|+|A|).\log(|S|)) = O(|A|.\log(|S|))$ pour un graphe où, par exemple, tous les sommets sont accessibles depuis la source.

On peut encore améliorer cette complexité et atteindre un $O(|S|.\log(|S|) + |A|)$ en utilisant un Tas de Fibonacci mais le jeu en vaut rarement la chandelle.

Exercice : appliquer l'algorithme de Dijkstra au graphe de la figure 5.1 en détaillant chaque étape.

5.3 Cas des graphes orientés sans circuits

Le cas particulier des GOSC mérite d'être traité. Dans ce cas, il est certain qu'aucun circuit absorbant n'existe, même si les poids sont négatifs. D'autre part, il va être possible de traiter le problème plus efficacement car on a va pouvoir utiliser un tri topologique pour ordonner les sommets et ainsi appliquer l'opération Relaxation le plus judicieusement possible.

L'algorithme (parfois appelé algorithme de Bellman, ou de Bellman-Kalaba) est incarné par la fonction GOSC-PlusCourt. Il s'applique sur des GOSC avec pondération quelconque (des poids peuvent être négatifs).

Algorithme 10 : G est le graphe, w sa pondération et s le sommet source.

```
Procedure GOSC-PlusCourt(G, w, s)
    Tri-Topologique(G);
    Initialisation-PCC(G, s);
    pour chaque sommet u de G pris dans l'ordre du tri topologique
    faire
        pour chaque sommet v ∈ Adj[u] faire
            Relaxation(u, v, w,);
```

Dans un GOSC, un chemin qui mène de s à u ne peut pas passer par un sommet qui suit u dans l'ordre de tri topologique. Il suffit alors d'appliquer la propriété de Bellman : il suffit de connaître les plus courts chemins de s aux sommets qui précèdent u dans l'ordre topologique puis d'utiliser la fonction Relaxation. L'algorithme ainsi défini est en $\theta(|S| + |A|)$ et est donc linéaire dans la taille de l'instance.

Exercice : on considère le GOSC de la figure 5.4 déjà trié topologiquement (de gauche à droite). Appliquer l'algorithme GOSC-PlusCourt à ce graphe en illustrant à chaque étape l'arbre des prédécesseurs et la valeurs des estimations.

Exercice : Dans un modèle de chaîne de Markov cachées, construire un graphe tel que la recherche d'un plus court chemin dans ce graphe résolve le problème de décodage par l'algorithme de Viterbi.

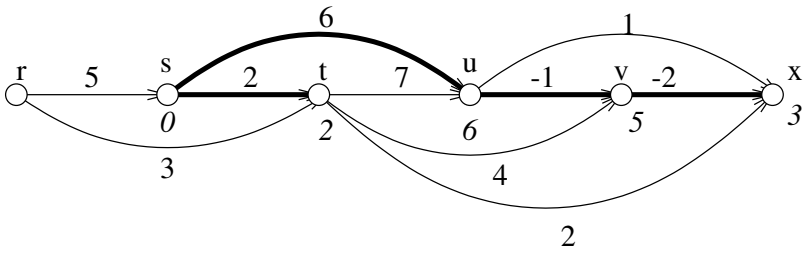

FIGURE 5.4 – Un GOSC, un arbre des plus courts chemins et les $\delta(s,u)$ en italique.

5.4 Problèmes de plus long chemin et ordonnancement

L'algorithme précédent permet, dans le cas de GOSC, de calculer les plus long chemins entre s et les autres sommets. Il suffit de changer le signe des pondérations w et de calculer les plus courts chemins. Il faut noter que pour un graphe quelconque, la détermination d'un plus long chemin simple entre deux sommets défini un problème de décision NP-complet.

La détermination des plus longs chemins dans un GOSC a une application importante : la gestion de projet ou ordonnancement simple (sans ressource, sans date de disponibilité ou de livraison). Dans un problème d'ordonnancement simple, on considère un ensemble de tâches dont on connaît la durée d'exécution *a priori*. Les tâches peuvent, comme dans le cas de l'exemple de l'algorithme Tri-Topologique, être reliées par une relation de dépendance : une tache i ne peut pas commencer avant que la tâche j soit finie. Il ne peut pas exister de circuit dans le graphe des dépendances car cela voudrait dire qu'une tâche doit être finie (transitivement) avant elle-même.

Deux représentation très proches, la représentation potentiel-tâches (proposée par des français) et la représentation PERT (*Projet Evaluation and Review Technique*, américaine et antérieure d'une année à la précédente) existent. Nous présenterons la première non pas par chauvinisme mais du fait de sa simplicité.

Le graphe considéré aura un sommet pour chaque tâche plus deux sommets représentant des tâches fictives notées d (début) et f (fin). Les arcs du graphe sont les contraintes de dépendances (ou précédences), pondérés par la durée de la tâche à l'extrémité initiale de l'arc. On ajoute enfin des arcs $d \rightarrow s$, de poids nul, pour tout sommet s sans prédécesseur et des arcs $s \rightarrow f$ pour tout sommet s sans successeur.

Considérons l'exemple suivant : *Pour rénover une maison, il est prévu de refaire l'installation électrique (3 jours), de réaménager (5 jours) et de carreler (2 jours) la salle de bains , de refaire le parquet de la salle de séjour (6 jours) et de repeindre les chambres (3 jours). La peinture et le carrelage ne pourront être faits qu'après la réfection de l'installation*

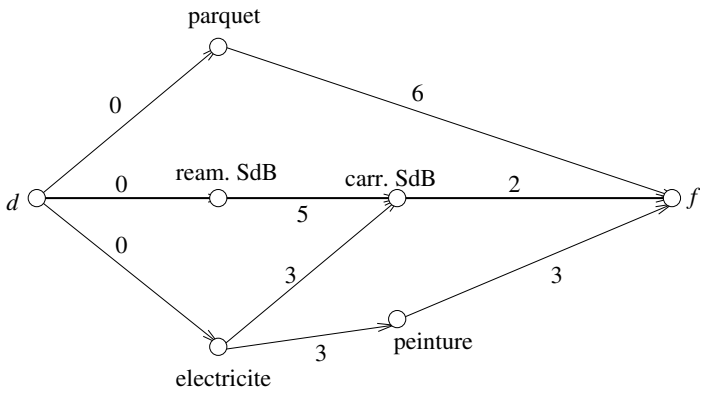

FIGURE 5.5 – Graphe potentiel/tâches et chemin critique

électrique.

Sa représentation sous forme de graphe apparaît dans la figure 5.5. L'application de l'algorithme GOSC-PlusCourt permet de calculer la *date de début au plus tôt* de chaque tâche : il s'agit de la longueur d'un plus long chemin de la source d à chacun des sommets. La durée minimale des travaux est obtenue en calculant la longueur du plus long chemin de d à f, ici 7 jours. Les plus longs chemins de d à f sont dits *critiques*. En effet, tout allongement de la durée d'une tâche sur un tel chemin va déplacer la fin du projet dans le temps. On peut enfin calculer la *date de début au plus tard* de chaque tâche en transposant le graphe (on retourne tous les arcs) et en calculant la longueur d'un plus long chemin de f à chacune des tâches et en soustrayant ce nombre de la longueur d'un chemin critique. On appelle *marge totale* d'une tâche la différence entre sa *date de début au plus tôt* et sa *date de début au plus tard* et donne le retard possible dans une tâche sans que la date finale de réalisation soit affectée.

La tableau suivant donne ces valeurs pour le graphe de la figure 5.5.

