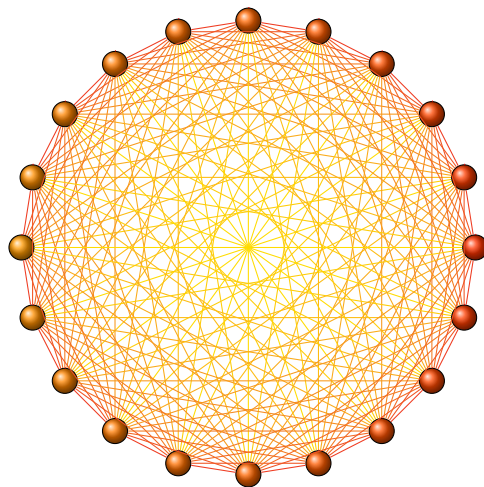


# Introduction à la recherche opérationnelle



R. Moitié  
`roderic.moitie@ensta-bretagne.fr`

C. Osswald  
`christophe.osswald@ensta-bretagne.fr`



## Table des matières

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Recherche opérationnelle . . . . .	5
1.2	Notions de complexité algorithmique . . . . .	7
1.3	Classes de complexité . . . . .	11
<b>2</b>	<b>Optimisation linéaire</b>	<b>19</b>
2.1	Programmes linéaires . . . . .	20
2.2	Algorithme du simplexe . . . . .	26
2.3	Algorithme du simplexe en tableaux . . . . .	29
2.4	Traitement des cas particuliers . . . . .	31
<b>3</b>	<b>Parcours de graphes</b>	<b>37</b>
3.1	Généralités . . . . .	37
3.2	Parcours d'un graphe . . . . .	44
3.3	Plus court chemin dans un graphe . . . . .	48
3.4	Ordonnancement . . . . .	56
<b>4</b>	<b>Graphes non orientés</b>	<b>67</b>
4.1	Généralités . . . . .	67
4.2	Arbres couvrants . . . . .	67
4.3	Graphes planaires . . . . .	74
4.4	Coloration de graphes . . . . .	76
<b>5</b>	<b>Flots</b>	<b>81</b>
5.1	Flots dans un graphe . . . . .	81
5.2	Flots de valeur maximum . . . . .	84
5.3	Flots de valeur maximum de coût minimum . . . . .	89
<b>6</b>	<b>Problèmes NP-Complets</b>	<b>93</b>
6.1	Problème du voyageur de commerce . . . . .	93
6.2	Problème du cycle Hamiltonien . . . . .	96
6.3	Variations autour de l'arbre couvrant minimum . . . . .	97
6.4	Problème de la clique maximum . . . . .	97
6.5	Isomorphisme de graphes . . . . .	97

<b>Table des figures</b>	<b>99</b>
<b>Liste des algorithmes</b>	<b>101</b>
<b>Index</b>	<b>102</b>
<b>Bibliographie</b>	<b>103</b>

## Sommaire

<b>1.1 Recherche opérationnelle</b>	<b>5</b>
1.1.1 Origine	5
1.1.2 Domaines d'application	6
<b>1.2 Notions de complexité algorithmique</b>	<b>7</b>
1.2.1 Complexité en temps	7
1.2.2 Complexité en espace	7
1.2.3 Ordre de grandeur	8
1.2.4 Notation asymptotique	8
<b>1.3 Classes de complexité</b>	<b>11</b>
1.3.1 Machine de Turing	11
1.3.2 Définition formelle d'une machine de Turing	12
1.3.3 Machine déterministe et machine non déterministe	13
1.3.4 Théorie de la complexité	14
1.3.5 Problème NP-Complet	15
1.3.6 Réduction de SAT vers 3-SAT	16

**1.1 Recherche opérationnelle****1.1.1 Origine**

L'ORIGINE de la R.O. est souvent attribuée à la seconde guerre mondiale. Cependant, certains travaux datant du XVII<sup>e</sup> siècle sont considérés comme faisant partie de ce domaine. En particulier, dès 1654, Blaise **Pascal** et Pierre de **Fermat** cherchaient à résoudre des problèmes de « décision dans l'incertain ». Puis, en 1776, Gaspard **Monge** a résolu un « problème économique de nature combinatoire » : le problème des déblais et remblais. Il a trouvé une solution optimale à un problème générique de transport de masse connu sous le nom de problème de *Monge-Kantorovitch*. En 1838, Antoine Augustin **Cournot** a fondé la « théorie mathématique des richesses », origine de l'économétrie. Il a travaillé sur les équilibres entre deux producteurs, appelés « équilibres de Cournot », et généralisés plus tard sous le nom « d'équilibres de **Nash** ».

Enfin, le début du XX<sup>e</sup> siècle marque une accélération dans le développement des théories liées à la recherche opérationnelle :

- Émile **Borel** a introduit la théorie des jeux à l'Académie des Sciences (1921-25) ;
- Agner Krarup **Erlang** a développé la théorie des files d'attente, utilisée lors de la conception des réseaux téléphoniques (1917) ;

- Dénes **König** a travaillé sur la théorie des graphes (1936) ;
- Leonid Vitaliyevich **Kantorovitch** a conçu et appliqué la programmation linéaire à la planification (il essaya d'optimiser la production économique soviétique).

Cependant, son grand essor date effectivement de la seconde guerre mondiale grâce à des mathématiciens de renom, dont **Von Neumann** et **Metropolis** (à l'origine de la méthode de Monte Carlo, fondée alors qu'ils travaillaient sur le projet Manhattan), Wald, **Wiener**, Dantzig, Bellman. Le nom « recherche opérationnelle » vient de la traduction d'*operations research*, qui se rapporte aux opérations militaires.

En 1940, le physicien anglais Blackett fut appelé à présider la première équipe de chercheurs opérationnels. Il a en particulier réussi à répondre rapidement et avec succès à la question de l'implantation optimale des radars de surveillance britanniques, ce qui eut un rôle déterminant lors de la bataille d'Angleterre.

Après la guerre, de nombreux essais pour étendre ces méthodes au domaine civil ont été effectués. Un certain délai a été nécessaire à cette extension. Les principales sources de ce délai sont les suivantes :

- les modèles mathématiques n'ont pas été immédiatement acceptés par les économistes ;
- les problèmes économiques ne sont devenus suffisamment complexes pour nécessiter l'utilisation d'une théorie mathématique pour leur résolution que suite à la croissance de la taille des entreprises ;
- les machines nécessaires pour la résolution des problèmes de recherche opérationnelle de grande échelle ne sont apparues qu'à la fin des années 50.

### 1.1.2 Domaines d'application

Voici une tentative de définition de la R.O. par Robert **Faure** extraite de [Faure, 1979] : la recherche opérationnelle peut être définie comme étant *l'ensemble des méthodes et techniques rationnelles d'analyse et de synthèse des phénomènes d'organisation utilisables pour élaborer de meilleures décisions*.

Ses principaux domaines d'application sont les situations dans lesquelles le sens commun se révèle impuissant, comme par exemple :

- les problèmes combinatoires ;
- les problèmes stochastiques ;
- les problèmes concurrentiels.

La première catégorie regroupe les problèmes pour lesquels il est impossible d'énumérer l'ensemble des solutions possibles. Par exemple, si on souhaite affecter 20 personnes à 20 postes, il existe  $20!$  possibilités. A raison d'un traitement par microseconde, le traitement de l'ensemble des possibilités prendrait alors 77096 ans.

La seconde catégorie regroupe tous les problèmes faisant intervenir le hasard. Par exemple, dans une entreprise, le bureau du correspondant de la sécurité sociale accueille un employé toutes les 4 minutes. Il faut en moyenne à l'employé 3 minutes 18 secondes pour servir une personne. En moyenne, un seul employé suffit donc pour traiter toutes les demandes. Cependant, il peut exister des périodes de pointe pendant lesquelles les employés devront attendre plusieurs heures avant d'être servis, ce qui nuit à la productivité de l'entreprise. Il peut donc être intéressant d'employer une deuxième personne, même si cette personne ne sert qu'à absorber les pointes de trafic. Pour répondre correctement au problème, il est donc nécessaire de modéliser le phénomène aléatoire dans toute son ampleur et ne pas se contenter d'une modélisation *en moyenne*.

Enfin, la troisième catégorie correspond au cas où la stratégie à mettre en œuvre dépend des choix effectués par des concurrents. La théorie des jeux permet de répondre à ces problèmes, ou en tout cas, apporte un cadre de réflexion qui facilite la décision.

## 1.2 Notions de complexité algorithmique

Avant d'aborder en détail les solutions apportées par la recherche opérationnelle à différents problèmes, il faut pouvoir déterminer l'efficacité des différents algorithmes, c'est-à-dire avoir des notions sur leur complexité.

Afin d'évaluer les performances d'un algorithme, il est possible de le tester avec un jeu de tests bien choisi. Cette méthode était utilisée jusque dans les années 1970. Le problème qui se pose alors est de choisir ce jeu de tests : comment s'assurer que tous les cas sont bien pris en compte ? Est-on bien tombé sur le cas le plus défavorable ?

Face aux limites de cette méthode, il a fallu s'orienter vers une étude plus systématique du comportement des algorithmes en introduisant la notion de complexité.

### 1.2.1 Complexité en temps

La notion de complexité permet d'évaluer l'efficacité des algorithmes. Elle permet de répondre à la question : entre différents algorithmes réalisant une même tâche, quel est le plus rapide et dans quelles conditions ?

Une approche indépendante des facteurs matériels est nécessaire pour évaluer cette efficacité. Donald Knuth fut un des premiers à l'appliquer systématiquement dans [Knuth, 1968]. La complexité d'un algorithme permet d'évaluer ses performances. Elle représente son nombre d'opérations caractéristiques. Elle est généralement évaluée en fonction de la taille  $N$  des données d'entrée.

Elle peut être évaluée dans le pire des cas, c'est-à-dire pour la répartition des données provoquant le plus d'opérations mais il est également possible de calculer la complexité moyenne, c'est-à-dire la moyenne des complexités pour toutes les données d'entrée possibles. Il est souvent nécessaire de spécifier une distribution de probabilité sur ces données d'entrées possibles, l'hypothèse implicite de l'équiprobabilité de tous les cas n'étant que rarement acceptable. Ainsi une probabilité plus importante pour des tableaux presque triés rendra le tri par insertion plus efficace en moyenne que le tri par sélection.

#### Remarque :

la complexité en moyenne est généralement beaucoup plus difficile à calculer que dans le pire des cas, mais elle donne souvent des informations plus pertinentes. La loi de probabilité sur les données peut être difficile à obtenir.

*Exemple 1* (Complexité en temps du tri bulle). Considérons le cas du tri bulle (figure 1).

Le nombre d'opérations effectuées par cet algorithme sur un tableau de taille  $N$  est :

$$\sum_{i=1}^{N-1} i = \frac{N(N-1)}{2} = \frac{N^2}{2} - \frac{N}{2} = \Theta(N^2)$$

### 1.2.2 Complexité en espace

On appelle complexité en espace la quantité de mémoire utilisée par un algorithme. Cette complexité devrait en toute rigueur s'évaluer en nombre d'octets, mais en pratique elle s'évalue souvent en nombre de mots mémoire (caractères, entiers, réels, ...). Dans les cas où l'on manipule de grands entiers (factorisation pour la cryptanalyse, certaines instances du problème du sac à dos, par exemple) il conviendra de se souvenir que l'encombrement mémoire de l'entier  $n$  est  $\log_2(n)$ .

**Algorithme 1** : Algorithme du tri bulles

---

```

Entrées : tab : entier[]
Données : i, j, n : entier
n ← taille(tab);
pour i ∈ [n − 1, 1] faire
    pour j ∈ [0, i − 1] faire
        si tab[j] > tab[j + 1] alors
            inverser(tab[j], tab[j + 1]);
        fin
    fin
fin

```

---

*Exemple 2* (Complexité en espace du tri bulle). Toujours dans le cas du tri bulle (figure 1) :

la quantité de mémoire nécessaire pour effectuer un tri bulle sur un tableau de taille  $N$  est  $N + 1$  ( $N$  variables pour le tableau, et une pour l'échange. Les indices ne sont pas pris en compte).

### 1.2.3 Ordre de grandeur

Il est généralement peu utile d'avoir l'expression exacte du temps de calcul d'un algorithme en fonction de la taille des données d'entrée. Dans la plupart des cas, seul l'ordre de grandeur de la complexité nous intéresse. On ne considérera donc que le terme dominant de la complexité. Par exemple dans le cas d'une complexité valant  $an^2 + bn + c$ , seul le terme  $an^2$  sera pris en compte.

De même, on ignorera le coefficient multiplicateur constant du terme prépondérant puisque les facteurs constants sont moins importants que l'ordre de grandeur dans l'évaluation de l'efficacité d'un algorithme. Nous dirons donc que la complexité du tri bulle est de  $\Theta(n^2)$ .

**Remarque :**

la notation  $\Theta$  sera définie dans la section 1.2.4.

### 1.2.4 Notation asymptotique

Cette partie définit les notations asymptotiques — ou notations de Landau — utilisées lors des calculs de complexité. Elles sont introduites par exemple dans [Cormen *et al.*, 1994] et bien entendu dans [Knuth, 1968].

**Notation  $\Theta$**

Pour une fonction  $g(n)$  donnée, on note  $\Theta(g(n))$  l'ensemble des fonctions suivant :

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2 \in \mathbb{R}^{+*}, n_0 \in \mathbb{N}, \text{ tel que } \forall n \geq n_0, 0 < c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

$\Theta(g(n))$  représente l'ensemble des fonctions de même ordre que  $g(n)$ . Une fonction  $f(n)$  appartient à  $\Theta(g(n))$  s'il existe deux constantes positives  $c_1$  et  $c_2$  telles que  $f(n)$  puisse être encadrée par  $c_1 g(n)$  et par  $c_2 g(n)$ .

La fonction  $f(n)$  est donc égale à la fonction  $g(n)$  à un facteur multiplicateur et un terme négligeable devant  $f$  (ou  $g$ , ce qui revient au même) près.

**Remarque :**

$\Theta(g(n))$  représente un ensemble, et il faut donc écrire en toute rigueur  $f(n) \in \Theta(g(n))$ . Il est cependant d'usage d'utiliser l'abus de notation  $f(n) = \Theta(g(n))$ .

*Exemple 3* (Notation  $\Theta$ ). Quelques exemples d'utilisation de  $\Theta$  :

— la complexité du tri bulle s'écrit  $\Theta(n^2)$  ;



- la complexité du tri par segmentation est  $\Theta(n^2)$  dans le pire des cas et  $\Theta(n \log n)$  en moyenne ;
- une complexité indépendante du nombre de données en entrée s'écrit  $\Theta(n^0)$  ou encore  $\Theta(1)$ . La deuxième notation est celle le plus couramment utilisée.

**Notation  $\mathcal{O}$** 

La notation  $\mathcal{O}$  est utilisée dans le cas où nous ne disposons que d'une majoration de la complexité. Partant d'une fonction  $g(n)$ , l'ensemble  $\mathcal{O}(g(n))$  est défini par :

$$\mathcal{O}(g(n)) = \{f(n) : \exists c \in \mathbb{R}^{+*}, n_0 \in \mathbb{N} / \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$$

La notation  $\mathcal{O}$  sert à majorer une fonction à un facteur constant près.

**Remarque :**

Contrairement à  $\Theta(g(n))$ ,  $\mathcal{O}(g(n))$  ne représente qu'une majoration de  $g(n)$  et contient donc plus que les fonctions du même ordre que  $g$ .

On remarquera que :

$$f(n) \in \Theta(g(n)) \Rightarrow f(n) \in \mathcal{O}(g(n))$$

ce qui peut également s'écrire :

$$\Theta(g(n)) \subseteq \mathcal{O}(g(n))$$

*Exemple 4* (Utilisation de la notation  $\mathcal{O}$ ). Soit  $f(n) = an + b$ . On montre facilement (la preuve est laissée à titre d'exercice) que  $f(n) \in \mathcal{O}(n)$  mais également, ce qui peut paraître plus surprenant, que  $f(n) \in \mathcal{O}(n^2)$ .

**Remarque :**

La notation  $\Theta$  définit des classes d'équivalence :

$$f(n) \in \Theta(g(n)) \iff \begin{cases} f(n) \in \mathcal{O}(g(n)) \\ g(n) \in \mathcal{O}(f(n)) \end{cases} \iff g(n) \in \Theta(f(n))$$

**Notation  $\Omega$** 

La notation  $\Omega$  est comparable à la notation  $\mathcal{O}$ , mais elle définit une borne inférieure asymptotique au lieu d'une borne supérieure asymptotique.

Étant donnée une fonction  $g(n)$ , on définit  $\Omega(g(n))$  par :

$$\Omega(g(n)) = \{f(n) : \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} / \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$$

**Remarque :**

soient deux fonctions  $f(n)$  et  $g(n)$ .

$$f(n) \in \Theta(g(n)) \iff f(n) \in \mathcal{O}(g(n)) \text{ et } f(n) \in \Omega(g(n))$$

**Opérations de base sur les fonctions**

Cette partie présente quelques résultats élémentaires sur les calculs de complexité.

$$\begin{aligned}
f(n) &= \Theta(f(n)) \\
c \cdot \Theta(f(n)) &= \Theta(f(n)) \\
\Theta(f(n)) + \Theta(f(n)) &= \Theta(f(n)) \\
\Theta(\Theta(f(n))) &= \Theta(f(n)) \\
\Theta(f(n))\Theta(g(n)) &= \Theta(f(n)g(n)) \\
\Theta(f(n)g(n)) &= f(n)\Theta(g(n)) \\
\sum_{i=0}^m a_i n^i &= \Theta(n^m) \\
\forall \alpha \in \mathbb{R}^{+*}, n^\alpha + \log n &= \Theta(n^\alpha) \\
\forall \alpha \in \mathbb{R}^{+*}, \forall x > 1, n^\alpha + x^n &= \Theta(x^n) \\
\forall x \in \mathbb{R}^{+*}, n! + x^n &= \Theta(n!) \\
2^{2^n} + n! &= \Theta(2^{2^n})
\end{aligned}$$

### Bons algorithmes

Un bon algorithme est un algorithme polynomial. Ou plus exactement, un mauvais algorithme est un algorithme non polynomial. En effet, l'augmentation de la puissance des machines peut permettre d'étudier des algorithmes polynomiaux sur des tailles de données plus importantes, mais elle n'a pratiquement pas d'influence pour les algorithmes exponentiels.