Tâches	début au plus tôt	début au plus tard	marge totale
parquet	0	1	1
réam. SdB	0	0	0
carr. SdB	5	5	0
electricité	0	1	1
peinture	3	4	1

On représente fréquemment le résultat de cette analyse par un diagramme dit « de Gantt », visible dans à la figure 5.6.

Les problèmes d'ordonnancement plus complexes, prenant en compte la disponibilité de ressources, éventuellement non partageables, de dates de disponibilité des produits et de dates de livraisons sont beaucoup plus difficiles et définissent habituellement des problèmes de décision NP-complets.

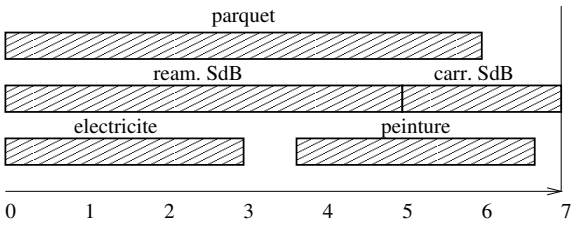

FIGURE 5.6 – Diagramme de Gantt associé.

5.5 Distance minimum entre toutes les paires

Elles peuvent se calculer en utilisant l'algorithme de Warshall vu pour calculer les fermetures transitives. Il suffit de remplacer l'algèbre $(\wedge, \vee)$ par $(\min, +)$.

Chapitre 6

Autres problèmes bien résolus

Nous présenterons rapidement dans cette section deux autres problèmes sur les graphes (pondérés) qui sont solubles en temps polynomial malgré une grande complexité apparente. Il s'agit des problème de couplage maximum (ou affectation) et des problèmes de flots maximum.

6.1 Couplages

On doit affecter un certain nombre de cours à un certains nombre d'enseignants. Chaque enseignant est qualifié dans certaines disciplines. On désire attribuer un cours à chaque enseignant de tel façon qu'un même enseignement ne soit pas affecté à deux professeurs différents. S'il n'est possible de satisfaire tous les enseignants, on veut satisfaire le maximum d'entre eux. Comment faire ? On va construire le graphe biparti suivant : chaque cours et chaque enseignant est représenté par un sommet. Un arc relie un cours et un enseignant si cet enseignant peut faire le cours correspondant.

Définition 18 *Étant donné un graphe non orienté* $G = (S, A)$*, un couplage sur* G *est formé d'un ensemble d'arêtes* C *tel qu'aucune paire d'arêtes de* C *ne partage un sommet commun. La taille du couplage est le nombre d'arêtes de* C*. Le couplage est* parfait *si tous les sommets du graphe sont représentés dans le couplage.*

Une affectation optimale des cours aux enseignants est alors obtenue par la détermination d'un *couplage* maximum dans G (cf. figure 6.1). Un tel couplage peut-être déterminé en $O(|S|.|A|)$ par l'algorithme des chaînes améliorantes (algorithme de Ford-Fulkerson). Cet algorithme a été ensuite amélioré par Hopcroft et Karp pour aboutir à un algorithme en $O(\sqrt{|S|}.|E|)$. D'autres algorithmes pouvant résoudre le problème sur des graphes non nécessairement bipartis existent. On se reportera à la littérature [GM79, AHU87, CLR90].

Pour décrire et comprendre l'algorithme des chaînes améliorantes, un certain nombre de notions sont nécessaires. Soit $G = (S, A)$ un graphe non orienté et

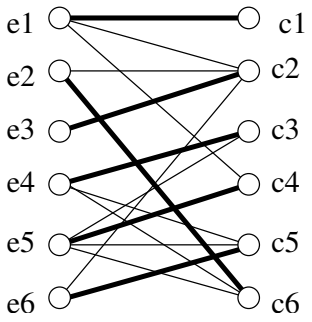

FIGURE 6.1 – Un graphe biparti et un couplage maximum (parfait)

soit $C \subset A$ un couplage de G. On dira d'un sommet $s \in S$ qu'il est *saturé* si c'est l'extrémité d'une arête de C. Sinon, il est *insaturé*. Une chaîne dans laquelle une arête sur deux est dans C et les autres dans $A - C$ est dite *alternée*. C'est par exemple le cas de la chaîne $\langle c_1, e_1, c_4, e_5, c_6 \rangle$ dans la figure 6.1. Une chaîne alternée est dite *améliorante* si elle relie deux sommets insaturés. On peut noter qu'une chaîne améliorante doit avoir une longueur impaire et que ses deux arêtes extrêmes sont nécessairement prises dans $A - C$.

L'observation cruciale pour le fonctionnement de l'algorithme des chaînes améliorantes consiste à remarquer que si l'on a un couplage C et une chaîne améliorante C_A, il est possible de construire un couplage $C' = C \oplus C_A$ obtenu en supprimant dans C toutes les arêtes qui sont aussi dans C_A et en rajoutant dans C toutes les arêtes qui sont dans C_A et qui ne sont pas dans C (le symbole $\oplus$ représente ici un "ou exclusif"). Le couplage C' contient donc les arêtes qui sont présentes dans C ou dans C_A mais pas dans les deux simultanément. C' est bien un couplage car toute arête ajoutée dans C connecte soit un sommet insaturé (extrémités de la chaîne) soit un sommet saturé mais dont l'arête de C qui le saturait est enlevé de C'. De plus, la chaîne ayant une longueur impaire et commençant par une arête qui n'est pas dans C, le nombre d'arêtes de $C - A$ qu'elle contient est supérieur d'une unité aux nombre d'arêtes de C qu'elle contient. Le couplage C' contient donc une arête de plus que le couplage C. On sait donc qu'un couplage maximum n'admet pas de chaînes augmentantes. Il reste à montrer la réciproque.

Soit le graphe $G = (S, A)$ et C et D deux couplages de G tels que $|C| < |D|$. On considère le graphe $G' = (S, C \oplus D)$. Comme C et D sont deux couplages, chaque sommet de S est l'extrémité d'au plus une arête de C et d'au plus une arête de D. Une composante connexe de G' forme donc une chaîne simple (éventuellement un cycle) dont une arête sur deux est dans C et les autres dans D. Comme $|C| < |D|$, $C \oplus D$ contient plus d'arêtes de D que de C. Les cycles contenant autant d'arêtes de C que de D, G' a forcément une chaîne qui n'est pas un cycle et qui contient plus d'arêtes de D que de C : c'est une chaîne améliorante.

Il reste maintenant à trouver une façon efficace d'identifier les chaînes améliorantes. On se limite ici au cas des graphes bipartis. Cela va pouvoir se faire par construc-

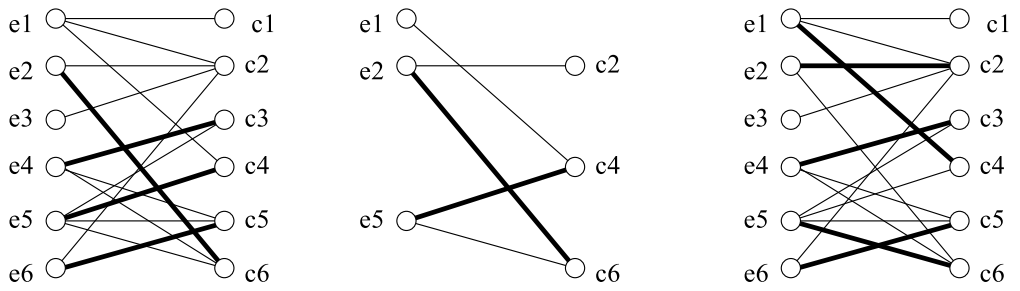

FIGURE 6.2 – Un graphe biparti et un couplage, une chaîne améliorante et le couplage amélioré

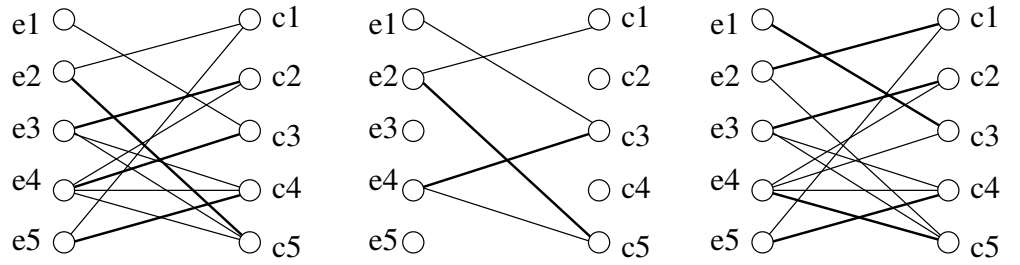

FIGURE 6.3 – Un graphe biparti et un couplage, le graphe des chaînes améliorantes et un couplage amélioré.

tion du *graphe des chaînes améliorantes*. Ce graphe se construit itérativement par niveau. Le niveau 0 est composé de tous les sommets insaturés de S_1 (dans un graphe biparti, $S = S_1 \cup S_2$ et aucune arête ne relie deux sommets de S_1 ou deux sommets de S_2). Au niveau i impair, on insère les sommets adjacents aux sommets insérés au niveau $i-1$ par une arête qui n'est pas dans le couplage courant, ainsi que cette arête. Au niveau i pair, on insère chaque sommet adjacent à un des sommets inséré au niveau $i-1$ par le biais d'une arête du couplage, ainsi que cette arête. On continue à construire ce graphe jusqu'à l'insertion à un niveau impair d'un sommet s insaturé (ou jusqu'à épuisement des sommets, chaque sommet n'étant inséré qu'une fois). Dans le premier cas, tout chemin de s au niveau 0 est une chaîne améliorante. Sinon, aucune chaîne améliorante n'existe. Si l'on utilise une liste d'adjacence, ce graphe peut être construit en $O(|A|)$ en utilisant une variante de la recherche en largeur d'abord. Partant d'un couplage vide, il faudra au plus $\frac{|S|}{2}$ chaînes augmentantes pour atteindre un couplage maximum et l'algorithme global est donc en $O(|A|.|S|)$.

La figure 6.3 présente un couplage et le graphe des chaînes améliorantes correspondant. Une chaîne améliorante est identifiée au niveau 7.Il s'agit de la chaîne $\langle e_1, c_3, e_4, c_2, e_3, c_4, e_5, c_1 \rangle$. L'utilisation de cette chaîne améliorante mène au couplage de droite. Sur ce couplage, il n'y a plus de chaîne améliorante (de façon triviale, le couplage étant parfait).

Le véritable problème d'affectation apparaît si les arêtes reçoivent des poids non négatifs représentant un coût. On cherche alors un couplage parfait de poids minimum. On pourra par exemple utiliser l'algorithme dit *hongrois* [GM79].

6.2 Graphe de transport et flot

On appelle réseau, ou graphe de transport, un graphe orienté $G = (S, A)$ muni pour chaque arc d'une pondération $c(u, v)$ appelée dans ce cas capacité de l'arc (u, v), toujours positive et de deux sommets particuliers s et t (respectivement source et puits). Pour toute paire de sommets (u, v), si $(u, v) \notin A$, on étend $c(u, v) = 0$. Sans perte de généralité, on suppose que le réseau est antisymétrique (si $(u, v) \in A$, alors $(v, u) \notin A$. Si ce n'est pas le cas, on peut ajouter un sommet intermédiaire x sur un des arcs, par exemple (u, v), remplaçant (u, v) par (u, x) et (x, v) de capacité $c(u, v)$).

Un flot net f sur ce graphe est une pondération positive f des arcs du graphe qui satisfait :

- **Respect des capacités** : $f(u, v) \leq c(u, v)$
- **Symétrie** : pour tout $u, v \in S$, $f(u, v) = -f(v, u)$
- **Conservation des flots** : pour tous les sommets $u \in S - \{s, t\}$, $\sum_{v \in V} f(u, v) = 0$ (tout ce qui rentre dans un sommet en sort).

On notera que toutes ces contraintes sont des contraintes linéaires, avec des coefficients qui appartiennent à $\{0, 1, -1\}$. Un flot représente par exemple un courant électrique dans un réseau et la loi de conservation des flots s'appelle loi de Kirchoff. La notion de flot présenté ici est souvent appelé "flot net". On note G^f le graphe d'origine pondéré par un flot f donné.

La valeur d'un flot f, notée $|f|$ est déterminée par tout ce que l'on arrive à « injecter » dans le sommet source (et qui est égal à ce que l'on récupère dans le sommet puits) : il s'agit donc de la somme des flots que parcourt la source $\sum_{v \in S} f(s, v)$.

Si l'on définit le flot positif entrant dans le sommet v par :

$$\sum_{u \in S, f(u,v) > 0} f(u, v)$$

et si l'on définit le flot positif sortant de v de façon symétrique, une autre interprétation de la contrainte de conservation des flots est que les flots positifs qui rentrent et qui sortent à un sommet sont identiques.

Pour alléger nos notations, nous utilisons par la suite une forme de sommation implicite en étendant la fonction de flot net f à des ensembles de sommets. pour tout ensembles X et Y de sommets, on notera :

$$f(X, Y) = \sum_{x \in X, y \in Y} f(x, y)$$

Avec cette notation, l'équation de conservation des flots du sommet u s'écrit simplement $f(u, S) = 0$ et la valeur du flot $|f| = f(s, S - s)$. Cette fonction étendue vérifie différentes propriétés simples :

- pour tout $X \subseteq S$, on a $f(X, X) = 0$.
- pour tout $X, Y \subseteq S$, on a $f(X, Y) = -f(Y, X)$.
- pour $X, Y, Z \subseteq S$ et $X \cap Y = \varnothing$, alors
  $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$
  $f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$

Exercice : démontrer ces propriétés et montrer, en les utilisant, que la valeur du flot $|f| = f(S, t)$.

6.2.1 Coupe d'un réseau

Définition 19 *Une coupe* (E, T) *d'un réseau de transport* $G = (S, A)$ *est une partition de* S *en* E *et* $T = S - E$ *telle que* $s \in E$ *et* $t \in T$. *On définit la capacité* $c(E, T)$ *de la coupe comme la somme des capacités des arcs* (u, v) *pour* $u \in E$ *et* $v \in T$.

Théorème 2 *Pour toute coupe* (E, T) *et tout flot* f, *la valeur* $|f|$ *du flot est majorée par la capacité de la coupe* $c(E, T)$.

Preuve : Montrons d'abord que $f(E, T) = |f|$ par récurrence. Si $|E| = 1$, c'est immédiat. Par récurrence, soit e un élément pris dans T et mis dans E (qui ne peut être ni t ni s). On a $f(E \cup e, T - e) = f(E, T) - f(E, e) + f(e, T) = f(E, T) + f(e, S) = f(E, T)$. Ce qui démontre la récurrence.