Ainsi, pour passer d'un problème de taille 50 à un problème de taille 100 en restant à temps constant, il faut multiplier la puissance par 10 pour un problème en  $n^3$ , par 4800 pour un problème en  $n^{\log n}$ , et par 100000 pour un problème en  $2^{\frac{n}{3}}$ .

Le tableau 1.1 présente les temps de calcul d'algorithmes de différentes complexités en fonction de la taille des données. On suppose qu'une opération est réalisée en  $1\mu s$ . Les cases non remplies ont une valeur supérieure à 1000 milliards d'années.

Complexité \ Taille	20	50	100	200	500	1000
$10^3 n$	0.02 s	0.05 s	0.1 s	0.2 s	0.5 s	1 s
$10^3 n \log n$	0.09 s	0.3 s	0.6 s	1.5 s	4.5 s	10 s
$100n^2$	0.04 s	0.25 s	1 s	4 s	25 s	2 mn
$10n^3$	0.02 s	1 s	10 s	1 mn	21 mn	27 h
$n^{\log n}$	0.4 s	1.1 h	220 j	12500 ans	$5.10^{10}$ ans	
$2^{\frac{n}{3}}$	$10^{-4}$ s	0.1 s	2.7 h	$3.10^6$ ans		
$2^n$	1 s	36 ans				
$3^n$	58 mn	$2.10^{11}$ ans				
$n!$	77100 ans					

TABLE 1.1 – Temps de calcul pour différentes complexités

### Remarque :

les complexités citées ci-dessus sont dites « élémentaires ». Il existe également des complexités non-élémentaires ou *hyperexponentielles*, c'est-à-dire non bornable par une tour d'exponentielle  $2^{2^{\dots^n}}$ . Un algorithme possédant une complexité non-élémentaire possède un intérêt purement théorique et ne pourra jamais être exécuté par un ordinateur, si puissant soit-il. Un exemple d'une telle complexité est décrit ci-dessous :

$$C(n) = 2^{2^{2^{\dots^2}}} \} n \text{ fois}$$

Autrement dit :  $C(n+1) = 2^{C(n)}$ .

### 1.3 Classes de complexité

Dans cette partie, nous allons essayer de préciser le sens de « problème intrinsèquement difficile ». Notons qu'il existe des problèmes indécidables pour lesquels il ne peut pas exister d'algorithme de résolution. Ces problèmes (mis en évidence par Alan Turing) sont bien entendu difficiles. Il existe cependant des problèmes difficiles hors de cette classe. De manière générale, on définit la classe P (définition 1) comme étant celle des problèmes faciles, et les problèmes qui n'en font pas partie sont alors considérés comme étant difficiles. Dans toute la suite de cette partie, nous nous intéresserons uniquement aux problèmes de reconnaissance (définition 2). Les notions de classe de complexité et de machine de Turing seront uniquement survolées dans cette section. Pour des informations complémentaires, consulter [Rey, 2004; Stern, 1990].

**Définition 1** (Classe P). Un problème fait partie de la classe P s'il existe un algorithme polynomial permettant de le résoudre.

*Exemple 5* (Problème polynomial). Le problème de tri d'un ensemble de nombres fait partie de la classe P. Il existe en effet des tris de complexité  $\Theta(n^2)$  et même  $\Theta(n \log n)$ .

**Définition 2** (Problèmes de reconnaissance). Un problème de reconnaissance est un problème pour lequel les résultats ne peuvent prendre que les valeurs VRAI ou FAUX.

*Exemple 6* (Problèmes de reconnaissance). On considère le problème de satisfaisabilité (problème SAT) : soit un ensemble  $X = \{x_1, \dots, x_n\}$  de variables booléennes. Soit une expression  $E = C_1 \wedge \dots \wedge C_m$  avec  $\forall i, C_i = u_{i,1} \vee \dots \vee u_{i,k_i}$ , chaque  $u_{i,j}$  étant une variable de  $X$ , complémentée<sup>1</sup> ou non. Le problème consiste à déterminer un ensemble de valeurs pour les  $x_i$  tel que  $E$  soit VRAI (noté 1).

Par exemple, pour  $E = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2) \wedge (x_3)$ , la réponse est VRAI pour  $x_1 = 0$  ou 1,  $x_2 = 1$ ,  $x_3 = 1$ .

Avant d'aborder quelques résultats sur les classes de complexité, il est utile d'introduire la notion de machine de Turing.

#### 1.3.1 Machine de Turing

Une machine de Turing est un modèle abstrait du fonctionnement d'un ordinateur et de sa mémoire, créé par Alan Turing en 1936 en vue de donner une définition précise au concept d'algorithme ou « procédure mécanique ». Ce modèle est toujours largement utilisé en informatique théorique, en particulier pour résoudre les problèmes de complexité algorithmique et de calculabilité. Une machine de Turing (représentée figure 1.1) est composée des éléments suivant :

- une mémoire ;
- un lecteur ;
- des états internes.

**La mémoire :** c'est là que s'inscrit l'information et que s'opèrent les altérations et transformations de celle-ci. Elle peut être représentée par un ruban de longueur infinie divisé en cases, chaque case pouvant recevoir une donnée. Les données font partie d'un alphabet  $\mathcal{A} = (S_0, \dots, S_n)$  pour lequel le symbole  $S_0$  (noté 0) désigne le symbole blanc, ou l'absence de symbole.

**Le lecteur :** le ruban défile devant un lecteur qui peut lire une case mémoire à la fois. Le ruban peut se déplacer devant le lecteur d'une case vers la gauche ou d'une case vers la droite.

1. Le complémentaire de  $x$ ,  $\bar{x}$ , est VRAI si et seulement si  $x$  est FAUX

**Les états internes :** à chaque instant, le lecteur peut effectuer les actions suivantes sur la mémoire une action sur la case mémoire courante et un déplacement sur le ruban :

1. rien ;
2. effacer le symbole lu;
3. effacer le symbole lu et écrire un autre symbole à la place ;
4. avancer le ruban d'une case vers la gauche ;
5. avancer le ruban d'une case vers la droite ;
6. effacer le symbole lu et avancer le ruban d'une case vers la gauche;
7. effacer le symbole lu, écrire un autre symbole à la place et avancer le ruban d'une case vers la gauche;
8. effacer le symbole lu et avancer le ruban d'une case vers la droite;
9. effacer le symbole lu, écrire un autre symbole à la place et avancer le ruban d'une case vers la droite;

L'action à effectuer dépend du symbole lu sur le ruban, mais également de l'état du système. On suppose que le système peut prendre un nombre fini d'états noté  $\mathcal{Q} = (q_0, \dots, q_n)$ .  $q_0$  représente l'état initial, et  $q_n$  (ou  $q_{acc}$ ) l'état final.

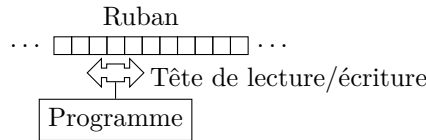


FIGURE 1.1 – Machine de Turing

#### Fonctionnement d'une machine de Turing :

la machine de Turing est initialisée avec un certain ruban, le lecteur placé sur une certaine case. Elle est initialement dans l'état  $q_0$ . Elle commence alors à fonctionner en lisant le symbole du ruban en face du lecteur, en effectuant l'action correspondante et en passant dans un nouvel état.

#### 1.3.2 Définition formelle d'une machine de Turing

**Définition 3** (Machine de Turing). On appelle machine de Turing le quintuplet  $\mathcal{T} = \{Q, A, \square, \delta, q_0\}$  avec :

- $Q$  ensemble fini d'états
- $A$  ensemble fini de symboles (alphabet)
- $\square$  symbole blanc
- $\delta : Q \times A \cup \{\square\} \rightarrow (A \cup \{\square\}) \times \{L, R, N\} \times Q$  une fonction de transition (L=left, R=right, N=none)
- $q_0 \in Q$  l'état initial

Et  $Q$  et  $A$  deux ensembles finis disjoints et  $\square \notin A \cup Q$

On note  $\hat{A} = A \cup \{\square\}$

Par la suite, on utilisera la notation  $\lambda x_1 \dots x_n [f(x_1, \dots, x_n)]$  pour représenter les fonctions  $f : \mathbb{N}^k \Rightarrow \mathbb{N}$ .

*Exemple 7* (Machine de Turing). On considère la machine de Turing calculant  $\lambda x[2x]$  utilisant l'alphabet  $A = \{1\}$  (uniquement le symbole 1). Pour définir cette machine, il reste à préciser l'ensemble d'états  $Q$  et la fonction de transition  $\delta$ .

Principe de la machine de Turing :

- si on est sur un  $\square$ , alors terminer l'exécution ;
- sinon, pour tous les 1 du nombre ;
  - effacer le premier 1 ;
  - ajouter deux 1 en fin de nombre ;
- à la fin, se placer sur le premier 1 ;
- pour que tout fonctionne, séparer les 1 des deux nombres par  $\square$ .

Un automate utilisant ce principe est représenté figure 1.2. Cet automate est représenté sous forme de tableau figure 1.3.

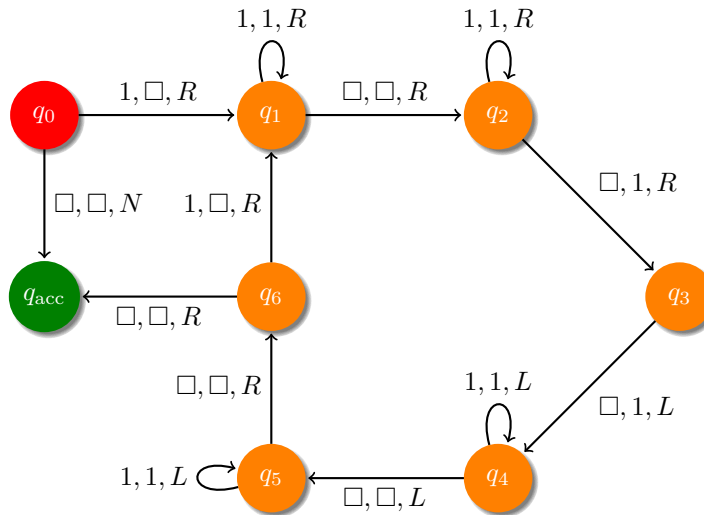


FIGURE 1.2 – Automate de la machine de Turing calculant  $\lambda x[2x]$

	$\square$	1
$q_0$	$\square, N, q_{acc}$	$\square, R, q_1$
$q_1$	$\square, R, q_2$	$1, R, q_1$
$q_2$	$1, R, q_3$	$1, R, q_2$
$q_3$	$1, L, q_4$	
$q_4$	$\square, L, q_5$	$1, L, q_4$
$q_5$	$\square, R, q_6$	$1, L, q_5$
$q_6$	$\square, R, q_{acc}$	$\square, R, q_1$

FIGURE 1.3 – Tableau représentant la fonction de transition de la machine de Turing

### 1.3.3 Machine déterministe et machine non déterministe

Une machine déterministe est le modèle formel d'une machine telle que nous les connaissons. Nos ordinateurs ainsi que les machines de Turing sont des machines déterministes.

Une machine non déterministe est une machine déterministe (de Turing par exemple) à laquelle on a greffé un *oracle*. L'oracle peut générer une suite quelconque de bits (ou de mots de

l'alphabet  $\mathcal{A}$ ) et la machine doit se contenter de vérifier que la suite de bits en question est solution du problème donné.

Par exemple, le problème consistant à dire si un entier est factorisable (c'est-à-dire non premier) se résout sur machine non déterministe avec un algorithme de ce type :

```
demander à l'oracle  $p > 1, p < x$ 
demander à l'oracle  $q > 1, q < x$ 
renvoyer vrai si  $pq=x$ 
```

La machine renvoie vrai s'il existe un  $p$  et un  $q$  tel que  $pq = x$ . Dans la pratique on peut émuler une machine non déterministe sur machine déterministe. Dans notre exemple il suffit de remplacer l'oracle par une fonction générant tout les couples  $(p, q)$  avec  $p, q > 1$  et  $p, q < x$ . Évidemment un tel calcul prend un temps énorme comparé à une simple multiplication. Résoudre un problème sur machine non déterministe équivaut donc à savoir vérifier si une solution proposée fonctionne.

La notion de machine de Turing non déterministe est à la base de la définition de la classe de problèmes NP.

**Définition 4** (Classe NP). Un problème fait partie de la classe NP s'il peut être résolu en temps polynomial sur une machine de Turing non déterministe.

#### Remarques

- :
- la classe NP n'est en aucun cas la classe des problèmes pouvant être résolus en un temps non polynomial ;
- la classe NP se compose des problèmes qui sont « vérifiables » par un algorithme polynomial. Autrement dit, possédant une solution, il est possible de vérifier si elle satisfait le problème en temps polynomial ;
- la classe NP contient trivialement la classe P.

### 1.3.4 Théorie de la complexité

En théorie de la complexité, un problème est représenté par un ensemble de données en entrée, et une question sur ces données (pouvant demander éventuellement un calcul). On ne traite que des problèmes de décision binaire, c'est-à-dire posant une question dont la réponse est oui ou non. Cependant on étend la notion de complexité aux problèmes d'optimisation (consistant à trouver la solution optimale d'un problème). En effet il est facile de transformer un problème d'optimisation en problème de décision.

Si par exemple on cherche à optimiser une valeur  $n$  on traite le problème de décision qui consiste à comparer  $n$  à un certain  $k$ . En traitant plusieurs valeurs de  $k$  on peut déterminer une valeur optimale.

Si l'on dispose d'une fonction Décision( $n$ ) qui détermine en un temps  $f(n)$  si la réponse est « oui » pour l'entier  $n$ , il suffit de faire des appels à Décision( $n$ ) en commençant à  $n = 1$  et en passant de  $n$  à  $2n$  si la réponse est non. Une fois atteinte la réponse oui, on dispose d'un intervalle  $[\frac{n}{2}, n]$  dans lequel se trouve la valeur  $k$  optimale, en  $\log_2(k)f(2k)$  opérations. Par dichotomie, avec  $\log_2(k)$  appels supplémentaires à Décision(.) on obtient la valeur  $k$  qui répond au problème d'optimisation. En notant que  $\log_2(k)$  est une fonction linéaire de la taille de la solution du problème d'optimisation, tout algorithme polynomial de résolution d'un problème de décision fournit ainsi un algorithme polynomial de résolution du problème d'optimisation associé. On confondra souvent un problème d'optimisation et son problème de décision associé.

On considère que l'ensemble des instances d'un problème est l'ensemble des données que peut accepter ce problème en entrée, par exemple l'ensemble des permutations de  $n$  entiers à trier pour un algorithme de tri.

On distingue les classes de complexité suivantes :

**L** : un problème de décision qui peut être résolu par un algorithme en espace logarithmique par rapport à la taille de l'instance avec une machine de Turing déterministe.

**NL** : cette classe correspond à la précédente mais pour une machine de Turing non déterministe.

**P** : un problème de décision est dans P s'il peut être décidé par un algorithme en un temps polynomial par rapport à la taille de l'instance sur une machine de Turing déterministe. On qualifie alors le problème de polynomial.

**NP** : c'est la classe des problèmes de décision pour lesquels la réponse oui peut être décidée par un algorithme en un temps polynomial par rapport à la taille de l'instance sur une machine de Turing non déterministe.

Autre définition de la classe NP : la classe NP est formée des problèmes de décision qui possèdent un vérifieur polynomial, c'est-à-dire qu'il est possible de vérifier si une valeur est solution du problème en temps polynomial.

**Co-NP** : nom parfois donné pour l'équivalent de la classe NP avec la réponse non (*i.e.* le problème peut être résolu par un algorithme qui vérifie qu'une valeur n'est pas solution du problème en temps polynomial).

**PSPACE** : les problèmes décidables par un algorithme en espace polynomial par rapport à la taille de son instance.

**NSPACE ou NPSPACE** : les problèmes décidables par un algorithme en espace polynomial par rapport à la taille de son instance sur une machine de Turing non déterministe.

**EXPTIME** : les problèmes décidables par un algorithme en temps exponentiel par rapport à la taille de son instance.

On a les inclusions:  $P \subseteq NP$ ,  $Co-NP \subseteq PSPACE = NPSPACE \subseteq EXP$  et  $P \subsetneq EXP$ .

**Remarque :**

- Il existe beaucoup d'autres classes de complexité qui ne seront pas détaillées dans ce cours. Vous pouvez en trouver un grand nombre à l'adresse suivante :

[https://complexityzoo.uwaterloo.ca/Complexity\\_Zoo](https://complexityzoo.uwaterloo.ca/Complexity_Zoo)

### 1.3.5 Problème NP-Complet

La notion de problème NP-complet s'appuie sur la notion de réduction polynomiale.

**Définition 5** (Réduction polynomiale). Soient  $(P_1)$  et  $(P_2)$  deux problèmes de reconnaissance. On dit que  $(P_1)$  se réduit polynomialement à  $(P_2)$  s'il existe un algorithme de résolution de  $(P_1)$  qui utilise un nombre polynomial d'appels à  $(P_2)$ .

**Théorème 1.** Si  $(P)$  se réduit polynomialement en  $(P')$  et si  $(P')$  peut être résolu en temps polynomial, alors il en est de même pour  $(P)$ .

**Définition 6** (NP-complet). Un problème de reconnaissance est dit NP-complet si tout problème de la classe NP peut se réduire polynomialement à lui.

La classe des problèmes NP-complets joue un rôle très important. En effet, selon le théorème 1, si on trouve un algorithme polynomial permettant de résoudre un problème NP-complet, alors tous les problèmes de la classe NP peuvent être résolus en temps polynomial.

**Remarque :**

la réduction d'un problème  $A$  de taille  $n$  en un problème  $B$  se fait en général en augmentant la taille de l'instance, celle-ci devenant  $f(n)$ . La réduction n'est polynomiale que si  $f$  est majorée par un polynôme.

**Remarque :**

de manière intuitive, dire qu'un problème peut être décidé à l'aide d'un algorithme polynomial sur une machine de Turing non déterministe signifie qu'il est facile, pour une solution donnée, de vérifier en un temps polynomial si celle-ci répond au problème pour une instance donnée mais que le nombre de solutions à tester pour résoudre le problème est exponentiel par rapport à la taille de l'instance. Le non-déterminisme permet de masquer la quantité exponentielle des solutions à tester tout en permettant à l'algorithme de rester polynomial.

**Théorème 2** (Théorème de Cook). *Il existe des problèmes NP-complets.*

Ce théorème a été prouvé en 1971 par Stephen **Cook** dans un article intitulé *The Complexity of Theorem Proving Procedures* [Cook, 1971]. Dans cet article, Cook a démontré que le problème SAT (le problème de l'exemple 6) est NP-complet.

Toutes les autres démonstrations de NP-complétude consistent à trouver une réduction polynomiale vers le problème SAT.

### 1.3.6 Réduction de SAT vers 3-SAT

Le problème SAT (exemple 6) se compose de  $n$  variables booléennes et de  $m$  disjonctions – dites clauses – comportant chacune au plus  $n$  variables. La taille occupée en mémoire par une instance de ce problème est majorée par  $mn$ .

Le problème 3-SAT est constitué par les instances de SAT ne comportant que des clauses de trois variables. Sa taille est  $3m$ .

**Théorème 3.** *3-SAT est NP-complet.*

PREUVE :

La preuve se fait réduction polynomiale du problème SAT en problème 3-SAT.

Considérons une instance de SAT constituée d'un ensemble  $X$  de  $n$  variables  $x_1, \dots, x_n$  et de  $m$  clauses,  $C_1, \dots, C_m$ . Une clause s'écrit  $C_k = u_{k1} \vee \dots \vee u_{kn_k}$ , où les  $u_{k,j}$  sont des éléments de  $X$ , et où le symbole  $\hat{x}$  peut désigner  $x$  ou  $\bar{x}$ .

Toutes les variables de l'instance de SAT sont reprises dans l'instance de 3-SAT.

Pour chaque clause  $C_k$  de l'instance de SAT, on générera des clauses à trois variables pour alimenter l'instance de 3-SAT, et éventuellement de nouvelles variables :

- si  $C$  est une clause à une variable :  $C = \hat{u}$ , on ajoute deux variables  $t$  et  $t'$ , et quatre clauses :  $\hat{u} \vee t \vee t'$ ,  $\hat{u} \vee \bar{t} \vee t'$ ,  $\hat{u} \vee t \vee \bar{t}'$  et  $\hat{u} \vee \bar{t} \vee \bar{t}'$ . Si  $\hat{u}$  est vraie les quatre clauses ajoutées sont vraies. Si  $\hat{u}$  est fausse, quelque soit la distribution de valeurs de vérité sur  $t$  et  $t'$ , l'une des quatre clauses est fausse, et donc la conjonction des quatre clauses est fausse ;
- si  $C$  est une clause à deux variables :  $C = \hat{u} \vee \hat{v}$ , on ajoute une variable  $t$  et deux clauses :  $\hat{u} \vee \hat{v} \vee t$ , et  $\hat{u} \vee \hat{v} \vee \bar{t}$ . Si  $\hat{u}$  ou  $\hat{v}$  est vraie, les deux clauses sont vraies. Si  $\hat{u}$  et  $\hat{v}$  sont fausses, quelque soit la valeur de vérité de  $t$ , l'une des deux clauses est fausse ;
- si  $C$  est une clause à trois variables, elle est simplement conservée dans l'instance de 3-SAT ;
- si  $C$  est une clause à quatre variables ou plus :  $C = \hat{u}_1 \vee \dots \vee \hat{u}_k$ , on ajoute  $k - 3$  variables  $t_1, \dots, t_{k-3}$  et  $k - 2$  clauses :  $\hat{u}_1 \vee \hat{u}_2 \vee t_1$ ,  $\bar{t}_1 \vee \hat{u}_3 \vee t_2$ ,  $\bar{t}_2 \vee \hat{u}_4 \vee t_3$ ,  $\dots$ ,  $\bar{t}_{k-4} \vee \hat{u}_{k-2} \vee t_{k-3}$  et  $\bar{t}_{k-3} \vee \hat{u}_{k-1} \vee \hat{u}_k$ . Chaque variable  $t_k$  apparaît deux fois : une fois sous forme directe, une fois sous forme complémentée. Chacune peut donc valider au plus une clause. Comme il y a  $k - 2$  clauses et  $k - 2$  variables ajoutées, si toutes les  $\hat{u}_j$  sont fausses l'une au moins des clauses ajoutées est fausse. Si l'une des  $\hat{u}_j$  ( $3 \leq j \leq k - 2$ ) est vraie, elle rend vraie la clause  $\bar{t}_{j-2} \vee \hat{u}_j \vee t_{j-1}$ . Mettre à vrai les  $t_i$  pour  $i \leq j - 2$  et à faux les  $t_i$  pour  $i \geq j - 1$  rend vraies les autres clauses. Si  $\hat{u}_1$  ou  $\hat{u}_2$  est vrai il suffit de mettre tous les  $t_i$  à faux pour valider les clauses. Si  $\hat{u}_{k-1}$  ou  $\hat{u}_k$  est vrai il suffit de mettre tous les  $t_i$  à vrai.

Ainsi, l'instance de 3-SAT construite est satisfaisable si et seulement si l'instance de SAT l'est.

Pour chaque clause de l'instance de SAT on a ajouté au plus  $n - 3$  variables, et  $n - 2$  clauses à trois variables. La taille de l'instance de 3-SAT obtenue a  $\mathcal{O}(nm)$  variables et  $\mathcal{O}(nm)$  clauses, chacune étant d'une longueur bornée (3). La taille de l'instance de 3-SAT obtenue est bien bornée par un polynôme de la taille de SAT, qui était déjà de l'ordre de  $nm$ .



Le procédé décrit précédemment définit bien une réduction polynomiale de SAT en 3-SAT, et 3-SAT est donc un problème NP-complet.  $\square$

De façon immédiate, pour  $K \geq 3$ , K-SAT est également NP-complet. Par contre, 2-SAT est dans P.

**Remarque :**

on note que la plupart des preuves de NP-complétude consistent à trouver une réduction de 3-SAT vers problème en question. Il est en effet plus simple de trouver une réduction partant de 3-SAT que de SAT.

**Exemples de problèmes NP-complets :**

le problème de satisfaisabilité de l'exemple 6 est NP-complet.

Il existe de nombreux problèmes NP-Complets célèbres liés à la théorie des graphes, dont les problèmes suivants :

- le voyageur de commerce,
- recherche de cycle hamiltonien,
- calcul de la clique maximum,
- colorations de graphes, pour au moins trois couleurs
- recherche d'ensemble dominant dans un graphe,
- recherche de couverture de sommets dans un graphe.

Certains de ces problèmes seront décrits dans la suite du cours.

**Remarque : P vs NP**

La figure 1.4 représente la relation entre les classes P, NP, et NP-complet.

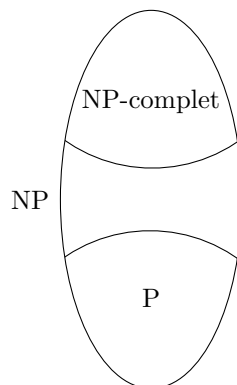



FIGURE 1.4 – P et NP

On a trivialement  $P \subseteq NP$  car un algorithme déterministe est un algorithme non déterministe particulier. Par contre la réciproque :  $NP \subseteq P$ , que l'on résume généralement à  $P = NP$  du fait de la trivialité de l'autre inclusion, est l'un des problèmes ouverts les plus fondamentaux et intéressants en informatique théorique. Cette question a été posée en 1970 pour la première fois et celui qui arrivera à prouver que  $P$  et  $NP$  sont différents ou égaux (ou que le problème est indécidable) recevra le prix Clay (plus de \$1.000.000)



## Sommaire

<b>2.1 Programmes linéaires</b>	<b>20</b>
2.1.1 Résolution géométrique de programmes linéaires dans $\mathbb{R}^2$	21
2.1.2 Forme canonique, forme standard	21
2.1.3 Programmes linéaires duaux	23
2.1.4 Théorème de dualité, théorème des écarts complémentaires	25
<b>2.2 Algorithme du simplexe</b>	<b>26</b>
2.2.1 Définitions	27
2.2.2 Initialisation de l'algorithme	27
2.2.3 Itération de l'algorithme	28
2.2.4 Critère d'arrêt	28
<b>2.3 Algorithme du simplexe en tableaux</b>	<b>29</b>
2.3.1 Lecture du tableau	30
2.3.2 Première itération	30
2.3.3 Itérations suivantes	30
<b>2.4 Traitement des cas particuliers</b>	<b>31</b>
2.4.1 Initialisation de l'algorithme	31
2.4.2 Solution non bornée	33
2.4.3 Système dégénéré	34
2.4.4 Implantation et performances de l'algorithme du simplexe	35

 OPTIMISATION linéaire a pour objet l'étude des *programmes linéaires*, c'est-à-dire des problèmes d'optimisation exprimés sous forme de systèmes linéaires. Elle représente une part importante de la recherche opérationnelle, permettant de résoudre de nombreux problèmes concrets.

Le terme *programmation linéaire* a été introduit par G. B. Dantzig, le père de la méthode du simplexe, au lendemain de la seconde guerre mondiale. Elle est abordée par tous les ouvrages se rapportant à la recherche opérationnelle comme [Sakarovitch \[1984a\]](#); [De Werra et al. \[2003\]](#); [Faure et al. \[2009\]](#) ainsi que dans la plupart des ouvrages d'algorithmique comme [Cormen et al. \[1994\]](#). Les ouvrages de programmation mathématique tel que [Minoux \[2008\]](#) décrivent également cette méthode.

Une fois un problème modélisé sous la forme d'équations linéaires, des méthodes assurent la résolution du problème de manière exacte. On distingue dans la programmation linéaire, la programmation linéaire en nombres réels, pour laquelle les variables des équations sont dans  $\mathbb{R}^+$

et la programmation en nombres entiers, pour laquelle les variables sont dans  $\mathbb{N}$ . Bien entendu, il est possible d'avoir les deux en même temps (on parle alors de programme linéaire mixte). Notons que la résolution d'un problème avec des variables entières est nettement plus difficile que dans le cas d'un problème en nombres réels.

## 2.1 Programmes linéaires

**Définition 7** (Programme linéaire). Un *programme linéaire* est un problème dans lequel les variables sont des réels qui doivent satisfaire un ensemble d'équations/inéquations linéaires appelées *contraintes* et une fonction linéaire de ces variables appelée *fonction objectif* doit être maximisée ou minimisée. Par la suite, nous noterons PL un programme linéaire.

*Exemple 8* (Programme linéaire). De nombreux problèmes d'optimisation peuvent se modéliser sous forme de programme linéaire. Par exemple, considérons le problème de production suivant : une entreprise fabrique deux produits A et B à partir des matières premières  $m_1$ ,  $m_2$  et  $m_3$ . Le produit A nécessite 2 unités de  $m_1$  et 1 de  $m_2$ , et le produit B 1 de  $m_1$ , 2 de  $m_2$  et 1 de  $m_3$ . Le profit réalisé par unité de A est de 4, et celui par unité de B est de 5. L'objectif est bien entendu de maximiser le profit sachant que les matières premières sont en quantité limitée : 8 pour  $m_1$ , 7 pour  $m_2$  et 3 pour  $m_3$ .

Le problème se formule alors par le PL suivant :

$$\begin{cases} \text{Maximiser} & 4x_1 + 5x_2 & (1) \\ \text{Sous les contraintes} & & \\ 2x_1 + x_2 & \leq 8 & (2) \\ x_1 + 2x_2 & \leq 7 & (3) \\ x_2 & \leq 3 & (4) \\ x_1, x_2 & \geq 0 & (5) \end{cases}$$

**Définition 8** (Programme mathématique). Soit un domaine  $\mathcal{D} \subset \mathbb{R}^n$  et une fonction

$$f : \mathcal{D} \rightarrow \mathbb{R}$$

On appelle programme mathématique un problème consistant à chercher un élément  $x \in \mathcal{D}$  pour lequel  $f(x)$  est maximum ou minimum. Ce problème se note symboliquement :

$$\begin{aligned} & \max_{x \in \mathcal{D}} (f(x)) \\ \text{ou} & \min_{x \in \mathcal{D}} (f(x)) \end{aligned}$$

*Remarque.* Étant donné un programme mathématique, les quatre cas suivants sont envisageables :

- $\mathcal{D} = \emptyset$  : le problème n'a pas de solution réalisable ;
- $\mathcal{D} \neq \emptyset$  et le problème a une solution optimale ;
- $\mathcal{D} \neq \emptyset$  et la fonction n'est pas bornée sur  $\mathcal{D}$  ; il n'existe pas de solution optimale ;
- $\mathcal{D} \neq \emptyset$ , la fonction est bornée, mais il n'existe pas de solution optimale.

Par exemple :  $\max_{x/x < 0} (x)$ . Ce cas ne peut pas se produire en programmation linéaire.

*Remarque.*

$$\min_{x \in \mathcal{D}} (f(x)) = -\max_{x \in \mathcal{D}} (-f(x)) \quad (2.1)$$

**Définition 9** (Solutions réalisables/optimales). Soit  $(P)$  un problème linéaire. Tout point  $x$  vérifiant les contraintes de  $(P)$  est appelé *solution réalisable*. Parmi les solutions réalisables, celles qui maximisent la fonction objectif sont appelées *solutions optimales*.

### 2.1.1 Résolution géométrique de programmes linéaires dans $\mathbb{R}^2$

Dans le cas des programmes linéaires en dimension 2, il est possible d'effectuer une résolution géométrique. Chacune des contraintes peut se traduire par un demi-plan dans lequel la solution doit se trouver. L'ensemble des contraintes donne alors une région du plan que nous noterons  $\mathcal{C}$ .

**Définition 10** (Région réalisable). On appelle *région réalisable* l'ensemble des valeurs des variables du programme qui satisfont toutes les contraintes.

Dans l'exemple 8 la région réalisable  $\mathcal{C}$  est la zone hachurée sur la figure 2.1. Nous voulons alors maximiser la fonction objectif, c'est-à-dire trouver le plus grand  $z$  tel que la droite d'équation  $4x_1 + 5x_2 = z$  ait une intersection non vide avec  $\mathcal{C}$ .

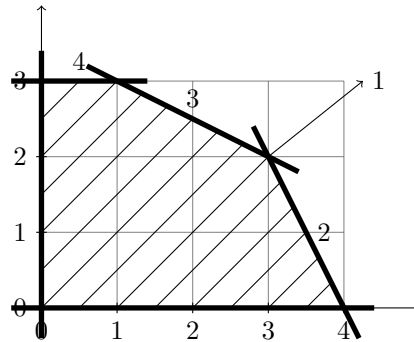


FIGURE 2.1 – Problème linéaire

Cette méthode de résolution peut être généralisée dans le cas des problèmes en dimension 3 ou plus, mais la représentation est plus délicate. . . . Dans le cas général de  $\mathbb{R}^n$ , la région réalisable n'a plus pour frontière un polygone convexe mais un polyèdre convexe.

**Définition 11** (Polyèdre convexe). On appelle *polyèdre convexe* de  $\mathbb{R}^n$  l'ensemble des  $x \in \mathbb{R}^n$  vérifiant un système d'inéquations linéaires :

$$\sum_{i=1}^n a_{ki} x_i \leq b_k \text{ pour } k \in [1, p]$$

*Remarques.* Dans le cas où le coût est linéaire :

- on constate que la solution optimale est un sommet de la région réalisable ;
- même si tout un côté du polygone est optimal, ou toute une face du polyèdre, il est toujours possible de choisir une solution optimale correspondant à un sommet ;
- le long d'un côté du polygone, la variation de la valeur de la fonction objectif est constante. La valeur de la fonction objectif y est donc soit constante, soit strictement croissante, soit strictement décroissante. Cette remarque est primordiale : elle est à la base de l'algorithme du simplexe.

### 2.1.2 Forme canonique, forme standard

**Définition 12** (Forme canonique). Soient une matrice  $A$  de taille  $m \times n$ , un vecteur colonne  $b$  de taille  $m$ , et un vecteur ligne  $c$  de taille  $n$ . Le PL  $\max_{x/Ax \leq b; x \geq 0} (cx)$  est dit écrit sous *forme canonique*. Ce programme s'écrit sous la forme suivante :

$$\begin{cases} \max cx \\ Ax \leq b \\ x \geq 0 \end{cases}$$

**Algorithme 2** : Principe de l'algorithme du simplexe

---

Choisir un sommet  $\tilde{x}$  de la région réalisable ;  
 Soit  $\mathcal{A}$  l'ensemble des côtés passant par  $\tilde{x}$  ;  
**tant que**  $\exists c \in \mathcal{A}/z$  *croît le long de*  $c$  **faire**  
     déterminer l'extrémité  $\tilde{y}$  de  $c$  ;  
      $\tilde{x} \leftarrow \tilde{y}$  ;  
**fin**

---

La forme canonique est caractérisée par les propriétés suivantes :

- toutes les variables  $x_1, \dots, x_n$  sont positives ou nulles ;
- toutes les contraintes sont des inéquations de type *inférieure ou égal*.

**Définition 13** (Forme standard). Soient une matrice  $A$  de taille  $m \times n$ , un vecteur colonne  $b$  de taille  $m$ , et un vecteur ligne  $c$  de taille  $n$ . Le PL  $\max_{x/Ax=b; x \geq 0} (cx)$  est dit écrit sous forme standard.

Ce programme s'écrit sous la forme suivante :

$$\begin{cases} \max cx \\ Ax = b \\ x \geq 0 \end{cases}$$

La forme standard est caractérisée par les propriétés suivantes :

- toutes les variables  $x_1, \dots, x_n$  sont positives ou nulles ;
- toutes les contraintes sont des équations.

**Définition 14** (Programmes linéaires équivalents). Deux programmes linéaires sont dits équivalents si pour toute solution réalisable de l'un il existe une solution réalisable de l'autre telles que les valeurs de leurs fonctions objectif soient égales.