Le théorème découle du fait que les flots sont toujours inférieurs aux capacités.

6.2.2 Flot maximum

Le problème le plus souvent considéré est de trouver un flot de valeur maximum (transporter un maximum de choses en utilisant le réseau muni de ses capacités de transport). Le problème est suffisant général pour pouvoir accommoder des sources et des puits multiples et l'on peut aussi mettre des capacités sur les sommets (il faut modifier le graphe en ajoutant des sommets et des arcs intermédiaires, de capacité bien choisies).

Exemple : une entreprise a une usine de grues au sommet s et a un entrepôt au sommet t. Le transport des grues qu'elle fabrique nécessitent l'utilisation de convois exceptionnels. Un ensemble de routes existent entre les sommets s et t chaque route permettant un certain nombre de transports exceptionnels par mois. L'entreprise aimerait ajuster sa production mensuelle de façon a produire exactement le maximum d'objets qu'elle peut transporter dans son entrepôt chaque mois.

L'utilisation de flots nets rend implicite le problème dit "d'annulation des flots". Si l'on considère les sommets v_1 et v_2 du graphe dans plusieurs situations. On peut transporter 8 grues de v_1 vers v_2. Si à ce transport s'ajoute le transport de 3 grues de v_2 vers v_1, par annulation des flots, nous utiliserons un flot net de 5 grues de v_1 vers v_2 qui donnera effectivement un flot net de

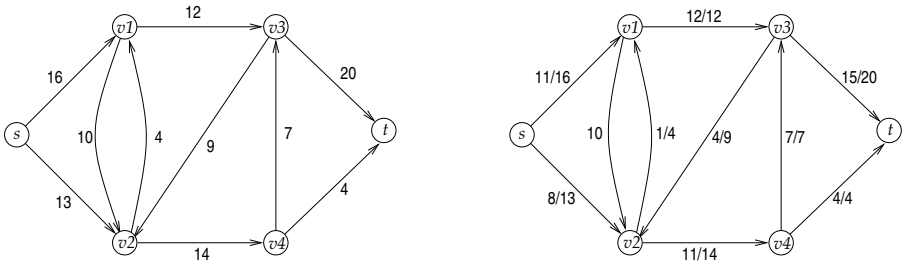

FIGURE 6.4 – Un problème de flots et un flot possible (chaque arc porte $c(u,v)/f(u,v)$. Seuls les flots strictement positifs sont indiqués.

-5 grues dans la direction opposée. L'utilisation des flots nets permet de mettre en évidence les transports de grues nécessaires : ce sont les flots nets qui sont strictement positifs. Un flot nul ou négatif ne nécessite aucun transport. Cette forme canonique des flots n'affecte pas le respect des capacités (les flots diminuent) et préserve la conservation des flots.

L'algorithme le plus connu pour résoudre ce problème est l'algorithme de Ford-Fulkerson (1962), en $O(|A|.f^*)$ où f^* est la valeur du flot maximum et ce pour des capacités entières [GM79, CLR90] [1].

C'est un algorithme itératif qui va continuellement améliorer un flot initial (par exemple le flot nul $f(u,v) = 0$) en augmentant sa valeur $|f|$.

Définition 20 *Étant donné un réseau* $G = (S, A)$ *de capacité* c *et un flot* f *sur* G*, on appelle capacité résiduelle* $c_f(u,v) = c(u,v) - f(u,v)$*. La capacité résiduelle est toujours positive ou nulle, par définition.*

Ainsi, si $c(u,v) = 10$ et $f(u,v) = 2$, la capacité résiduelle est de 8, indiquant la marge d'augmentation du flot sur cet arc. Si $f(u,v)$ est négatif, la capacité résiduelle est plus grande que la capacité, indiquant que l'on peut augmenter le flot sur (u,v) en profitant de la capacité disponible mais aussi en diminuant le flot opposé selon (v,u).

Le graphe des capacités résiduelles G_f est pondéré par les capacités résiduelles et présente un arc entre chaque paire de sommets pour laquelle la capacité résiduelle est strictement positive. Dans ce graphe, un chemin de s à t est appelé chemin augmentant. Il indique comment exploiter ces capacités résiduelles : pour chaque arc parcouru par le chemin, on peut augmenter le flot dans le sens

1. C'est une version plus générale de l'algorithme que nous venons de voir pour les problèmes de couplage dans les graphes bipartis. En effet, le problème de couplage maximum dans un graphe biparti est un cas particulier de problème de flots à valeurs entières. Il suffit de rajouter un sommet source fictif, relié à tous les sommets de S_1 par des arcs de capacité unitaire, de rajouter un sommet puit fictif, relié à tous les sommets de S_2 par de arcs de capacité unitaire et de transformer toutes les arêtes du graphe biparti en arc orientés de S_1 vers S_2 et de capacité unitaire. Un flot maximum à valeurs entières représente alors un couplage maximum. Dans ce cas f^* vaut au plus $\frac{|S|}{2}$ et on aboutit bien à une complexité $O(|A|.|S|)$.

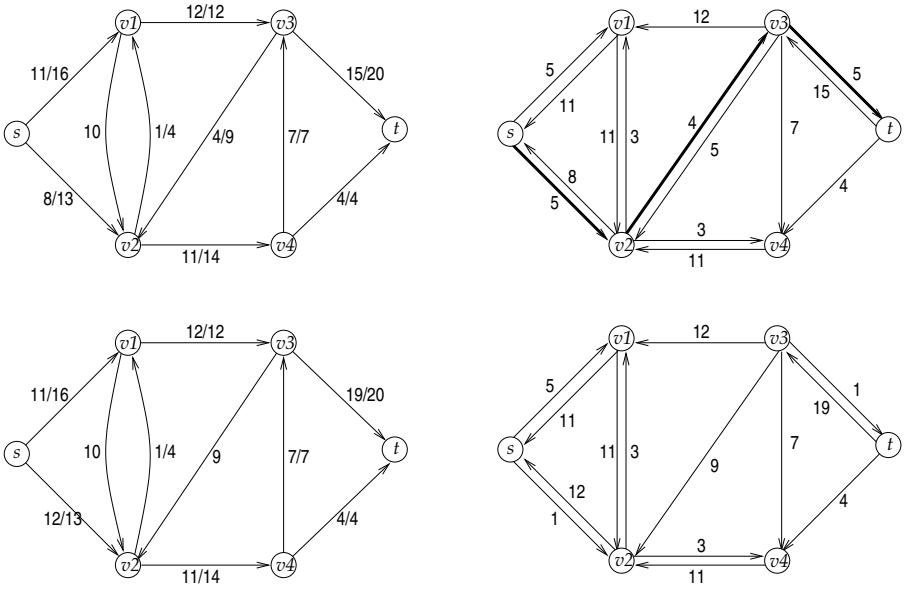

FIGURE 6.5 – à partir du flot de la figure 6.4, le graphe des capacités résiduelles, un chemin augmentant, le graphe résultat et le graphe des capacités résiduelles

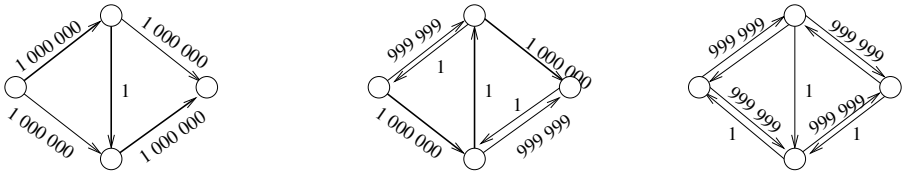

FIGURE 6.6 – Un réseau difficile pour Ford-Fulkerson : graphe et capacités résiduelles successives

indiqué par le chemin et d'une quantité égale à la plus petite capacité résiduelle sur le chemin.