Lorsque deux programmes linéaires sont équivalents, on dit que l'un a été écrit sous forme de l'autre.

**Théorème 4.** *Il y a équivalence entre les programmes linéaires sous forme canonique et sous forme standard :*

1. *Tout PL sous forme standard peut s'écrire sous forme canonique.*
2. *Tout PL sous forme canonique peut s'écrire sous forme standard.*

*Démonstration.* de la partie 1 Soit un programme écrit sous forme standard. La  $i^e$  équation s'écrit :

$$A_i x = b_i$$

Cette équation peut s'écrire sous forme de deux inéquations :

$$A_i x \leq b_i \text{ et } A_i x \geq b_i$$

Ou encore :

$$A_i x \leq b_i \text{ et } -A_i x \leq -b_i$$

C'est-à-dire deux inéquations sous forme canonique. □

*Démonstration.* de la partie 2 Soit un programme linéaire écrit sous forme canonique. La  $i^e$  équation s'écrit :

$$A_i x \leq b_i \tag{2.2}$$

En ajoutant des variables supplémentaires  $\xi_i$ , cette équation peut s'écrire sous la forme suivante :

$$A_i x + \xi_i = b_i \text{ avec } \xi_i \geq 0 \quad (2.3)$$

C'est-à-dire une équation sous forme standard.  $\square$

**Définition 15** (Variable d'écart). La variable  $\xi_i$  de l'équation (2.3) est appelée variable d'écart. Elle sert à mesurer la différence entre les deux termes de l'inéquation (2.2).

**Théorème 5.** *Tout programme linéaire peut être écrit sous forme canonique ou sous forme standard.*

*Démonstration.* Grâce au théorème 4, il suffit de montrer que tout PL peut s'écrire sous forme canonique. Il nous faut donc montrer que tout PL peut s'écrire avec des inéquations *inférieure ou égale*, mettant en oeuvre des variables positives ou nulles, et une fonction objectif de type maximisation.

Le cas où le PL contient des équations est traité dans la démonstration du théorème 4. Il existe donc trois cas à examiner :

- Tout d'abord le cas où une contrainte est  $\geq$ . Dans ce cas, il suffit de prendre l'opposé de la contrainte.

$$A_i x \geq b_i \iff -A_i x \leq -b_i$$

- Examinons ensuite le cas où la fonction objectif est une minimisation. Dans ce cas, il faut maximiser l'opposé de cette fonction.

$$\min A_i x \iff \max -A_i x$$

- Il reste alors à examiner le cas où les variables  $x_i$  peuvent prendre des valeurs aussi bien positives que négatives<sup>1</sup>. On peut alors poser

$$x_i = x'_i - x''_i \text{ avec } x'_i \geq 0 \text{ et } x''_i \geq 0$$

Alors, les deux problèmes suivants sont équivalents :

$$\begin{aligned} (P) \quad & \begin{cases} \max cx \\ Ax \leq b & x_i \in \mathbb{R}, \forall j \neq i, x_j \geq 0 \end{cases} \\ (P') \quad & \begin{cases} \max c_i x'_i - c_i x''_i + c_j x_j \\ A_i x'_i - A_i x''_i + A_j x_j \leq b & x'_i, x''_i, x_j \geq 0 \end{cases} \end{aligned}$$

De plus, le problème  $(P')$  est sous forme canonique.  $\square$

### 2.1.3 Programmes linéaires duaux

Il est possible d'associer à tout programme linéaire un autre programme appelé dual. Il existe de nombreuses relations entre un programme linéaire et son dual, et en particulier la résolution de l'un des deux permet de trouver une solution à l'autre. Ainsi, un programme linéaire et son dual représentent deux facettes du même problème.

---

1. Le cas où  $x_i \in \mathbb{R}^-$  en est un cas particulier.

**Définition 16** (Programme dual). Soit le PL suivant :

$$\begin{cases} \max cx \\ Ax \leq b \quad x \geq 0 \end{cases}$$

On appelle *dual* de ce programme le programme suivant :

$$\begin{cases} \min yb \\ yA \geq c \quad y \geq 0 \end{cases}$$

Dans ce cas,  $y$  est un vecteur ligne. Le programme dual peut également être représenté par :

$$\begin{cases} \min {}^tby' \\ {}^tAy' \geq {}^tc \quad y' \geq 0 \end{cases}$$

avec  $y'$  un vecteur colonne.

*Exemple 9* (Système dual). Le dual du système représenté dans l'exemple 8 est le suivant :

$$\begin{cases} \min & 8y_1 + 7y_2 + 3y_3 \\ \text{s.c.} & \\ & 2y_1 + y_2 \geq 4 \\ & y_1 + 2y_2 + y_3 \geq 5 \end{cases}$$

**Théorème 6.** *Le dual du dual d'un programme linéaire est le primal.*

*Démonstration.* Soit un PL :

$$(P) \begin{cases} \max cx \\ Ax \leq b \quad x \geq 0 \end{cases}$$

Son dual est par définition :

$$(D) \begin{cases} \min {}^tby' \\ {}^tAy' \geq {}^tc \quad y' \geq 0 \end{cases}$$

Et le dual du dual est alors :

$$(P') \begin{cases} \min -cx \\ -Ax \geq -b \quad x' \geq 0 \end{cases}$$

En multipliant  $(P')$  par -1 et en posant  $z = -z'$ , on retrouve le programme initial.  $\square$

Le passage d'un programme à son dual se fait en utilisant les transformations suivantes :

Primal	Dual
Fonction objectif à maximiser	Fonction objectif à minimiser
$i^{\text{e}}$ contrainte $\geq$	$i^{\text{e}}$ variable $\leq 0$
$i^{\text{e}}$ contrainte $\leq$	$i^{\text{e}}$ variable $\geq 0$
$i^{\text{e}}$ contrainte $=$	$i^{\text{e}}$ variable $= 0$
$j^{\text{e}}$ variable $\geq 0$	$j^{\text{e}}$ contrainte $\geq$
$j^{\text{e}}$ variable $\leq 0$	$j^{\text{e}}$ contrainte $\leq$
$j^{\text{e}}$ variable $= 0$	$j^{\text{e}}$ contrainte $=$

**Théorème 7.** *Soient  $(P)$  et  $(D)$  deux programmes linéaires duaux :*

$$(P) \begin{cases} \max cx \\ Ax \leq b \quad x \geq 0 \end{cases} \quad (D) \begin{cases} \min yb \\ yA \geq c \quad y \geq 0 \end{cases} \quad (2.4)$$

Pour tout couple de solution  $\bar{x}, \bar{y}$  du système et de son dual, on a la relation :  $c\bar{x} \leq \bar{y}b$ .



**Corollaire 1.** *Pour tout couple de solution  $\bar{x}, \bar{y}$  d'un programme linéaire et de son dual, si  $c\bar{x} \geq \bar{y}b$ , alors  $c\bar{x} = \bar{y}b$  et  $\bar{x}, \bar{y}$  sont les solutions optimales du programme et de son dual.*

*Démonstration.* S'il existait une solution  $\bar{x}'$  de (P) avec  $c\bar{x}' > c\bar{x}$  on aurait alors  $c\bar{x}' > \bar{y}b$  ce qui est impossible selon le théorème 7.  $\square$

Le corollaire 1 permet de déterminer l'optimalité d'une solution proposée.

*Exemple 10* (Utilisation du programme linéaire dual). Nous avons montré graphiquement que la solution  $x_1 = 3, x_2 = 2$  était réalisable dans le PL de l'exemple 8. La fonction objectif vaut alors  $z = 4x_1 + 5x_2 = 22$ .

Considérons maintenant le programme dual, défini dans l'exemple 9. On vérifie facilement que  $y_1 = 1, y_2 = 2, y_3 = 0$  est une solution réalisable et que la valeur de la fonction objectif associée est :  $w = 8y_1 + 7y_2 + 3y_3 = 22$ . Selon le corollaire, nous avons alors une paire de solutions optimales pour le programme linéaire et son dual.

#### 2.1.4 Théorème de dualité, théorème des écarts complémentaires

**Théorème 8** (Dualité). *Soient (P) et (D) deux programmes linéaires duaux :*

$$(P) \begin{cases} \max & cx \\ & Ax \leq b \quad x \geq 0 \end{cases} \quad (D) \begin{cases} \min & yb \\ & yA \geq c \quad y \geq 0 \end{cases}$$

1. Si (P) et (D) admettent une solution réalisable, ils ont tous les deux une solution optimale, et les valeurs des fonctions objectifs sont égales.
2. Si l'un des deux admet une classe de solutions réalisables pour lesquelles la fonction objectif n'est pas bornée (supérieurement pour (P) et inférieurement pour (D)), alors l'autre n'a pas de solution réalisable.
3. Si (P) (resp. (D)) a une solution réalisable mais pas (D) (resp. (P)), alors (P) (resp. (D)) admet une classe de solutions réalisables pour lesquelles la fonction objectif n'est pas bornée supérieurement (resp. inférieurement).
4. Il se peut que ni (P) ni (D) n'ait de solution réalisable.

**Définition 17.** Soit  $\bar{x}$  une solution réalisable de (P). La  $i^e$  contrainte sera dite :

- serrée si  $A_i\bar{x} = b_i$
- lâche si  $A_i\bar{x} < b_i$

**Théorème 9** (écarts complémentaires). *Une condition nécessaire et suffisante pour qu'un couple de solutions réalisables des programmes linéaires duaux (P) et (D) soit un couple de solutions optimales est que :*

- si une contrainte de l'un des programmes linéaires est lâche, la variable correspondante du dual est nulle ;
- si une variable de l'un des programmes linéaires est positive, la contrainte correspondante du dual est serrée.

*Démonstration.* En ajoutant des variables d'écart, les programmes linéaires (P) et (D) deviennent :

$$(P) \begin{cases} \max & cx \\ & Ax + \xi = b \\ & x, \xi \geq 0 \end{cases} \quad (D) \begin{cases} \min & yb \\ & yA - \eta = c \\ & y, \eta \geq 0 \end{cases}$$

Soient  $\bar{x}, \bar{\xi}$  une solution réalisable de (P), et  $\bar{y}, \bar{\eta}$  une solution réalisable de (D). Multiplions la  $i^e$  contrainte de (P) par la variable duale correspondante  $\bar{y}^i$ , et calculons la somme pour  $i \in \{1, \dots, m\}$ . On obtient :

$$\bar{y}A\bar{x} + \bar{y}\bar{\xi} = \bar{y}b \quad (2.5)$$

Multiplions la  $j^{\text{e}}$  contrainte de  $(D)$  par la variable correspondante de  $(P)$   $\bar{x}_j$  et calculons la somme pour  $j \in \{1, \dots, n\}$ . On obtient :

$$\bar{y}A\bar{x} - \bar{\eta}\bar{x} = c\bar{x} \quad (2.6)$$

(2.5) – (2.6)  $\Rightarrow$

$$\bar{y}\bar{\xi} + \bar{\eta}\bar{x} = \bar{y}b - c\bar{x} \quad (2.7)$$

*Condition nécessaire* : si  $\bar{x}$  est solution optimale de  $(P)$  et  $\bar{y}$  solution optimale de  $(D)$ , d'après le théorème 8 et d'après l'équation (2.7) :

$$\bar{y}\bar{\xi} + \bar{\eta}\bar{x} = \sum_{i=1}^m \bar{y}^i \bar{\xi}_i + \sum_{j=1}^n \bar{\eta}^j \bar{x}_j = 0$$

Il s'agit d'une somme de termes positifs ou nuls. Pour que la somme soit nulle, il est donc nécessaire que chaque terme le soit, et les équations suivantes doivent donc être respectées :

$$\begin{cases} \bar{y}^i > 0 \Rightarrow \bar{\xi}_i = 0 \\ \bar{\xi}_i > 0 \Rightarrow \bar{y}^i = 0 \\ \bar{\eta}^j > 0 \Rightarrow \bar{x}_j = 0 \\ \bar{x}_j > 0 \Rightarrow \bar{\eta}^j = 0 \end{cases} \quad (2.8)$$

*Condition suffisante* : si les conditions énoncées dans le théorème sont satisfaites, les implications (2.8) le sont également. Donc, d'après (2.7)  $\bar{y}b - c\bar{x} = 0$ , c'est-à-dire que les valeurs des fonctions objectif de  $(P)$  et  $(D)$  sont égales. Selon le corollaire 1 (p. 25), les solutions  $\bar{x}$  et  $\bar{y}$  sont donc optimales.  $\square$

Le théorème des écarts complémentaires est très important : il permet de montrer l'optimalité d'une solution ou de prouver qu'une solution est bien optimale sans avoir à appliquer l'algorithme.

*Exemple 11.* Soient les deux programmes linéaires duaux :

$$(P) \begin{cases} \min & x_1 & +x_2 \\ & 3x_1 & +x_2 \geq 4 \\ & x_1 & +4x_2 \geq 5 \\ & x_1, & x_2 \geq 0 \end{cases} \quad (D) \begin{cases} \max & 4y_1 & +5y_2 \\ & 3y_1 & +y_2 \leq 1 \\ & y_1 & +4y_2 \leq 1 \\ & y_1, & y_2 \geq 0 \end{cases}$$

On estime que  $\bar{x}_1 = \bar{x}_2 = 1$  est une solution optimale de  $(P)$ . Étant donné que  $\bar{x}_1 > 0$  et  $\bar{x}_2 > 0$ , il suffit pour s'en assurer de résoudre le système linéaire :

$$\begin{cases} 3y_1 + y_2 = 1 \\ y_1 + 4y_2 = 1 \end{cases}$$

La solution de ce système est :  $\bar{y}_1 = \frac{3}{11}, \bar{y}_2 = \frac{2}{11}$ . Donc  $\bar{y}_1 > 0$  et  $\bar{y}_2 > 0$ , et  $\bar{x}_1 = \bar{x}_2 = 1$  est une solution optimale de  $(P)$ .

## 2.2 Algorithme du simplexe

L'algorithme du simplexe a été découvert par G. B. Dantzig en 1947. Il marque une étape décisive dans l'histoire de la recherche opérationnelle et représente un outil très efficace pour résoudre les problèmes modélisés sous forme de programmes linéaires. Le nombre de problèmes pouvant se modéliser sous forme de programme linéaire a grandement contribué au succès de cet algorithme.

*Remarque.* Nous distinguerons la méthode du simplexe de l'algorithme du simplexe, ce dernier supposant que le problème est écrit sous forme standard par rapport à une base réalisable.

L'algorithme du simplexe permet de déterminer une solution optimale d'un PL si elle existe, ou l'absence de solution le cas échéant.

### 2.2.1 Définitions

*Exemple 12* (Forme standard). Considérons l'exemple 8. Il s'écrit sous forme standard de la manière suivante :

$$\left\{ \begin{array}{rclclclcl} \max & 4x_1 & + & 5x_2 & & & & \\ & 2x_1 & + & x_2 & + & x_3 & & = 8 \\ & x_1 & + & 2x_2 & & & + & x_4 = 7 \\ & & & x_2 & & & + & x_5 = 3 \end{array} \right.$$

**Définition 18** (Variables de base, variables hors base). Soit un PL sous forme standard :  $Ax = b$ , avec  $A$  une matrice de  $m$  lignes et  $n$  colonnes telle que  $\text{Rang}(A) = m$ .

Il existe alors un ensemble  $B$  d'indices tels que la famille des colonnes  $A^j$  de  $A$  avec  $j \in B$  soit libre. La matrice  $A_B$  formée de ces colonnes est donc inversible.

On dit que l'ensemble des indices  $B$  est une base. Les  $m$  variables associées sont appelées *variables de base*. Les  $n - m$  variables restantes, qui peuvent être arbitrairement fixées à 0, sont appelées *variables hors base*.

Dans l'exemple 12, les variables hors base sont  $x_1$  et  $x_2$ , et les variables de base sont  $x_3$ ,  $x_4$  et  $x_5$ .

**Définition 19** (Solution de base). On appelle solution de base d'un programme linéaire ( $P$ ) une solution pour laquelle les variables hors base ont été fixées à 0. La solution de base est dite réalisable si elle vérifie les contraintes de positivité.

*Exemple 13* (Solution de base réalisable). Dans l'exemple 12, la solution de base  $x_1 = 0, x_2 = 0, x_3 = 8, x_4 = 7, x_5 = 3$  est réalisable.

*Remarque.* La notion géométrique de sommet du polyèdre correspond à la notion de solution de base réalisable.

**Définition 20** (Solutions de base adjacentes). Deux solutions de base sont dites adjacentes si toutes les variables de base de la première sauf une sont variables de base de la seconde.

*Exemple 14* (Solutions de base adjacentes). Dans l'exemple 12, les solutions de base suivantes sont adjacentes :

$$\begin{aligned} (x_1 = 0, x_2 = 0, x_3 = 8, x_4 = 7, x_5 = 3) \\ (x_1 = 0, x_2 = 3, x_3 = 5, x_4 = 1, x_5 = 0) \end{aligned}$$

*Remarque.* La notion géométrique de sommets du polyèdre adjacents correspond à la notion de solutions de base adjacentes.

### 2.2.2 Initialisation de l'algorithme

Afin d'initialiser l'algorithme, il faut une solution de base réalisable. Dans les cas où l'origine fait partie de la région réalisable (ce qui est très souvent le cas), il suffit de la prendre comme point de départ. Ainsi, dans l'exemple 12, il est possible de partir de  $(x_1, x_2) = (0, 0)$ , et la solution de base réalisable de départ est alors :  $(0, 0, 8, 7, 3)$ .

Dans le cas où l'origine ne fait pas partie de la région réalisable, il faut déterminer une solution réalisable par une phase préliminaire appelée phase I de la méthode du simplexe (voir section 2.4.1 p. 31).

### 2.2.3 Itération de l'algorithme

Une itération de l'algorithme du simplexe consiste à se déplacer d'un sommet à un sommet adjacent en améliorant la fonction objectif. Partant d'une solution de base réalisable, nous allons donc nous déplacer vers une autre solution de base réalisable. Pour cela, il faut choisir une direction de déplacement, c'est-à-dire choisir une variable hors base à faire entrer en base.

Le choix de la variable à faire entrer dans la base doit se faire de manière à augmenter le plus possible la fonction objectif. On choisira donc la variable hors base dont le coefficient dans la fonction objectif est le plus élevé. Pour cela, il faut bien entendu que la fonction objectif soit définie uniquement en fonction des variables hors base.