Comme dans le cas des couplages, on peut montrer que si aucun chemin augmentant n'existe, alors le flot est maximum. Chaque augmentation du flot améliore le flot de 1 au moins. Chaque recherche de chemin (accessibilité) peut se faire par un algorithme d'exploration du graphe (en largeur ou en profondeur d'abord) qui est en $O(|A|)$. Chaque augmentation incrémente le flot. On obtient ainsi la complexité. Attention, cet algorithme peut être très inefficace et peut converger très lentement. Pour des capacités entières on pourra considérer l'exemple de la figure 6.6 ou FordFulkerson atteint sa complexité maximale.

Le choix de déterminer un chemin augmentant par un algorithme en largeur d'abord permet de garantir que le chemin augmentant trouvé est le plus court (en nombre d'arêtes). L'algorithme ainsi défini est appelé algorithme de Edmonds-Karp[?]. On va montrer que le nombre d'augmentations dans ce cas est en $O(|S|.|A|)$ ce qui donne une complexité en $O(|S|.|A|^2)$. Pour cela, on montre d'abord que :

```
;; On note c_f(u,v) la capacité résiduelle pour le flot f.;
Procedure FordFulkerson(G, s, t)
    pour chaque arc (u,v) ∈ A faire
        f[u,v] ← 0;
        f[v,u] ← 0;
    tant que ∃ un chemin p de s vers t dans le graphe résiduel G_f
    faire
        a ← min{c_f(u,v) : (u,v) ∈ p};
        pour chaque arc (u,v) de p faire
            f[u,v] ← f[u,v] + a;
            f[v,u] ← −f[u,v];
```

Théorème 3 *Lors de l'exécution de l'algorithme d'Edmonds-Karp, pour tous les sommets $v \in S - \{s,t\}$, la distance de plus court chemin $\delta_f(s,v)$ dans le réseau résiduel G_f augmente de façon monotone avec chaque augmentation de flot.*

Preuve : On raisonne par l'absurde : on suppose que, pour un certain sommet $v \in S - s,t$, il existe une augmentation de flot qui provoque la diminution de la distance de plus court chemin entre s et v. Soit f le flot juste avant la première augmentation qui diminue une certaine distance de plus court chemin, et soit f' le flot juste après. Soit v le sommet ayant le $\delta_{f'}(s,v)$ minimal dont la distance a été diminuée par l'augmentation, de sorte que $\delta_{f'}(s,v) < \delta_f(s,v)$. Soit $p' = s \rightsquigarrow u \rightarrow v$, un plus court chemin de s à v dans $G_{f'}$, de sorte que $(u,v) \in A_{f'}$ et

$$\delta_{f'}(s,u) = \delta_{f'}(s,v) - 1$$

Compte tenu de la façon dont on a choisi v, on sait que l'étiquette distance du sommet u n'a pas diminué, c'est-à-dire que

$$\delta_{f'}(s,u) \geq \delta_f(s,u)$$

Alors $(u,v) \notin A_f$. En effet, si l'on avait $(u,v) \in A_f$, alors on aurait aussi

$$\begin{aligned} \delta_f(s,v) &\leq \delta_f(s,u) + 1 && (6.1)\\ &\leq \delta_{f'}(s,u) + 1 && (6.2)\\ &= \delta_{f'}(s,v) && (6.3)\\ & && (6.4) \end{aligned}$$

(la première inégalité découle de l'inégalité triangulaire).

Ceci contredit notre hypothèse que $\delta_{f'}(s,v) < \delta_f(s,v)$. Comment peut-on avoir $(u,v) \notin A_f$ et $(u,v) \in A_{f'}$? L'augmentation doit avoir accru le flot entre

v et u. L'algorithme d'Edmonds-Karp augmente toujours le flot sur des plus courts chemins, et donc le plus court chemin de s à u dans G_f a (v,u) pour dernier arc. Par conséquent, $\delta_f(s,v) = \delta_f(s,u) - 1 \leq \delta_f(s,u) - 1 = \delta_{f'}(s,v) - 2$, ce qui contredit notre hypothèse que $\delta_{f'}(s,v) < \delta_f(s,v)$. On en conclut que notre hypothèse selon laquelle il existe un tel sommet v est erronée.

Théorème 4 *L'algorithme de Edmonds-Karp effectue un nombre total d'augmentations en* $O(|S||A|)$.

Preuve : un arc (u,v) du réseau résiduel G_f sera dit *critique* sur un chemin améliorant p si la capacité résiduelle de p est la capacité résiduelle de (u,v). Après avoir augmenté le flot le long d'un chemin améliorant, tout arc critique du chemin disparaît du réseau résiduel. De plus, un arc au moins est critique dans chaque chemin améliorant. Nous allons montrer que chacun des $|A|$ arcs peut devenir critique au plus $|S|/2 - 1$ fois.

Soit u et v deux sommets de S reliés par un arc de A. Puisque les chemins améliorants sont des plus courts chemins, quand (u,v) est critique pour la première fois, on a $\delta_f(s,v) = \delta_f(s,u) + 1$.

Après augmentation du flot, l'arc (u,v) disparaît du réseau résiduel. Il ne pourra pas réapparaître sur un autre chemin améliorant tant que le flux de u à v n'aura pas diminué. Ceci n'arrive que si (v,u) apparaît sur un chemin améliorant. Si f' est le flot de G au moment de cet événement, on a $\delta_{f'}(s,u) = \delta_{f'}(s,v) + 1$. Comme $\delta_f(s,v) \leq \delta_f(s,v)$ d'après le lemme précédent, on a :

$$\begin{aligned} \delta_{f'}(s,u) &= \delta_{f'}(s,v) + 1 && (6.5)\\ &\geq \delta_f(s,v) + 1 && (6.6)\\ &= \delta_f(s,u) + 2 && (6.7)\\ & && (6.8) \end{aligned}$$

En conséquence, entre le moment où (u,v) devient critique et celui où il devient à nouveau critique, la distance entre u et la source augmente au moins de 2. La distance entre u et la source est initialement au moins égale à 0. Les sommets intermédiaires d'un plus court chemin de s à u ne peuvent pas contenir s, u ou t (car (u,v) sur le chemin critique entraîne que $u \neq t$). Donc, jusqu'à ce que u devienne éventuellement inaccessible à partir de la source, sa distance est au plus égale à $|S| - 2$. Donc, (u,v) peut devenir critique au plus $(|S| - 2)/2 = |S|/2 - 1$ fois. Puisqu'il existe $O(|A|)$ paires de sommets pouvant être reliés par un arc dans un graphe résiduel, le nombre total d'arcs critiques pendant toute l'exécution de l'algorithme d'Edmonds-Karp est $O(|S||A|)$. Chaque chemin améliorant comporte au moins un arc critique, et le théorème en découle.

6.2.3 Améliorations

En fait, le premier algorithme polynomial pour ce problème est celui de [?]. L'histoire de cet algorithme publié dans une revue russe, et donc peu connu à

l'époque dans le monde occidental, a été décrite en détail dans [?]. Cet algorithme choisit de saturer tous les plus courts chemins à chaque étape et résout le problème en $O(|S|^2|A|)$.