Le choix de la variable à sortir de la base doit se faire en respectant les contraintes de positivité. On va donc choisir la première variable de base à s'annuler.

*Exemple 15* (Changement de variable de base). Toujours dans l'exemple 12, on choisit de faire entrer  $x_2$  en base. Les contraintes de positivité à respecter sont alors :

$$\begin{cases} (L1) & x_3 = 8 - x_2 \geq 0 \\ (L2) & x_4 = 7 - 2x_2 \geq 0 \\ (L3) & x_5 = 3 - x_2 \geq 0 \end{cases}$$

$$\Leftrightarrow \begin{cases} x_3 = 1(8 - x_2) \geq 0 \\ x_4 = 2(3.5 - x_2) \geq 0 \\ x_5 = 1(3 - x_2) \geq 0 \end{cases} \quad \text{On a ainsi}$$

$$x_2 \leq \min(8, 3.5, 3) = 3$$

La variable sortante choisie est alors  $x_5$  qui s'annule pour  $x_2 = 3$ , ce qui est la contrainte la plus forte.

Le calcul du nouveau sommet se fait par combinaisons linéaires des équations du programme linéaire. Il faut éliminer la variable de base de toutes les équations, sauf celle qui définit la nouvelle variable de base. Dans notre exemple, il faut éliminer  $x_2$  des équations (L1) et (L2) en utilisant l'équation (L3). Le système devient alors :

$$\begin{cases} \max & 4x_1 & & & - & 5x_5 & +15 \\ & 2x_1 & & + & x_3 & & - & x_5 & = & 5 \\ & x_1 & & & & + & x_4 & - & 2x_5 & = & 1 \\ & & x_2 & & & & + & x_5 & = & 3 \end{cases}$$

Les variables hors base étant  $x_1$  et  $x_5$ , la solution de base s'écrit donc :

$$(x_1, x_2, x_3, x_4, x_5) = (0, 3, 5, 1, 0) \quad (2.9)$$

### 2.2.4 Critère d'arrêt

Après la première itération, la fonction objectif s'écrit :  $z = 4x_1 - 5x_5 + 15$ . Nous remarquons qu'elle peut être améliorée en entrant  $x_1$  en base.

Le critère d'arrêt est alors le suivant : la solution de base courante est optimale si tous les coefficients objectifs sont négatifs lorsque  $z$  est exprimée uniquement en fonction des variables hors base.

*Exemple 16* (Arrêt de l'algorithme du simplexe). Poursuivons l'exemple 12. Il est possible d'entrer  $x_1$  en base. La contrainte de positivité la plus forte vient de l'équation :

$$x_1 + x_4 - 2x_5 = 1$$

Le système devient alors :

$$\begin{cases} \max & & & - & 4x_4 & + & 3x_5 & +19 \\ & & x_3 & - & 2x_4 & + & 3x_5 & = & 3 \\ x_1 & & & + & x_4 & - & 2x_5 & = & 1 \\ & x_2 & & & & + & x_5 & = & 3 \end{cases}$$

La solution de base vaut alors :  $(x_1, x_2, x_3, x_4, x_5) = (1, 3, 3, 0, 0)$

Cette solution n'est toujours pas optimale. Il est encore possible de faire entrer  $x_5$  en base. L'équation la plus contraignante est  $x_3 - 2x_4 + 3x_5 = 3$ .

Le système devient alors :

$$\begin{cases} \max & -x_3 - 2x_4 & +22 \\ & x_3 - 2x_4 + 3x_5 & = 3 \\ x_1 & + \frac{2}{3}x_3 - \frac{1}{3}x_4 & = 3 \\ & x_2 - \frac{1}{3}x_3 + \frac{2}{3}x_4 & = 2 \end{cases}$$

Il est cependant plus pratique de normaliser les équations par rapport aux variables de base. Dans notre cas, il faut diviser la première équation par 3 :

$$\begin{cases} \max & -\frac{1}{3}x_3 - \frac{2}{3}x_4 & +22 \\ & \frac{1}{3}x_3 - \frac{2}{3}x_4 + x_5 & = 1 \\ x_1 & + \frac{2}{3}x_3 - \frac{1}{3}x_4 & = 3 \\ & x_2 - \frac{1}{3}x_3 + \frac{2}{3}x_4 & = 2 \end{cases}$$

La solution de base optimale vaut alors :  $(x_1, x_2, x_3, x_4, x_5) = (3, 2, 0, 0, 1)$

Partant de l'origine, le trajet suivi par l'algorithme du simplexe peut se représenter sur la figure 2.2 :

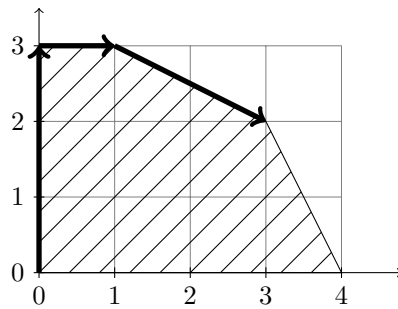


FIGURE 2.2 – Algorithme du simplexe

### 2.3 Algorithme du simplexe en tableaux

Une méthode plus pratique pour utiliser l'algorithme du simplexe consiste à présenter le système d'équations sous forme de tableau. Ce tableau contient les coefficients de la fonction objectif, les coefficients des variables dans le membre gauche des contraintes, et les coefficients du membre de droite des contraintes.

*Exemple 17* (Algorithme du simplexe en tableaux). Reprenons l'exemple 12 :

$$\begin{cases} \max & 4x_1 + 5x_2 \\ & 2x_1 + x_2 + x_3 & = 8 \\ & x_1 + 2x_2 + x_4 & = 7 \\ & & x_2 + x_5 & = 3 \end{cases}$$

Le système s'écrit sous forme de tableau de la manière suivante :

z	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	
1	-4	-5	0	0	0	0
0	2	1	1	0	0	8
0	1	2	0	1	0	7
0	0	1	0	0	1	3

### 2.3.1 Lecture du tableau

- les valeurs du membre de droite donnent les valeurs courantes des variables de base<sup>2</sup> ;
- la première ligne donne l'opposé des coefficients objectifs ;
- le dernier coefficient de la première ligne donne la valeur de l'objectif.
- la sous-matrice identité (ou simplement diagonale) identifie les variables de base. Ici ce sont  $x_3$ ,  $x_4$  et  $x_5$ .

### 2.3.2 Première itération

On choisit la colonne pour laquelle le coefficient de la première ligne est le plus négatif. On le représente en encadrant la colonne :

z	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	
1	-4	-5	0	0	0	0
0	2	1	1	0	0	8
0	1	2	0	1	0	7
0	0	1	0	0	1	3

Il faut ensuite choisir la ligne correspondant à la variable sortante. On choisit la ligne pour laquelle le rapport entre le coefficient de la colonne de droite et le coefficient de la colonne sélectionnée est le plus petit, tout en restant positif.

z	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	
1	-4	-5	0	0	0	0
0	2	1	1	0	0	8
0	1	2	0	1	0	7
0	0	1	0	0	1	3

Il faut enfin éliminer la variable sélectionnée par combinaison linéaire de la ligne choisie. Le système devient alors :

z	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	
1	-4	0	0	0	5	15
0	2	0	1	0	-1	5
0	1	0	0	1	-2	1
0	0	1	0	0	1	3

### 2.3.3 Itérations suivantes

il faut maintenant poursuivre les itérations jusqu'à ce que tous les coefficients de la première ligne soient positifs. On obtient finalement le système suivant :

z	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	
1	0	0	1	2	0	22
0	0	0	1/3	-2/3	1	1
0	1	0	2/3	-1/3	0	3
0	0	1	-1/3	2/3	0	2

La solution de base optimale vaut :  $(x_1, x_2, x_3, x_4, x_5) = (3, 2, 0, 0, 1)$ , et la valeur de la fonction objectif est 22.

2. Au cours des itérations de l'algorithme du simplexe, toutes ces valeurs doivent donc rester positives ou nulles.

## 2.4 Traitement des cas particuliers

### 2.4.1 Initialisation de l'algorithme

L'algorithme du simplexe fonctionne à partir d'une solution de base réalisable. Nous avons considéré le cas général pour lequel une solution est immédiatement disponible (par exemple l'origine). Nous allons maintenant traiter les cas particuliers pour lesquels une telle solution n'existe pas.

Supposons qu'au moins un des coefficients du membre de droite soit négatif. Dans ce cas, il n'est pas sûr que le problème ait une solution réalisable, et si elle existe il faut la déterminer.

La réponse à ces deux questions passe par la résolution d'un système auxiliaire. Considérons par exemple le PL suivant :

$$\left\{ \begin{array}{llll} \max & x_1 & -x_2 & +x_3 \\ & 2x_1 & -x_2 & +2x_3 \leq 4 \\ & -2x_1 & +3x_2 & -x_3 \geq 5 \\ & x_1 & -x_2 & +2x_3 \geq 1 \\ & x_1 & , x_2 & , x_3 \geq 0 \end{array} \right.$$

qui peut s'écrire sous forme standard :

$$\left\{ \begin{array}{llllllll} \max & x_1 & -x_2 & +x_3 & & & & \\ & 2x_1 & -x_2 & +2x_3 & +x_4 & & & = 4 \\ & 2x_1 & -3x_2 & +x_3 & & +x_5 & & = -5 \\ & -x_1 & +x_2 & -2x_3 & & & +x_6 & = -1 \\ & x_1 & , x_2 & , x_3 & , x_4 & , x_5 & , x_6 & \geq 0 \end{array} \right.$$

Il faut ensuite rendre la partie droite du système positive en lui ajoutant une quantité positive  $x_0$ , ou en retranchant cette quantité à sa partie gauche.

$$\left\{ \begin{array}{llllllll} \max & x_1 & -x_2 & +x_3 & & & & \\ & 2x_1 & -x_2 & +2x_3 & +x_4 & & -x_0 & = 4 \\ & 2x_1 & -3x_2 & +x_3 & & +x_5 & -x_0 & = -5 \\ & -x_1 & +x_2 & -2x_3 & & & +x_6 & -x_0 = -1 \end{array} \right.$$

L'objectif est, dans un premier temps, d'éliminer cette variable  $x_0$ . Pour cela, il faut la minimiser (ou maximiser son opposé). Nous ajoutons alors la fonction objectif suivante à notre système :  $\max -x_0$ .

Le problème obtenu est appelé *phase I de l'algorithme du simplexe*.

Une fois le problème reformulé, il faut déterminer une solution de base réalisable. Une opération de pivotage non standard est pour cela nécessaire : partant d'une base de départ *non réalisable* formée des variables d'écart, on fait entrer  $x_0$  en base en la permutant avec la variable de base la plus négative. Il est alors possible de résoudre le problème auxiliaire par l'algorithme du simplexe.

Les différents états du PL sont décrits par les tableaux suivants :

w	$x_0$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	
1	1	0	0	0	0	0	0	0
0	-1	2	-1	2	1	0	0	4
0	-1	2	-3	1	0	1	0	-5
0	-1	-1	1	-2	0	0	1	-1

Première itération :

w	$x_0$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	
1	0	2	-3	1	0	1	0	-5
0	0	0	2	1	1	-1	0	9
0	1	-2	3	-1	0	-1	0	5
0	0	-3	4	-3	0	-1	1	4

On peut remarquer que la valeur de la fonction objectif a diminué lors de cette opération. Notons que ceci ne peut se produire que dans le cas de l'opération de pivotage non standard.

Deuxième itération :

w	$x_0$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	
1	0	-1/4	0	-5/4	0	1/4	3/4	-2
0	0	3/2	0	5/2	1	-1/2	-1/2	7
0	1	1/4	0	5/4	0	-1/4	-3/4	2
0	0	-3/4	1	-3/4	0	-1/4	1/4	1

Troisième itération :

w	$x_0$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	
1	1	0	0	0	0	0	0	0
0	-2	1	0	0	1	0	1	3
0	4/5	1/5	0	1	0	-1/5	-3/5	8/5
0	3/5	-3/5	1	0	0	-2/5	-1/5	11/5

*Remarques.*

- le programme linéaire de la *phase I* est toujours réalisable et sa solution est toujours bornée. Il s'agit en effet de minimiser une variable positive ;
- si la valeur de la fonction objectif à l'issue de la phase I est non nulle, le problème initial n'est pas réalisable ;
- sinon, il admet une solution et on dispose d'une solution de base réalisable ;
- la valeur de la fonction objectif à l'issue de la phase I ne peut *jamaïs* être positive (la fonction objectif est  $w = -x_0$ , et  $x_0 \geq 0$ ).

Dans le cas où le problème admet une solution, on peut supprimer  $x_0$  du système, reprendre la fonction objectif initiale et commencer la résolution.

z	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	
1	-1	1	-1	0	0	0	0
0	1	0	0	1	0	1	3
0	1/5	0	1	0	-1/5	-3/5	8/5
0	-3/5	1	0	0	-2/5	-1/5	11/5

Il faut maintenant exprimer la fonction objectif en fonction des variables hors base, c'est-à-dire en fonction de  $x_1, x_5, x_6$ . Le résultat, obtenu en ajoutant la deuxième contrainte et en retranchant la troisième à la fonction objectif, est le suivant :

z	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	
1	-1/5	0	0	0	1/5	-2/5	-3/5
0	1	0	0	1	0	1	3
0	1/5	0	1	0	-1/5	-3/5	8/5
0	-3/5	1	0	0	-2/5	-1/5	11/5

Il est maintenant possible de résoudre la problème. On obtient le résultat suivant :

z	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	
1	1/5	0	0	2/5	1/5	0	3/5
0	1	0	0	1	0	1	3
0	4/5	0	1	3/5	-1/5	0	17/5
0	-2/5	1	0	1/5	-2/5	0	14/5

La solution optimale vaut donc :  $(x_1, x_2, x_3, x_4, x_5, x_6) = (0, \frac{14}{5}, \frac{17}{5}, 0, 0, 3)$  et la valeur de la fonction objectif est  $z = \frac{3}{5}$ .



Exemple 18 (Problème non réalisable). Considérons le PL suivant :

$$\begin{cases} \max & 3x_1 & +x_2 \\ & -x_1 & +x_2 \geq 1 \\ & x_1 & +x_2 \geq 3 \\ & 2x_1 & +x_2 \leq 2 \\ & x_1 & , x_2 \geq 0 \end{cases}$$

qui se met sous forme standard :

$$\begin{cases} \max & 3x_1 & +x_2 \\ & x_1 & -x_2 & +x_3 & & = -1 \\ & -x_1 & -x_2 & & +x_4 & = -3 \\ & 2x_1 & +x_2 & & & +x_5 = 2 \\ & x_1 & , x_2 & , x_3 & , x_4 & , x_5 \geq 0 \end{cases}$$

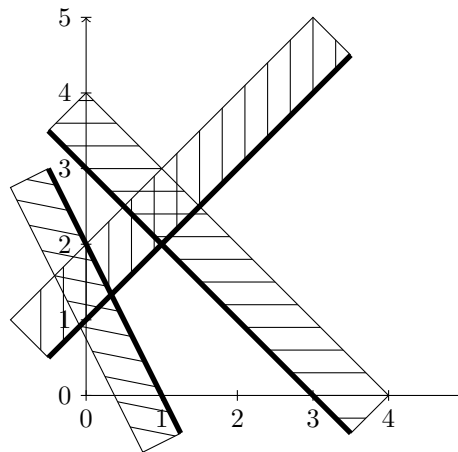


FIGURE 2.3 – Problème sans solution

Après les calculs de la *phase I*, on obtient le système suivant :

w	$x_0$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	
1	0	1/2	0	0	1/2	1/2	-1/2
0	0	2	0	1	-1	0	2
0	1	-1/2	0	0	-1/2	-1/2	1/2
0	0	-3/2	1	0	-1/2	1/2	5/2

La *phase I* se termine alors que  $w \neq 0$ , donc le problème n'admet pas de solution. La représentation graphique (figure 2.3) permet également de se rendre compte de l'absence de solution.

#### 2.4.2 Solution non bornée

La variable sortante choisie est celle qui produit la contrainte la plus forte sur la variable entrante choisie. Cette règle de sélection est ambiguë et peut donner lieu à plusieurs ou aucun candidat.

Ceci est en particulier le cas pour les programmes linéaires admettant des solutions non bornées.

*Exemple 19* (Programme linéaire non borné). Considérons l'exemple suivant dont la région réalisable est représentée en figure 2.4.

$$\begin{cases} \max & 2x_1 & +x_2 & & & \\ & x_1 & -2x_2 & +x_3 & & = 2 \\ & -2x_1 & +x_2 & & +x_4 & = 2 \\ & x_1 & & ,x_2 & ,x_3 & ,x_4 \geq 0 \end{cases}$$

Après la première itération, on obtient le système suivant :

z	$x_1$	$x_2$	$x_3$	$x_4$	
1	0	-5	2	0	4
0	1	-2	1	0	2
0	0	-3	2	1	6

La variable choisie pour entrer en base est obligatoirement  $x_2$ , mais tous les coefficients de sa colonne sont négatifs. Aucune des variables de base n'induit de contrainte pour  $x_2$ . Cette variable peut donc croître indéfiniment.

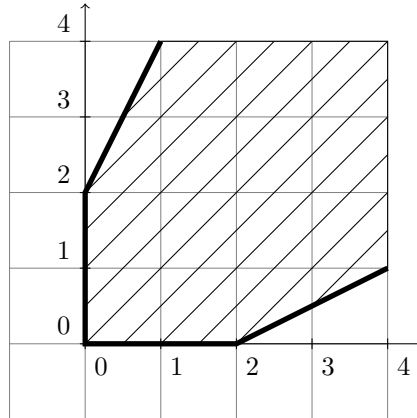


FIGURE 2.4 – Problème non borné

### 2.4.3 Système dégénéré

S'il existe plusieurs candidats pouvant sortir de la base, la solution de base peut dégénérer.