Avec le temps, de nombreux progrès ont été réalisés pour résoudre ce problème de flot maximum avec une meilleure complexité asymptotique. On citera l'algorithme de Goldberg et Tarjan (1986) permet de résoudre ce problème de flot maximum en $O(|S|.|A|.\log(\frac{|S|^2}{|A|}))$ [GM79, CLR90]. L'algorithme de Dimic, implémenté à l'aide d'une structure de donnée spécialisée appelée "arbre dynamique" [?] améliore la complexité en $O(|S||A|.\log(|S|)$. Plus récemment, l'algorithme de King, Rao et Tarjan (1994) est en $O(|S||A|\log_{|A|/|S|\log|S|}|S|)$.

6.2.4 Théorème de Ford-Fulkerson

On a vu que la valeur de tout flot est majoré par la valeur d'une coupe quelconque. On montre en fait que ce majorant est exact dans le sens où le la valeur d'un flot maximum est précisément égal à la capacité d'une coupe minimum.

Théorème 5 *Si* f *est un flot dans un réseau de transport* $G = (S, A)$ *de source* s *et de puits* t. *les conditions suivantes sont équivalentes.*

1. f *est un flot maximum de* G
2. *le réseau résiduel* G_f *ne contient aucun chemin améliorant*
3. $|f| = c(E, T)$ *pour une certaine coupe* (E, T) *de* G

Preuve : Si f est un flot maximum et que G_f contient un chemin améliorant p alors on peut améliorer le flot f ce qui contredit l'hypothèse de flot maximum.

Si G_f ne contient pas de chemin améliorant, on définit

$$E = \{v \in S \ : \text{ il existe un chemin de } s \text{ vers } v \text{ dans } G_f\}$$

et $T = S - E$. La partition (E, T) est une coupe. Pour chaque paire de sommets $(u, v) \in E \times T$, on a $f(u, v) = c(u, v)$ car sinon une capacité résiduelle non nulle existe entre u et v et $v \in E$. On a alors $|f| = f(E, T) = c(E, T)$.

Enfin, on sait que $|f| \leq c(E, T)$. Donc si $|f| = c(E, T)$, f est forcément maximum.

6.2.5 Fermeture de bénéfice maximal

Un certain nombre de tâches $j \in J$ avec un bénéfice associé $b(j)$ sont définies. Le bénéfice peut être négatif (une perte).

Les tâches sont soumises à des contraintes de fermeture : le choix d'une tâche j_1 peut impliquer le choix d'une tâche j_2. On représente cette relation dite de fermeture (ou implication) par un graphe dont les sommets sont les tâches, et où il existe un arc (j_1, j_2) si et seulement si le le choix de j_1 implique le choix

de j_2. L'objectif est de trouver un ensemble de tâches cohérentes (respectant les contraintes) et de bénéfices maximal. Plus formellement :

Déterminer un sous ensemble $U \subset J$ tel que (1) aucun arc ne sorte de U (cohérence) et (2) $\sum_{j \in U} b(j)$ soit maximum.

Exercice : modéliser ce problème par un graphe de transport tel qu'un flot maximum (ou une coupe minimum) dans ce graphe donne une solution optimale du problème.

Solution : on construit un réseau de transport dont les sommets sont les tâches plus un sommet source s et puits t.

- Pour toute tâche j de bénéfice positif, on relie la source s à t par un arc de capacité $b(j)$.
- Pour toute tâche j de bénéfice négatif, on relie le sommet de la tâche j au puits t par un arc de capacité $-b(j)$.
- Si une tâche j_1 implique une tâche j_2, on ajoute l'arc (j_1, j_2) avec une capacité infinie au réseau.

Considérons une coupe (E, T) de capacité finie du réseau (avec $s \in E$). Comme la capacité est finie, E ne coupe aucun arc de capacité infinie et est donc cohérente. Pour tout ensemble de tâches $H \subseteq J$ on note H^+ l'ensembles des tâches de H avec $b(t) \geq 0$ et H^- celles qui sont associées à une perte. La valeur de la coupe est :

$$
\begin{aligned}
c(E,T) &= \sum_{j \in T^+} c(s,j) + \sum_{j \in E^-} c(j,t) && (6.9)\\
&= \sum_{j \in T^+} b(j) - \sum_{j \in E^-} b(j) && (6.10)\\
&= \sum_{j \in J^+} b(j) - \sum_{j \in E^+} b(j) - \sum_{j \in E^-} b(j) && (6.11)\\
&= \sum_{j \in J^+} b(j) - \sum_{j \in E} b(j) && (6.12)\\
& && (6.13)
\end{aligned}
$$

Le premier terme est une constante et minimiser la coupe est équivalent à maximiser le bénéfice de E.

Exercice :

- Une denrée est disponible dans des centres de distribution $A_1, A_2, \ldots, A_p$. La quantité disponible en A_i est notée a_i.
- Des demandes sont effectuées par des clients $B_1, B_2, \ldots, B_q$. La quantité demandée par B_i est notée b_i.
- Le client B_i ne peut être livré que par un sous-ensemble $L_i \subset A_i$ de l'ensemble des centres de distributions.

Déterminer l'approvisionnement maximal de l'ensemble des clients en le modélisant comme un problème de calcul de flot maximum dans un réseau de transport que l'on explicitera.

6.3 Flot de coût minimum

Le problème de flot maximum a été très étudié. De nombreuses versions plus complexes du problèmes (prise en compte de coût de transport, coexistence simultanée de plusieurs flots consommant de la capacité ou "multi-commodity flow problem"...).

Nous nous intéresserons ici au calcul d'un flot de coût minimum dans un graphe de transport où chaque arc (u, v) est pondéré par un coût de transport unitaire noté $w(u, v)$, en sus de sa capacité $c(u, v)$.

Les flots manipulés ne seront plus des flots nets car le coût de transport d'une unité de flot de u vers v est fixé indépendemment du coût de v vers u. On notera un tel flot g. Si les contraintes de capacité sont conservées, on abandonne la contrainte de symétrie et la contrainte de conservation des flots se réécrie :

$$\forall v \neq s, t \quad \sum_{v \in \Gamma^+(u)} g(u, v) = \sum_{v \in \Gamma^-(u)} g(v, u)$$

À ces contraintes, s'ajoute une contrainte de "flot demandé" : $|g| = d$. Sans cette contrainte, le flot trivial nul est sinon une solution optimale. Dans certains cas, on exige que le flot recherché soit de valeur maximale. Dans ce cas, on parle de problème de flot maximum de coût minimum.

Pour un flot brut g sur un réseau de transport $G = (S, A)$ avec coûts, le coût total de g est défini par :

$$\sum_{(u,v) \in A} g(u, v).w(u, v)$$

une fonction linéaire du flot.

De nombreux algorithmes existent pour résoudre ces problèmes (dont certains dérivés de la programmation linéaire). Nous avons choisi de présenter ici l'algorithme de Busaker et Gowen ou "Successive shortest path"algorithm car il peut être vu comme un simple généralisation de l'algorithme de Ford-Fulkerson.

6.3.1 Graphe des capacités résiduelles

Si un arc (u, v) supporte déjà un peu de flot, il est possible de diminuer ce flot, cette diminution entraînera une baisse du coût de $w(u, v)$ par unité de flot. Pour représenter ce phénomène, le graphe des capacités résiduelles est donc étendu comme suit :

Définition 21 *Étant donné un réseau* $G = (S, A)$ *de capacité* c *et de coût* w *et un flot brut* g *sur* G*, le graphe des capacités résiduelles avec coûts* G_f *a les même sommets que* G*. A tout arc* (u, v) *tel que* $g(u, v) < c(u, v)$ *correspond un arc* (uv) *dans* G_g *ayant une capacité résiduelle de* $c(u, v) -$

$g(u,v)$ *et un coût* $w_g(u,v) = w(u,v)$ *(appelé arc avant). A tout arc* (u,v) *tel que* $g(u,v) > 0$ *correspond un arc* (v,u) *dans* G_g *avec une capacité de* $g(u,v)$ *et un coût* $w_g(u,v) = -w(u,v)$ *(appelé arc inverse).*

Un chemin π de G_g de s à t correspond à un chemin améliorant de G : on peut augmenter le flot de 1 (au moins) sur chaque arc avant parcouru par le chemin π dans G_g et le diminuer de 1 sur chaque arc inverse pour augmenter le flot. L'impact de cette modification sur le coût du flot sera précisément égal à la longueur du chemin.

Théorème 6 *Un flot brut* g *de valeur* |g| *ne peut être de coût minimum parmi tous les flots de valeur* |g| *si il existe un circuit négatif dans* G_g.

Preuve : S'il existe un circuit absorbant dans G_g, il est possible de modifier le flot g selon ce circuit en augmentant g de une unité le long des arcs avant du circuit et en le diminuant de 1 pour les arcs inverses. Globalement, ce circuit ne modifie par la valeur du flot mais affecte son coût, qui décroît autant que le coût du circuit dans G_g. Ceci contredit son optimalité.

Nous ne montrerons pas, mais la réciproque est également vraie (une preuve est accessible par exemple dans [BJG00], page 132).

Ce résultat mène à deux approches algorithmiques possibles :

1. Partir d'un flot du débit désiré et le rendre minimum en supprimant progressivement les circuits négatifs.
2. Construire une série de flots de débits croissants, mais tous de coût minimum pour leur débit (sans circuit négatif).

Des algorithmes existent pour chacune de ces approches. Nous présentons ici l'algorithme de Busacker et Gowen qui utilise la deuxième approche.

6.3.2 Algorithme de Busacker et Gowen

Théorème 7 *A chacune des itérations, le flot en cours est de coût minimum parmi tous ceux de débit* F.

Preuve : elle se fait par récurrence sur la valeur du flot. Le théorème est vrai initialement pour un flot égal à zéro.

Supposons que l'on ait un flot g de coût minimum au début d'une itération et que l'augmentation de flot sur un chemin π de coût minimum conduise à un flot g' non optimal en fin d'itération.

D'après le théorème 6, il existe un circuit de coût négatif ρ dans le graphe $G_{g'}$ qui n'existe pas dans G_g. Ce circuit emprunte donc nécessairement un arc créé par l'augmentation de flot selon π, donc un arc (u,v) qui était un arc (v,u) sur le chemin π (seule façon de faire apparaître un circuit).

Ainsi, dans les figures 6.7(a) et 6.7(b), un arc (u,v) d'un coût de 2 existe à gauche dans le graphe d'écart G_g sans circuit négatif. Supposons une augmentation de flot le long d'un chemin passant par (u,v) sans atteindre la capacité

Algorithme 11 : Algorithme de Busacker et Gowen

Procedure *BusackerGowen(*G, c, w*)*
- $F = 0; K = 0;$
- $g(u, v) = 0 \quad \forall (u, v) \in A;$
- **répéter**
  - Chercher un chemin π de coût minimum z de s à t dans G_g;
  - **si** *un chemin existe* **alors**
    - Calculer δ l'augmentation de flot possible (la capacité minimum);
    - $\delta = \min(\delta, F_{cible} - F);$
    - Augmenter F et le flot g sur les arcs avants de δ unités;
    - Diminuer le flot g sur les arcs inverses de δ unités;
    - $K = K + \delta \times z;$
- **jusqu'à** *aucun chemin n'existe ou* $F = F_{cible}$;

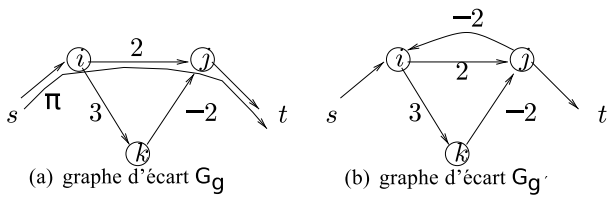

FIGURE 6.7 – Création d'un circuit négatif $c = (i, k, j)$ après augmentation de flot le long de π.

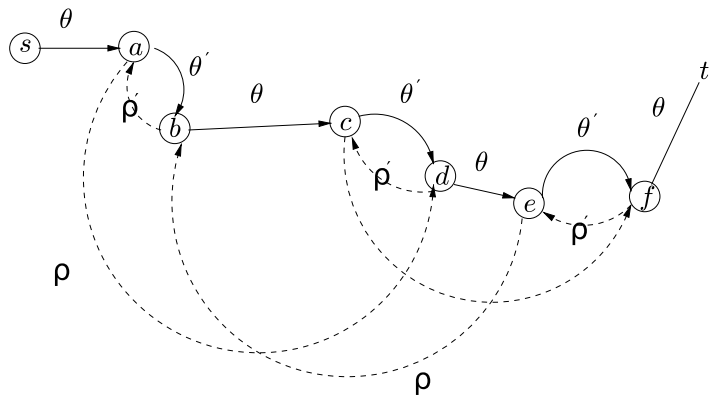

FIGURE 6.8 – Illustration de la preuve du théorème.

de (u, v) (sinon il ne reste que l'arc inverse). Dans le graphe d'écart $G_{g'}$ (voir la figure 6.7(b)), un arc (v, u) de coût -2 apparaît, formant un circuit de coût négatif égal à -1. Soit π' l'ensemble des arcs (u, v) de π qui apparaissent dans ρ sous la forme (v, u), et ρ' le sous-ensemble de ces arcs (v, u) dans ρ (donc π' est sur le chemin π et donc pour le circuit ρ les arcs seront noté ρ'). (Tous les arcs inversés ne créent pas forcément un cycle). ρ' apparaît dans $G_{g'}$ mais n'apparaît pas dans G_g. Ainsi, dans la figure 6.8, nous avons $\pi' = (a, b)$ et $\rho' = (b, a)$.

L'ensemble des arcs $\theta = (\pi - \pi') \cup (\rho - \rho')$ est constitué par un chemin de s à t dans $G_{g'}$ et une union de circuits (voir figure 6.8). Ces circuits existaient également dans G_g, ils ont donc tous un coût positif car il n'existe pas de circuit de coût négatif dans G_g. Par définition la somme du coût des arcs de π' est l'opposée de celle des coûts de ρ' (par construction de $G_{g'}$), par conséquent le coût de θ est la somme du coût de π et de ρ. Le coût de ρ étant négatif, le coût de θ est strictement inférieur au coût de π. En conclusion le chemin θ est un chemin de s à t dans G_g de coût plus petit que le coût de π, ce qui est impossible.

Dans l'exemple donné par la figure 6.8, le chemin θ rencontre successivement s, a, d, e, b, c, f, t. Le chemin ρ rencontre successivement a, d, c, f, e, b, a et le chemin π rencontre successivement s, a, b, c, d, e, f, t.