*Exemple 20* (Dégénérescence). Considérons le système suivant :

z	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	
1	-2	1	-8	0	0	0	0
0	0	0	2	1	0	0	1
0	2	-4	6	0	1	0	3
0	-1	3	4	0	0	1	2

La variable choisie pour entrer en base est  $x_3$ , mais les trois variables  $x_4, x_5, x_6$  induisent la même contrainte sur  $x_3$ . Ces trois variables peuvent donc indifféremment sortir de la base. Choisissons par exemple  $x_4$ . Le système devient alors :

z	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	
1	-2	1	0	4	0	0	4
0	0	0	1	1/2	0	0	1
0	2	-4	0	-3	1	0	0
0	-1	3	0	-2	0	1	0

Les variables de base  $x_5$  et  $x_6$  sont alors de valeur nulle. Une solution de base avec des variables nulles est dite *dégénérée*. Lors de l'itération suivante,  $x_1$  entre en base et  $x_5$  limite sa valeur à 0. Donc, la valeur de  $x_1$  et celle de la fonction objectif restent identiques.

Des itérations permettant de changer de base sans changer la valeur de la solution de base sont appelées *itérations dégénérées*. En général, ce problème est passager, c'est-à-dire qu'après quelques itérations dégénérées la valeur de la fonction objectif du système recommence à croître.

#### Problèmes de cycles

Dans le cas où le programme linéaire contient des solutions de base dégénérées, il existe des itérations (dégénérées) pour lesquelles la valeur de la fonction objectif n'augmente pas. Il est alors possible que l'algorithme ne se termine pas.

Il existe différentes règles dans le choix des variables qui permettent d'éviter ce phénomène, comme par exemple la *règle de Bland* : choisir comme variable sortante la première de coefficient négatif, et comme variable entrante la première qui peut sortir (Bland [1977]). Pour de plus amples informations sur cette règle, il est également possible de se reporter à Avis et Chvátal [1978].

*Remarque.* Notons que le phénomène de cycle est extrêmement rare et que dans la pratique, il est possible de l'ignorer.

#### 2.4.4 Implantation et performances de l'algorithme du simplexe

L'algorithme du simplexe est relativement facile à implanter dans la plupart des langages de programmation. Seul le traitement exhaustif des cas particuliers donne un aspect rébarbatif à sa programmation.

Notons que l'algorithme du simplexe fournit une méthode très efficace de résolution des problèmes linéaires et qu'il permet de résoudre des problèmes contenant des milliers de variables et des milliers de contraintes.

#### Complexité

Klee et Minty ont montré dans Klee et Minty [1972] qu'il était possible de construire des problèmes pour lesquels la méthode du simplexe nécessite l'examen d'un nombre de sommets dépendant exponentiellement de la taille du problème<sup>3</sup>. La complexité de l'algorithme du simplexe dans le pire des cas est donc exponentielle.

Notons cependant que l'algorithme du simplexe est en moyenne très efficace sur les problèmes d'origine pratique. Dans Dantzig [1963], Dantzig montre statistiquement que l'algorithme nécessite en moyenne entre  $m$  et  $3m$  itérations ( $m$  étant le nombre de contraintes). Par la suite, des analyses probabilistes menées par de nombreux auteurs (comme Borgwardt dans Borgwardt [1982]) ont permis d'établir des estimations polynomiales du nombre moyen d'opérations de la méthode du simplexe.

---

3. Nombre de variables et nombre d'équations.



## Parcours de graphes

### Sommaire

<b>3.1 Généralités</b>	<b>37</b>
3.1.1 Historique	37
3.1.2 Définitions	38
3.1.3 Cycles eulériens et hamiltoniens	41
3.1.4 Représentation des graphes	42
<b>3.2 Parcours d'un graphe</b>	<b>44</b>
3.2.1 Parcours en largeur	44
3.2.2 Parcours en profondeur	45
3.2.3 Composantes fortement connexes	45
3.2.4 Sommets d'articulation, ponts, et k-connexité	47
<b>3.3 Plus court chemin dans un graphe</b>	<b>48</b>
3.3.1 Plus court chemin à origine unique	48
3.3.2 Plus court chemin pour tout couple de sommet	53
<b>3.4 Ordonnancement</b>	<b>56</b>
3.4.1 Méthode PERT	57
3.4.2 Méthode MPM	58
3.4.3 Exemple	59
3.4.4 Intervention de modifications	63

## 3.1 Généralités

**N**IENT que la notion de graphe ait été formalisée très récemment (au cours du XX<sup>e</sup> siècle), elle est aujourd'hui indispensable dans de nombreux domaines (informatique théorique et appliquée, optimisation). La théorie des graphes représente une part importante de la recherche opérationnelle. Elle permet de traiter de nombreux problèmes de R.O., et en particulier les problèmes de type combinatoire. Elle a été introduite en France par Claude Berge [Berge, 1967].

### 3.1.1 Historique

Un des premiers résultats de la théorie des graphes est dû à Leonhard Euler et a été publié dans un article se rapportant aux sept ponts de Königsberg en 1736. Le problème des sept ponts de Königsberg est un problème mathématique historique, qui fit la célébrité de la ville de Königsberg (aujourd'hui Kaliningrad ou Калининград en russe). Sa résolution fut recherchée par ses habitants tout au long du XVIII<sup>e</sup> siècle.

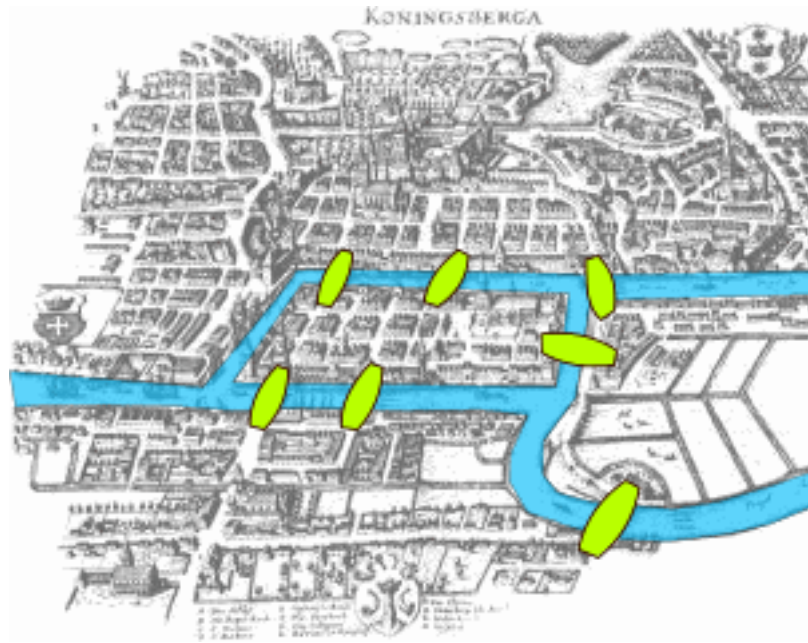


FIGURE 3.1 – Königsberg et ses ponts

Le problème est le suivant (la ville de Königsberg est représentée figure 3.1) : *étant donné que la ville est construite sur deux îles reliées au continent par six ponts, et entre elles par un pont, trouver un chemin quelconque permettant, à partir d'un point de départ au choix, de passer une et une seule fois par chaque pont, et de revenir à son point de départ, étant entendu qu'on ne peut traverser l'eau qu'en passant par les ponts*. Euler fut le premier à apporter une solution formelle à ce problème, et posa pour y parvenir les fondations de la théorie des graphes.

En 1845, Gustav Kirchhoff publia la loi de Kirchhoff dans les circuits permettant de calculer la tension et le courant dans les circuits électriques. Toute une catégorie de problèmes sur les graphes (les problèmes de flots) se modélisent en respectant ces lois.

En 1852, Francis Guthrie posa le problème des quatre couleurs qui consiste à savoir s'il est possible de colorer une carte en utilisant uniquement quatre couleurs, de manière à ce que deux pays frontaliers soient toujours de couleurs différentes. L'histoire de ce théorème est détaillée section 4.4.3 p. 78. En essayant de résoudre ce problème les mathématiciens ont inventé de nombreux concepts fondamentaux de la théorie des graphes.

### 3.1.2 Définitions

Dans cette partie, nous établissons quelques définitions importantes relatives aux graphes. Pour obtenir plus de détails sur les graphes et leurs propriétés, consulter [Sakarovitch, 1984a; Fournier, 2006] ou éventuellement [Kaufman, 1964].

Un graphe peut être défini de manière informelle comme étant un ensemble de points reliés par des lignes dans le plan. Cette définition a l'avantage d'être simple à comprendre, mais elle est trop imprécise pour permettre de bâtir une théorie. Par exemple, elle ne permet pas de remarquer que les graphes des figures 3.2 et 3.3 représentent le même graphe. On formalise donc cette notion avec la définition 21.

**Définition 21** (Graphe). Un graphe  $G$  est un couple  $(S, A)$  où  $S$  est un ensemble fini et  $A \subseteq S \times S$  une relation binaire sur  $S$ .  $S$  est appelé ensemble des sommets, et  $A$  ensemble des arcs. On note  $|S|$  (resp.  $|A|$ ) le cardinal de  $S$  (resp.  $A$ ), c'est-à-dire le nombre de sommets (resp. d'arcs). Un

graphe est dit pondéré si un poids est associé à chacun de ses arcs. Le poids d'un arc  $u$  est noté  $\omega(u)$ . Si aucun qualificatif n'est utilisé, le graphe est non pondéré.

**Définition 22** (Graphe non orienté). Les graphes non orientés sont des graphes pour lesquels la relation binaire entre sommets est symétrique. Les éléments de  $A$  sont des paires (sans ordre) de sommets et se nomment arêtes.

*Exemple 21* (Graphe de Petersen). Un exemple de graphe non orienté est représenté figure 3.2. Il s'agit du graphe de Petersen qui possède beaucoup de propriétés intéressantes. Par exemple, ce graphe est non planaire. Il est également unitaire, c'est-à-dire qu'il peut être représenté dans le plan avec des arcs de même taille. Il est souvent utilisé lors de la recherche de contre-exemples.

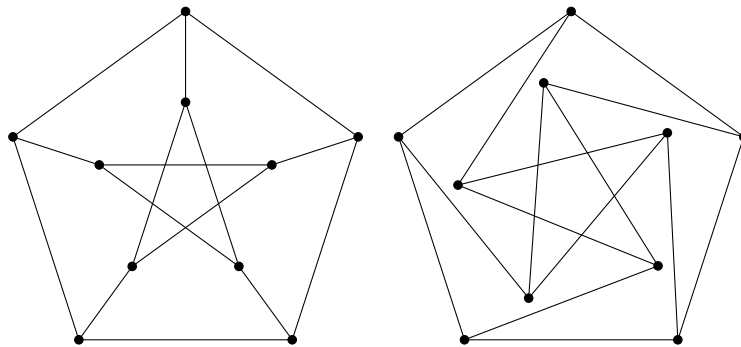


FIGURE 3.2 – Deux représentations du graphe de Petersen

**Remarque :**

la manière de représenter un graphe n'est pas unique. Par exemple, la figure 3.2 contient deux représentations du graphe de Petersen. La figure 3.3 contient deux autres représentations du même graphe.

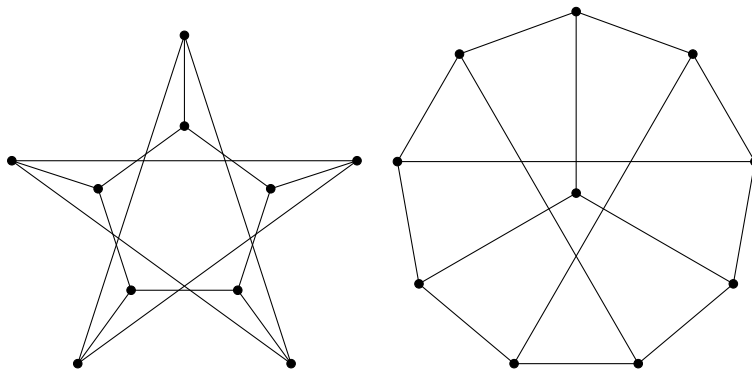


FIGURE 3.3 – Deux représentations supplémentaires du graphe de Petersen

**Définition 23** (Graphe orienté). Les graphes orientés sont des graphes pour lesquels la relation binaire n'est pas symétrique. Les éléments de  $A$  sont des couples (ordonnés) de sommets et se nomment arcs (là où une relation est symétrique, on la matérialise alors par deux arcs de sens opposé).

**Définition 24** (Extrémités d'un arc). Soit  $G = (S, A)$  un graphe orienté. Soit un arc  $u \in A$ . On note  $I$  et  $T$  les deux application qui associent à  $u$  son extrémité initiale et son extrémité terminale.

$$\begin{aligned} I : A &\rightarrow S \\ u = (a, b) &\mapsto a \end{aligned}$$

$$\begin{aligned} T : A &\rightarrow S \\ u = (a, b) &\mapsto b \end{aligned}$$

Soit  $G = (S, A)$  un graphe orienté, et  $X \subset S$ . On note :

$$\begin{aligned} \Omega^+(X) &= \{u \in A / I(u) \in X, T(u) \notin X\} \\ \Omega^-(X) &= \{u \in A / I(u) \notin X, T(u) \in X\} \end{aligned}$$

**Définition 25** (Isomorphisme de graphe). On définit un isomorphisme d'un graphe  $G_1 = (S_1, A_1)$  sur un graphe  $G_2 = (S_2, A_2)$  par deux bijections  $\Phi : S_1 \rightarrow S_2$  et  $\Psi : A_1 \rightarrow A_2$  telles que  $\forall u \in A_1, \Psi(u) = (\Phi(I(u)), \Phi(T(u)))$ .

De manière informelle, deux graphes sont isomorphes s'ils ont la même structure et qu'ils ne diffèrent que par leurs étiquettes (c'est-à-dire les noms des sommets).

**Définition 26** (Chemin, chaîne). Dans un graphe orienté, un chemin entre deux sommets  $a$  et  $b$  est une suite finie de  $n$  sommets  $(s_i)$  tels que  $s_1 = a$ ,  $s_n = b$  et  $\forall i \in [1, n-1], (s_i, s_{i+1}) \in A$ . Un chemin est dit élémentaire si chaque sommet  $y$  est présent au plus une fois. Dans le cas d'un graphe non orienté, on parle de chaîne. S'il existe un chemin entre  $a$  et  $b$ , on notera  $a \rightsquigarrow b$ .

**Définition 27** (Circuit, cycle, boucle). Un circuit est un chemin dont le sommet initial coïncide avec le sommet final. Une boucle est un circuit de longueur 1. Dans le cas d'un graphe non orienté, on parle de cycle.

**Définition 28** (Degré d'un sommet). Dans un graphe non orienté, le degré d'un sommet est le nombre d'arêtes issues de ce sommet. La somme des degrés de chaque sommet est égale au double du nombre total d'arêtes.

Dans un graphe orienté, on distingue pour un sommet  $a$  le degré entrant et le degré sortant. Le premier correspond au nombre d'arcs dont l'extrémité finale est  $a$ . Le second est le nombre d'arcs dont l'extrémité initiale est  $a$ . Le degré d'un sommet  $a$  dans un graphe orienté est la somme du degré entrant et sortant de  $a$ . Un sommet dont le degré entrant est 0 sera appelé une *source* et un sommet dont le degré sortant est 0 sera appelé un *puits*.

**Définition 29** (Graphe connexe). Un graphe non orienté est dit connexe si pour tout couple de sommets  $a$  et  $b$  du graphe, il existe une chaîne entre  $a$  et  $b$ .

**Définition 30** (Graphe fortement connexe). Un graphe orienté est dit fortement connexe si pour tout couple de sommets  $a$  et  $b$  du graphe,  $a \rightsquigarrow b$  et  $b \rightsquigarrow a$ .

**Définition 31** (Graphe complet). Un graphe complet est un graphe pour lequel tous les sommets sont reliés entre eux deux à deux. Le graphe complet à  $n$  sommets est noté  $K_n$ .



### 3.1.3 Cycles eulériens et hamiltoniens

**Définition 32** (Cycle eulérien, circuit eulérien). Un circuit d'un graphe (resp. cycle d'un graphe non orienté)  $G$  est dit eulérien s'il inclut tous les arcs (resp. arêtes) de  $G$ .

**Définition 33** (Graphe eulérien). Un graphe (non orienté) est eulérien s'il admet un cycle eulérien.

**Définition 34** (Sous graphe, graphe partiel, sous graphe partiel). Soit  $G = (S, A)$  un graphe. Soient  $S' \subset S$  et  $A' \subset A$ . Posons  $A_{S'} = \{u \in A / I(u) \in S', T(u) \in S'\}$ . Alors :

- $G_{S'} = (S', A_{S'})$  est appelé sous graphe de  $G$  induit par  $S'$
- $(S, A')$  est appelé graphe partiel de  $G$  construit sur  $A'$
- $(S', A' \cap A_{S'})$  est appelé sous graphe partiel de  $G$

**Lemme 1.** Soit  $A'$  l'ensemble des arcs inclus dans un cycle de  $G = (S, A)$ . Tous les sommets du graphe partiel  $G' = (S, A')$  sont de degré pair.

PREUVE :

Si un sommet est de degré impair, il ne peut pas faire partie d'un cycle. □

**Théorème 10** (CNS de graphe (non orienté) eulérien). Une condition nécessaire et suffisante pour qu'un graphe soit eulérien est qu'il soit connexe et que tous ses sommets soient de degré pair.

PREUVE :

Cette preuve utilise des résultats relatifs aux arbres énoncés dans le chapitre 4.2.

Selon le lemme 1, la condition est nécessaire. Montrons maintenant qu'elle est suffisante :

Supposons qu'il existe des graphes connexes dont tous les sommets sont de degré pair et qui ne soient pas eulériens. Soit  $G$  un tel graphe minimum (i.e. tout graphe connexe dont les sommets sont de degré pair et ayant moins d'arcs que  $G$  est eulérien).

$G$  possède au moins un cycle, sinon  $G$  serait un arbre (selon le théorème 15 p. 68) et posséderait donc des sommets de degré 1.

Soit  $\Gamma$  un cycle de  $G$  ayant un nombre maximum d'arcs. Soit  $G'$  une composante connexe du graphe partiel  $(S, A - \Gamma)$ .