Complexité : Soit $B = \max_{(u,v)\in A} c(u, v)$ la capacité maximale du réseau. Si B est le maximum des capacités, la capacité de toute coupe est au plus de $p = B|S|$. Nous pouvons avoir p itérations si $\delta = 1$ pour tous les plus courts chemins trouvés. Pour calculer un chemin de coût minimal en présence de coûts négatifs dans le graphe résiduel, il faut utiliser un algorithme de plus court chemin tel que l'algorithme de Bellman-Ford-Moore (le graphe résiduel n'a pas de circuit de coût négatif à chaque étape, mais contient des coûts négatifs), en $O(|S||A|)$. Donc la complexité de l'algorithme de Busacker et Gowen est de $O(|S|^2|A|B)$. L'algorithme est donc seulement pseudo-polynomial.

6.3.3 Chargement de bateaux

Une petite entreprise de transport utilise un bateau qui est capable de transporter r unités d'un bien donné. Le bateau suit une longue route avec plusieurs arrêts entre le début et la fin de son voyage. A chacun de ces ports, il est possible de décharger et de recharger des unités.

Chaque port i a une quantité b_{ij} à transporter vers le port $j > i$ (les ports sont numérotés dans l'ordre de leur visite). Soit f_{ij} le revenu obtenu en transportant un container de i à j.

Le but de l'entreprise est de décider combien de containers il faut charger à chaque port de façon à maximiser son revenu et sans dépasser la capacité du bateau.

Modéliser ce problème sous forme d'un réseau de transport avec coûts.

Solution : Elle est donnée dans le réseau ci-dessous. Chaque sommet $i = v_1, v_2, v_3, v_4, v_5$ représente un port. Il sont reliés entre eux de façon successive par des arcs de capacité r représentant le bateau. Des sommets additionnels v_{ij} représentent la possibilité de transport entre i et j. Ils sont accessibles depuis la source par un arc de capacité b_{ij} et de coût nul. Ces b_{ij} unités potentielles peuvent être transportés de i à j par le transporteur et un arc de capacité infinie et coût $-f_{ij}$ relie v_{ij} à i. En minimisant l'opposé du gain, on maximise le gain. Un second arc relie v_{ij} à j. Il représente l'excès qui est présent et ne peut être transporté en j par l'entreprise. L'arc est de capacité infinie et de coût nul. Enfin, des arcs relie chaque port (sauf v_1) à t. La capacité de l'arc quittant la ville j est égal à la somme des b_{ij} pour $i < j$. Le flot cible est la somme des b_{ij} pour tout j et $i < j$.

Ce flot est atteignable car il est disponible en capacité depuis s jusqu'à t en empruntant les arcs de coût nul. Mais il n'est pas optimal car il possible de transporter certaines unités via les arcs de coût négatif puis les arcs de capacité r. Un flot de coût optimal maximise les gains tout en respectant les capacités du bateau. Les chargements et déchargements effectués à chaque port sont accessibles dans les arcs (v_{ij}, v_i) et (v_i, t) respectivement.

6.4 Problèmes NP-complets sur les graphes

Jusqu'ici nous nous sommes limités à l'étude de problèmes dits "faciles" ou traitables, pour lesquels il existe un algorithme dont la complexité croît comme un polynôme de la taille des entrées.

De nombreux problèmes sur les graphes ne sont pas, hélas, si bien équipés. Ainsi, le problème de k-colorabilité d'un graphe que nous avons considéré au début du cours pour traiter le problème de gestion de feux rouges n'a pas de solution efficace, il est NP-complet.

Les techniques utilisées utilisées pour résoudre ces problèmes sont très nombreuses. Les plus traditionnelles incluent la programmation linéaire en nombres

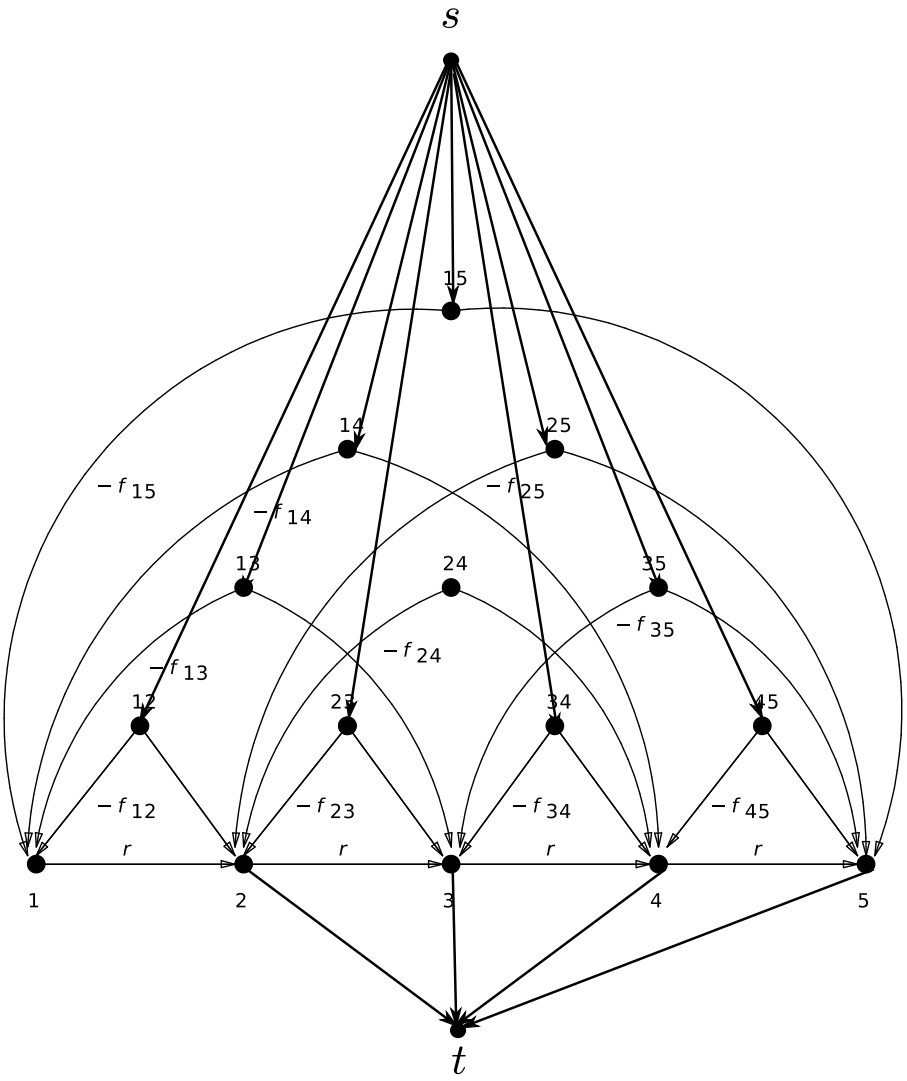

FIGURE 6.9 – Transport par bateau et réseau de transport.

entiers, la programmation par contraintes ou la logique propositionnelle (solveur SAT).

Bibliographie

[AHU87] Alfred Aho, John Hopcroft, and Jeffery Ullman, *Structures de données et algorithmes*, InterÉditions, 1987, Version originale américaine de 1983.

[BJG00] Jorgen Bang-Jensen and Gregory Gutin, *Digraphs : Theory, algorithms and applications*, 1st ed., Springer Verlag, 2000, Disponible gratuitement sur http ://www.cs.rhul.ac.uk/books/dbook/.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to algorithms*, MIT Press, 1990, ISBN : 0-262-03141-8.

[GM79] M. Gondran and M. Minoux, *Graphes et algorithmes*, Eyrolles, 1979, ISSN 0399-4198.