Par construction,  $G'$  est connexe, tous ses sommets de degré pair, et moins d'arcs que  $G$ . Par hypothèse, il admet un cycle eulérien  $\Gamma'$ .

$\Gamma$  et  $\Gamma'$  sont deux cycles de  $G$  qui ont au moins un sommet en commun. Ils peuvent donc être fusionnés en un seul cycle  $\Gamma''$ .  $\Gamma''$  a plus d'arcs que  $\Gamma$  et est eulérien, ce qui est en contradiction avec les hypothèses. □

**Corollaire 2.** Un graphe connexe possède une chaîne eulérienne si et seulement si le nombre de sommets de degrés impairs est inférieur ou égal à 2.

PREUVE :

On rappelle qu'un graphe possède obligatoirement un nombre pair de sommets de degrés impairs. Soit  $G = (S, A)$  un graphe.

- Si le graphe ne possède aucun sommet de degré impair, alors selon le théorème 10, il possède un cycle eulérien qui est également une chaîne eulérienne.
- Si le graphe possède deux sommets  $x$  et  $y$  de degrés impairs, alors on considère  $G' = (S, A \cup (x, y))$ .  $G'$  ne possède aucun sommet de degré impair, et il possède donc un cycle eulérien  $\Gamma$ . De plus,  $\Gamma - (x, y)$  est une chaîne eulérienne de  $G$ .
- Si le graphe possède plus de deux sommets de degrés impairs, alors il ne peut pas avoir de cycle eulérien ni de chaîne eulérienne non fermée : une chaîne eulérienne non fermée a nécessairement pour extrémités des sommets de degrés impairs. Il ne peut donc y en avoir plus de deux dans le graphe. □

**Définition 35** (Circuit élémentaire, cycle élémentaire). Un cycle ou un circuit est dit élémentaire si tout sommet est adjacent à deux arcs de la séquence au maximum.

**Définition 36** (Circuit hamiltonien, cycle hamiltonien). Un circuit d'un graphe (resp. cycle d'un graphe non orienté)  $G$  est dit hamiltonien (d'après **Hamilton**) s'il est élémentaire et comporte  $|S|$  arcs (resp. arêtes), c'est-à-dire s'il passe par tous les sommets du graphe.

**Définition 37** (Graphe hamiltonien). Un graphe orienté (resp. non orienté) est hamiltonien s'il possède un circuit (resp. cycle) hamiltonien.

**Remarque :**

bien que les définitions de cycle eulérien et de cycle hamiltonien soient très proches, les problèmes de leur existence sont très différents quand à leur complexité. En effet, il existe une condition nécessaire et suffisante simple permettant de déterminer s'il existe un cycle eulérien dans un graphe alors qu'il n'existe rien de tel dans le cas des cycles hamiltoniens. De plus, la recherche de cycles hamiltoniens dans un graphe est un problème NP-complet.

### 3.1.4 Représentation des graphes

Il existe deux méthodes principales pour représenter numériquement un graphe : les listes d'adjacences et les matrices d'adjacences. Ces méthodes sont décrites par exemple dans [Cormen *et al.*, 1994; De Werra *et al.*, 2003].

Dans le cas d'un graphe peu dense ( $|A| \ll |S|^2$ ), la méthode des listes d'adjacences sera préférée. Par contre, dans le cas d'un graphe très dense, voire d'un graphe complet ( $|A| \simeq |S|^2$ ), la méthode des matrices d'adjacences sera favorisée.

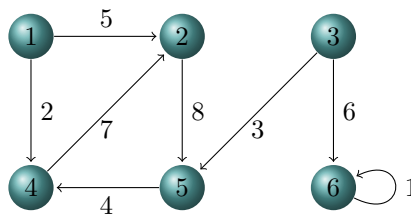


FIGURE 3.4 – Graphe orienté pondéré

La figure 3.4 représente un graphe. Ce graphe peut être représenté par la liste d'adjacences figure 3.5 ou par la matrice d'adjacences figure 3.6.

#### Listes d'adjacences

La représentation par listes d'adjacences d'un graphe  $G = (S, A)$  consiste en un tableau  $Adj$  de  $|S|$  listes, une par sommet de  $G$ . Pour chaque sommet  $x \in S$ , la liste d'adjacences  $Adj[x]$  est la liste des sommets  $y$  pour lesquels il existe un arc  $(x, y) \in A$ .

Dans le cas d'un graphe pondéré, un poids doit être associé à chaque arc. Ce poids est alors simplement stocké avec le sommet, dans la liste d'adjacences.

L'inconvénient principal de la représentation par listes d'adjacences est le temps nécessaire pour tester si un arc  $(x, y)$  existe dans le graphe : il faut parcourir la liste d'adjacences de  $x$ , ce qui a comme coût le degré de  $x$ . C'est pourquoi la représentation par matrice d'adjacences sera préférée dans le cas de graphes denses.

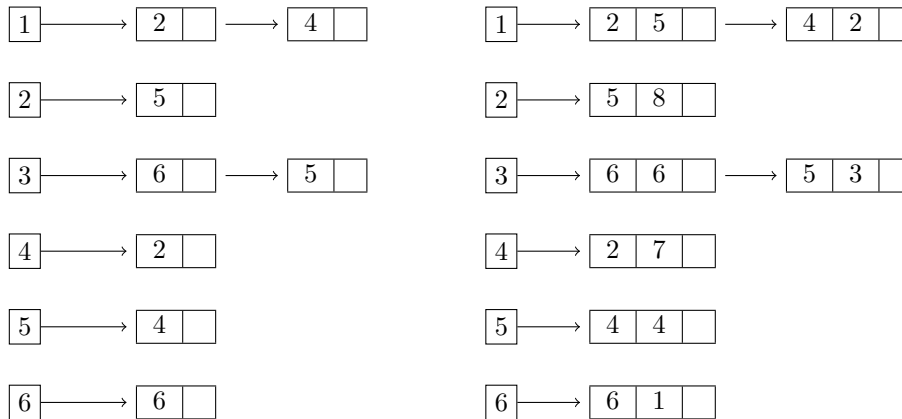


FIGURE 3.5 – Liste d'adjacences, graphe pondéré et non pondéré

**Matrice d'adjacences**

Pour la représentation par matrices d'adjacences d'un graphe  $G = (S, A)$ , il faut numéroté les sommets de 1 à  $|S|$ . Le graphe est alors représenté par une matrice  $|S| \times |S|$ ,  $M = (a_{ij})$  définie par :

$$a_{ij} = \begin{cases} 1 & \text{si } (i, j) \in A \\ 0 & \text{sinon} \end{cases}$$

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

	1	2	3	4	5	6
1	0	5	0	2	0	0
2	0	0	0	0	8	0
3	0	0	0	0	3	6
4	0	7	0	0	0	0
5	0	0	0	4	0	0
6	0	0	0	0	0	1

FIGURE 3.6 – Matrice d'adjacences, graphe pondéré et non pondéré

**Remarque :**

dans le cas d'un graphe non orienté, la matrice d'adjacences est symétrique. Il est alors possible de représenter le graphe par la partie de la matrice située au dessus de la diagonale.

La matrice d'adjacences peut également servir à représenter un graphe pondéré. Dans ce cas, les coefficients  $a_{ij}$  de la matrice contiennent le poids de l'arc reliant  $i$  à  $j$ . Il faut alors utiliser une valeur particulière pour signifier qu'il n'existe pas d'arc entre  $i$  et  $j$ , comme par exemple 0 ou  $\infty$ .

**Remarque :**

il est souvent intéressant de limiter au maximum la taille des données de la matrice d'adjacences. En particulier, dans le cas de graphes non pondérés, une matrice de booléens suffit.

## 3.2 Parcours d'un graphe

Le premier traitement à appliquer à un graphe est le parcours ou exploration : partant d'un sommet  $s$ , parcourir un graphe consiste à déterminer l'ensemble des descendants de  $s$ , c'est-à-dire l'ensemble des sommets atteignables partant de  $s$  : l'ensemble  $\{x \in S/s \rightsquigarrow x\}$ .

Il existe différents parcours possibles dont les plus classiques sont les parcours en largeur et en profondeur. À la différence des arbres, les graphes peuvent contenir des cycles. Les parcours demandent donc de prendre quelques précautions : il faut marquer les sommets visités afin d'éviter de les traiter plusieurs fois.

### 3.2.1 Parcours en largeur

Le parcours en largeur est l'un des parcours les plus simples, et il est à la base de nombreux algorithmes sur les graphes, comme par exemple l'algorithme du plus court chemin de [Dijkstra](#). Il est également appelé BFS (*Breath First Search*).

Le principe de l'algorithme de parcours en largeur partant d'un sommet  $s$  de  $G$  est de découvrir tous les sommets voisins de  $s$ , puis les voisins des voisins, et de manière plus générale de découvrir les sommets situés à une distance  $k$  de  $s$  avant d'explorer les sommets situés à une distance  $k + 1$  de  $s$ . L'algorithme de parcours est décrit figure 3.

Pour garder une trace de la progression, le parcours en largeur colorie les sommets. Initialement, tous les sommets sont coloriés en blanc, puis ils sont coloriés en noir lorsqu'ils sont visités.

---

**Algorithme 3 :** Parcours en largeur: bfs(graphe  $G$ , nœud  $depart$ )

---

```

Données :  $f$  file de nœuds
Données :  $x, z$  nœuds
enfiler  $depart$  dans  $f$  ;
marquer  $depart$  ;
tant que  $f \neq \emptyset$  faire
    défiler  $x$  de  $f$  ;
    pour tous  $z \in successeur(x)$  faire
        si  $z$  non marqué alors
            marquer  $z$  ;
            enfiler  $z$  dans  $f$  ;
        fin
    fin
fin

```

---

#### Analyse de l'algorithme

Nous allons analyser le temps d'exécution de l'algorithme de parcours en largeur pour un graphe  $G = (S, A)$  dans le cas d'une représentation par listes d'adjacences. Tout d'abord, remarquons que chaque sommet est marqué avant d'être entré dans la file, et qu'il ne sera donc enfilé qu'une fois. Le coût nécessaire pour enfiler ou défiler un élément étant de  $\Theta(1)$ , le temps total nécessaire pour les opérations sur la file sont donc de  $\Theta(|S|)$ . Remarquons ensuite que la liste d'adjacences de chaque sommet n'est visitée qu'une fois. En effet, elle n'est balayée que lorsque le sommet est défilé. La somme des longueurs des listes d'adjacences étant de  $|A|$ , le temps total pour balayer les listes d'adjacences est de  $\Theta(|A|)$ . La complexité du parcours en profondeur du graphe est donc de  $\Theta(|A| + |S|)$ .

#### Propriétés de l'algorithme

L'algorithme de parcours en largeur vérifie les propriétés suivantes :

- BFS est complet : il parcourt tous les nœuds du graphe ;

- BFS a une complexité en espace de  $\Theta(|A| + |S|)$  ;
- BFS a une complexité en temps de  $\Theta(|A| + |S|)$ .

### 3.2.2 Parcours en profondeur

Le principe du parcours en profondeur est de descendre plus profondément dans le graphe chaque fois que possible. Un parcours de graphe sera dit en profondeur si, à chaque étape, l'arc  $(x, y)$  est choisi tel que  $x$  soit le dernier sommet visité non marqué. Il est également appelé DFS (*Depth First Search*).

Pour implanter un parcours en profondeur, il est nécessaire d'utiliser une pile de sommets visités. Il est cependant possible d'utiliser la pile d'appel pour stocker les sommets (c'est-à-dire d'utiliser des appels récursifs). L'algorithme récursif de parcours en profondeur d'un graphe est décrit figure 4.

Comme dans le cas du parcours en largeur, il est nécessaire de marquer les sommets visités.

---

**Algorithme 4** : Parcours en profondeur: dfs(graphe  $G$ , nœud  $n$ )

---

```

Données :  $z$  nœud
marquer  $n$  ;
pour tous  $z \in \text{successeur}(n)$  faire
    si  $z$  non marqué alors
        marquer  $z$  ;
        dfs( $G, z$ ) ;
    fin
fin

```

---

#### Analyse de l'algorithme

Nous allons analyser le temps d'exécution de l'algorithme de parcours en profondeur pour un graphe  $G = (S, A)$  dans le cas d'une représentation par listes d'adjacences. Tout d'abord, remarquons que chaque sommet est marqué avant d'être visité, et qu'il ne sera donc visité qu'une fois. Le traitement d'un sommet se faisant en temps constant, le temps total nécessaire pour visiter les sommets est donc de  $O(|S|)$ .

Remarquons ensuite que la liste d'adjacences de chaque sommet n'est visitée qu'une fois. En effet, elle n'est balayée que lorsque le sommet est traité. La somme des longueurs des listes d'adjacences étant de  $|A|$ , le temps total pour balayer les listes d'adjacences est de  $O(|A|)$ . La complexité du parcours en profondeur du graphe est donc de  $O(|A| + |S|)$ .

#### Propriétés de l'algorithme

L'algorithme de parcours en largeur vérifie les propriétés suivantes :

- DFS est complet : il parcourt tous les nœuds du graphe ;
- DFS a une complexité en espace linéaire par rapport à la profondeur de la recherche ;
- DFS a une complexité en temps de  $\Theta(|A| + |S|)$ .

### 3.2.3 Composantes fortement connexes

Une des applications du parcours en profondeur d'un graphe  $G$  est la décomposition en composantes fortement connexes de  $G$ .

De nombreux algorithmes travaillant sur les graphes orientés commencent par une décomposition en composantes fortement connexes. Ceci permet de diviser le problème initial en sous-problèmes, un par composante fortement connexe.

**Définition 38** (Composante fortement connexe). On considère la relation d'équivalence : « il existe au moins un chemin de  $x$  et  $y$  et un chemin de  $y$  à  $x$  ». On appelle alors composante

fortement connexe d'un graphe les classes d'équivalence de cette relation. Un graphe fortement connexe peut alors être défini comme un graphe possédant une seule composante fortement connexe.

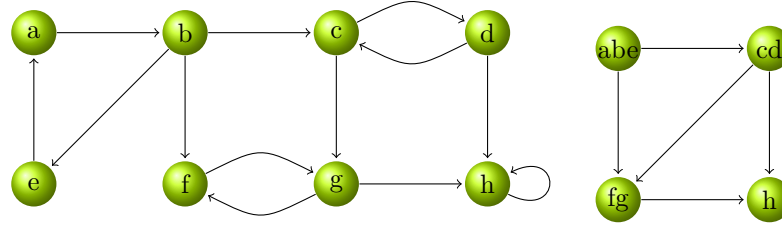


FIGURE 3.7 – Graphe et décomposition en composantes fortement connexes

Par exemple, la figure 3.7 représente un graphe et sa décomposition en composantes fortement connexes.

**Définition 39** (Transposée d'un graphe). On appelle transposée d'un graphe  $G = (S, A)$ , et on note  $G^T = (S, A^T)$  le graphe défini par  $A^T = \{(x, y) / (y, x) \in A\}$ .

Transposer un graphe revient donc à inverser l'orientation de tous ses arcs.

**Remarque :**

dans le cas d'une représentation par listes d'adjacences, la création de  $G^T$  est une opération de complexité  $\Theta(|S| + |A|)$ .

---

**Algorithme 5 :** Composantes fortement connexes:  $\text{cfc}(\text{Graphe } G)$

---

**Données :**  $ncfc$  entier : nombre de composantes fortement connexes

**Données :**  $nc$  tableau d'entiers : numéro de CFC de chaque sommet

$ncfc \leftarrow 0$  ;

**pour tous**  $i \in [1, |S|]$  **faire**

$nc[i] \leftarrow 0$  ;

**fin**

**pour tous**  $i \in [1, |S|]$  **faire**

**si**  $nc[i] = 0$  **alors**

$\text{dfs}(G, s_i)$  ;

$\text{dfs}(G^T, s_i)$  ;

$ncfc \leftarrow ncfc + 1$  ;

**pour tous**  $i/s_i$  *marqué pour les deux explorations* **faire**

$nc[i] \leftarrow ncfc$  ;

**fin**

**fin**

**fin**

---

**Remarque :**

$G$  et  $G^T$  ont les mêmes composantes fortement connexes.

Un algorithme simple de recherche de composantes fortement connexes est défini en 5. Le principe de l'algorithme est de partir d'un sommet  $s$ , de marquer tous les sommets de  $G$ , puis tous les sommets de  $G^T$  atteignables depuis  $s$ . Les sommets marqués par les deux parcours font partie de la même composante fortement connexe. On efface alors le marquage des sommets marqués une seule fois, et on recommence le traitement en partant d'un sommet non marqué.

Cet algorithme est peu efficace et sa complexité est de  $\Theta(|S| \times |A|)$  dans le pire des cas.

Il existe des algorithmes plus efficaces, comme par exemple l'algorithme de [Tarjan \(6\)](#) qui possède une complexité de  $\Theta(\max(|S|, |A|))$ . Pour la preuve de cette proposition, consultez [[Tarjan, 1972](#)].

---

**Algorithme 6** : Algorithme de Tarjan: `tarjan(Graphe G)`


---

**Données** :  $V \leftarrow \emptyset$  : ensemble des sommets visités  
**Données** :  $r \leftarrow 0$  : compteur pour les rangs  
**Données** :  $k \leftarrow 0$  : numéro de composante fortement connexe  
**Données** :  $\Pi \leftarrow \text{pileVide}$   
**tant que**  $V \neq S$  **faire**  
    choisir  $s$  dans  $S - V$  ;  
    desc( $s$ ) ;  
**fin**

---



---

**Procédure** desc(sommet  $s$ )

---

empiler  $s$  dans  $\Pi$  ;  
 $V \leftarrow V \cup \{s\}$  ;  
 $r \leftarrow r + 1$  ;  
 $r(s) \leftarrow r$  ;  
 $\theta(s) \leftarrow r$  ;  
**pour tous**  $x \in \text{succ}(s)$  **faire**  
    **si**  $x \notin V$  **alors**  
        desc( $x$ ) ;  
         $\theta(s) \leftarrow \min(\theta(s), \theta(x))$  ;  
    **sinon si**  $x \in \Pi$  **alors**  
         $\theta(s) \leftarrow \min(\theta(s), r(x))$  ;  
    **fin**  
**fin**  
**si**  $\theta(s) = r(s)$  **alors**  
     $k \leftarrow k + 1$  ;  
    **répéter**  
         $z \leftarrow \text{depiler}(\Pi)$  ;  
         $c(z) \leftarrow k$  ;  
    **jusqu'à**  $z = s$  ;  
**fin**

---

### 3.2.4 Sommets d'articulation, ponts, et k-connexité

**Définition 40** (Sommets d'articulation). Soit  $G = (S, A)$  un graphe connexe. On appelle sommet d'articulation de  $G$  tout sommet dont la suppression rend  $G$  non connexe.

**Définition 41** (Pont, isthme). Soit  $G = (S, A)$  un graphe connexe. On appelle pont ou isthme de  $G$  tout arc dont la suppression rend  $G$  non connexe.

Les notions de points d'articulation et d'isthmes servent à mesurer la vulnérabilité d'un réseau. En effet, en cas d'incident sur un point d'articulation ou sur un isthme, le réseau n'est plus connexe et ne peut plus donc remplir son office.

**Définition 42** (k-connexité). Un graphe non orienté est k-connexé s'il reste connexe après suppression d'un ensemble quelconque de k-1 arêtes et s'il existe un ensemble de k arêtes dont la suppression le déconnecte. Autrement dit, un graphe est k-connexé si et seulement s'il existe au moins k chaînes indépendantes (arcs-disjointes) entre chaque couple de sommets. Cette notion est utilisée en électronique, en calcul de la fiabilité, et dans l'étude de jeux de stratégie comme le *cut and connect*.

La notion de k-connexité permet de mesurer la robustesse d'un réseau à un certain nombre de pannes : un réseau k-connexé sera robuste à au moins  $k - 1$  pannes.

### 3.3 Plus court chemin dans un graphe

Dans cette partie, nous allons nous intéresser aux différents problèmes de chemins optimaux dans un graphe. Ceci représente une part importante des problèmes de recherche opérationnelle.

Les différents algorithmes de plus court chemin ainsi que de nombreux algorithmes de recherche opérationnelle sur les graphes sont décrits dans [Sakarovitch, 1984b; Fournier, 2007a].

**Définition 43** (Circuit absorbant). Soit  $G = (S, A)$  un graphe orienté pondéré. On appelle circuit absorbant un circuit le long duquel la somme des poids des arcs est négative, c'est-à-dire un circuit  $\Gamma$  tel que :

$$\sum_{u \in \Gamma} \omega(u) < 0$$

**Définition 44** (Coût d'un chemin). Soient  $G = (S, A)$  un graphe pondéré avec une fonction de pondération  $\omega : A \rightarrow \mathbb{R}$ , et  $(a, b) \in S \times S$ . Soit  $p = (a_0, \dots, a_n)$  un chemin de  $a$  à  $b$ . On appelle coût du chemin  $p$  la somme des coûts des arcs le composant :

$$\omega(p) = \sum_{i=1}^n \omega(a_{i-1}, a_i)$$

Dans un graphe orienté sans circuit absorbant, rechercher le plus court chemin allant d'un sommet  $a$  à un sommet  $b$  consiste à chercher un chemin de coût minimal entre  $a$  et  $b$ . Dans le cas d'un graphe contenant au moins un cycle absorbant, le problème du plus court chemin n'a pas de sens : en effet, il existe alors des chemins de coût  $-\infty$ .

**Définition 45** (Plus court chemin). On définit la longueur du plus court chemin entre  $a$  et  $b$  par :

$$\delta(a, b) = \begin{cases} \min\{\omega(p) : a \xrightarrow{p} b\} & \text{si } a \rightsquigarrow b \\ +\infty & \text{sinon} \end{cases}$$

Un plus court chemin de  $a$  à  $b$  est alors défini comme un chemin  $p$  de longueur  $\omega(p) = \delta(a, b)$ .

Les algorithmes de plus court chemin se divisent en deux catégories : les algorithmes à origine unique, et les algorithmes pour tout couple de sommets.

La première catégorie est la plus utilisée : partant d'un sommet  $s$  du graphe, quel est la distance minimale à tous les autres sommets, et quel est le chemin qui permet d'obtenir la distance minimale ?

La seconde catégorie consiste à calculer un *distancier*, c'est-à-dire une matrice  $|S| \times |S|$  contenant les distances entre tous les couples de sommets.

#### 3.3.1 Plus court chemin à origine unique

Dans cette partie, nous nous concentrons sur le problème du plus court chemin à origine unique : étant donné un graphe orienté pondéré  $G = (S, A)$ , on souhaite déterminer le plus court chemin depuis un sommet d'origine  $s \in S$  vers n'importe quel sommet  $a \in S$ . Beaucoup de problèmes sont équivalents à celui-ci, et en particulier :



- plus court chemin à destination unique. Pour se ramener au problème du plus court chemin à origine unique, il suffit de chercher le chemin dans  $G^T$  ;
- plus court chemin pour un couple de sommets donné. Ce problème est contenu dans le problème de plus court chemin à origine unique ;
- plus court chemin pour tout couple de sommet. Ce problème peut être résolu en considérant  $n$  problèmes de plus court chemin à origine unique, avec  $n = |S|$ . Il existe cependant des algorithmes plus efficaces.

#### Représentation des plus courts chemins

Dans le cas de la résolution d'un problème de plus court chemin entre deux sommets  $a$  et  $b$ , on souhaite connaître la distance minimale entre  $a$  et  $b$ , mais également le chemin permettant d'aboutir à cette distance. La représentation utilisée pour stocker les chemins est la suivante : étant donné un graphe  $G = (S, A)$ , on gère pour chaque sommet  $a \in S$  un prédécesseur  $\pi[a]$  qui est soit un autre sommet, soit *null*. Ainsi, partant de l'arrivée, il est possible de remonter à l'origine par applications successives de  $\pi$ .

L'initialisation de  $\pi$  et de  $d$  se fait par l'algorithme 7.

---

**Algorithme 7** : Initialisation: init(graphe G, sommet s)

---

```

Données : a sommet
pour tous  $a \in S$  faire
     $d[a] \leftarrow \infty$  ;
     $\pi[a] \leftarrow \text{null}$  ;
fin
 $d[s] \leftarrow 0$  ;

```

---

#### Relâchement

La plupart des algorithmes de plus court chemin à origine unique utilisent la technique du *relâchement*. Pour chaque sommet  $a \in S$ , on gère un attribut  $d[a]$  qui est un majorant de la longueur du plus court chemin entre l'origine  $s$  et  $a$ , et est mis à jour par la procédure de relâchement.

---

**Algorithme 8** : Algorithme de relâchement: relacher(sommet a, sommet b)

---

```

si  $d[b] > d[a] + \omega(a, b)$  alors
     $d[b] \leftarrow d[a] + \omega(a, b)$  ;
     $\pi[b] \leftarrow a$  ;
fin

```

---

Le relâchement d'un arc  $(a, b)$  consiste à tester s'il est possible d'améliorer le plus court chemin trouvé jusqu'à présent en passant par  $a$ , et si tel est le cas de mettre à jour  $d[b]$  et  $\pi[b]$ . Il est décrit par l'algorithme 8.

*Propriétés 1* (Propriétés du relâchement). Soit un graphe  $G = (S, A)$ . Les propriétés ci-dessous se rapportent aux plus courts chemins issus de  $s \in S$ .

**Inégalité triangulaire** :  $\forall (a, b) \in A$ , on a  $\delta(s, b) \leq \delta(s, a) + \omega(a, b)$ .

**Propriété du majorant** :  $\forall a \in S, d[a] \geq \delta(s, a)$ . De plus, lorsque  $d[a] = \delta(s, a)$ , sa valeur ne change plus.

**Propriété aucun-chemin** : Si  $s \not\rightsquigarrow a$ , alors  $d[a] = \delta(s, a) = +\infty$

**Propriété de convergence :** Soit  $(a, b) \in A$ , et  $\Upsilon = (s, \dots, a)$  un plus court chemin de  $G$ . Si  $d[a] = \delta(s, a)$  à un certain instant antérieur au relâchement de  $(a, b)$ , alors  $d[b] = \delta(s, b)$  en permanence après le relâchement.

**Propriété de relâchement de chemin :** Soit  $\Upsilon = (s_0, \dots, s_n)$  un plus court chemin de  $a = s_0$  à  $b = s_n$ . Si les arcs sont relâchés dans l'ordre  $(s_0, s_1), (s_1, s_2), \dots, (s_{n-1}, s_n)$ , alors  $d[b] = \delta(a, b)$ . Cette propriété reste vraie même si d'autres relâchements s'entrelacent avec les relâchements des arcs de  $\Upsilon$ .

**Propriété de sous graphe prédécesseur :** Soit  $a \in S$ . Une fois que  $d[a] = \delta(s, a)$ , le sous graphe prédécesseur  $G_\pi$  est une arborescence de plus courts chemins de racine  $s$ .

#### Algorithme de Bellman-Ford

L'algorithme de **Bellman-Ford** permet de résoudre le problème du plus court chemin à origine unique dans le cas général où les poids peuvent avoir des valeurs négatives (à condition bien entendu qu'il n'y ait pas de cycle absorbant). Le principe de l'algorithme est très simple : il s'agit d'appliquer la procédure de relâchement à tous les arcs du graphe autant de fois qu'il y a de sommets.

Une extension de l'algorithme permet de déterminer la présence de cycle absorbant. L'algorithme complet est présenté en 9.

---

#### Algorithme 9 : Algorithme de Bellman-Ford: bellmanFord(graphe G, sommet s)

---

```

init(G,s) ;
pour i ∈ [1, |S| - 1] faire
    pour tous (a, b) ∈ A faire
        relacher(a, b) ;
    fin
fin
pour tous (a, b) ∈ A faire
    si d[b] > d[a] + ω(a, b) alors
        retourner FAUX ;
    fin
fin
retourner VRAI ;

```

---

L'algorithme de Bellman-Ford a une complexité de  $\Theta(|S| \cdot |A|)$  : l'initialisation est en  $\Theta(|S|)$ , et chacun des  $|S| - 1$  passage de la boucle *pour* s'exécute en  $\Theta(|A|)$ .

**Lemme 2.** Soit  $G = (S, A)$  un graphe orienté pondéré (de fonction de pondération  $\omega$ ), sans cycle absorbant. Alors après  $|S| - 1$  itérations de la boucle de l'algorithme 9,  $\forall a \in S/s \rightsquigarrow a, d[a] = \delta(s, a)$

PREUVE :

Soient deux sommets  $a$  et  $b$  tels que  $a \rightsquigarrow b$ . Soit un plus court chemin élémentaire de  $a$  à  $b$  :  $\Upsilon = (s_0, \dots, s_n)$  avec  $s_0 = a$  et  $s_n = b$ . Le chemin  $\Upsilon$  possède au plus  $|S| - 1$  arcs, donc  $n \leq |S| - 1$ . Lors de la  $i^{\text{e}}$  itération de la boucle principale de l'algorithme 9, on relâche tous les arcs, et en particulier  $(s_{i-1}, s_i)$ . Selon la propriété de relâchement des propriétés 1, après cette itération,  $\Upsilon$  étant un plus court chemin,  $d[s_i] = \delta(a, s_i)$ .  
Donc, après  $|S| - 1$  itérations,  $d[s_n] = d[b] = \delta(a, b)$ . □

**Théorème 11** (Validité de l'algorithme de Bellman-Ford). Soit  $G = (S, A)$  un graphe orienté pondéré, de fonction de pondération  $\omega$ . Si  $G$  ne contient aucun circuit absorbant, alors l'algorithme 9 retourne VRAI,  $\forall a \in S, d[a] = \delta(s, a)$ , et le sous graphe prédécesseur  $G_\pi$  est une arborescence de plus courts chemins de racine  $s$ .

Si  $G$  contient un circuit absorbant, alors l'algorithme retourne FAUX.

PREUVE :

- Supposons que le graphe ne possède aucun circuit absorbant.
  - Selon le lemme 2, si  $s \rightsquigarrow a$  alors à la fin de l'algorithme,  $d[a] = \delta(s, a)$ .
  - De plus, si  $s \not\rightsquigarrow a$ , la propriété aucun chemin des propriétés 1 implique  $d[a] = \delta(s, a)$ .
  - Selon la propriété de sous graphe prédécesseur,  $G_\pi$  est une arborescence de plus courts chemins.
  - Après l'exécution de l'algorithme,  $\forall (a, b) \in A$  :

$$\begin{aligned} d[b] &= \delta(s, b) \\ &\leq \delta(s, a) + \omega(a, b) \\ &\leq d[a] + \omega(a, b) \end{aligned}$$

Le test de l'algorithme de Bellman-Ford retournera donc VRAI.

- Supposons que le graphe ait un circuit absorbant  $\Gamma = (s_0, \dots, s_n)$  avec  $s_0 = s_n$  accessible depuis  $s$ . Alors

$$\sum_{i=1}^n \omega(s_{i-1}, s_i) < 0$$

Si l'algorithme retournerait VRAI, alors  $\forall i \in \{1, \dots, n\}, d[s_i] \leq d[s_{i-1}] + \omega(s_{i-1}, s_i)$ . La somme de ces inégalités sur le circuit  $\Gamma$  donne :

$$\begin{aligned} \sum_{i=1}^n d[s_i] &\leq \sum_{i=1}^n (d[s_{i-1}] + \omega(s_{i-1}, s_i)) \\ &\leq \sum_{i=1}^n d[s_{i-1}] + \sum_{i=1}^n \omega(s_{i-1}, s_i) \end{aligned}$$

Comme  $s_0 = s_n$

$$\sum_{i=1}^n d[s_i] = \sum_{i=1}^n d[s_{i-1}]$$

Donc,

$$0 \leq \sum_{i=1}^n \omega(s_{i-1}, s_i)$$

ce qui est en contradiction avec l'hypothèse. L'algorithme retourne donc bien faux si le graphe possède un cycle absorbant. □

Un exemple de plus court chemin est représenté sur la figure 3.8. Sur cet exemple, le chiffre à droite du nom de sommet représente la distance à l'origine, et les flèches rouges indiquent le plus court chemin.

#### Algorithme de Dijkstra

Edsger Dijkstra ['et,sxər 'deɪk,stra] (1930-2002), était un mathématicien et informaticien hollandais. Il est particulièrement connu pour ses travaux se rapportant à la théorie des graphes (et pour l'algorithme de plus court chemin qui porte son nom), ainsi que pour ses travaux sur la programmation concurrente (qui aboutit à la notion de sémaphore). Il est également connu pour sa mauvaise opinion de l'instruction GOTO qui aboutit à l'article « A Case against the GO TO Statement » disponible à l'adresse : <https://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF>. Cet article a fortement contribué au remplacement de l'instruction GOTO par des structures de contrôle, comme la boucle *while*.

Dijkstra reçut le *prix Turing* en 1972 pour ses contributions en informatique théorique concernant la théorie des langages. Une traduction du discours (*The humble programmer*) qu'il prononça

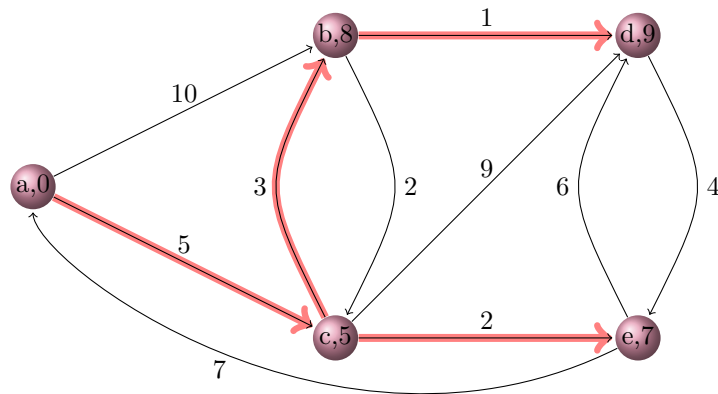


FIGURE 3.8 – Plus court chemin dans un graphe

à cette occasion est disponible à l'adresse suivante :

<https://artechnic.wordpress.com/2014/08/27/le-programmeur-modeste/>

L'algorithme de Dijkstra (publié dans [Dijkstra, 1959]) permet de résoudre le problème du plus court chemin à origine unique dans un graphe orienté pondéré  $G = (S, A)$  dans le cas où tous les arcs ont des poids positifs ou nuls. On suppose donc que  $\forall (a, b) \in A, \omega(a, b) \geq 0$ .

Le principe de l'algorithme de Dijkstra est semblable à celui de Bellman : il s'agit d'appliquer la procédure de relâchement aux arcs du graphe. Cependant, grâce à l'hypothèse supplémentaire de positivité des poids des arcs, il est possible de relâcher chaque arc une seule fois. Pour cela, il suffit de relâcher systématiquement les arcs partant du sommet non traité le plus proche de la source.

Pour repérer le sommet non traité le plus proche de la source, l'algorithme de Dijkstra peut par exemple s'appuyer sur une file de priorité  $F$  de sommets ayant pour clé la valeur de leur attribut  $d$ . Il utilise également un ensemble  $E$  de sommets dont les longueurs finales du plus court chemin à partir de l'origine ont déjà été calculées.

L'algorithme de Dijkstra est décrit dans 10.

---

**Algorithme 10 :** Algorithme de Dijkstra: `dijkstra(graphe G, sommet s)`

---

```

init(G,s) ;
E ← ∅ ;
F ← S ;
tant que F ≠ ∅ faire
    a ← extraireMin(F) ;
    E ← E ∪ {a} ;
    pour tous b ∈ succ(a) faire
        relacher(a, b) ;
    fin
fin

```

---

On remarque que  $F = S - E$ . Ceci est vrai initialement ( $E = \emptyset, F = S$ ) et à chaque itération, le sommet supprimé de  $F$  est ajouté dans  $E$ .

On remarque ensuite que l'algorithme de Dijkstra choisit à chaque itération le sommet de

$S - E$  qui a le coût le plus faible. On dit qu'il utilise une *stratégie gloutonne* : il choisit ce qui semble être la meilleure possibilité à chaque itération sans jamais remettre ce choix en cause. Il faut noter qu'une stratégie gloutonne n'aboutit pas toujours à un résultat optimal, mais que dans le cas de l'algorithme de Dijkstra on obtient effectivement le plus court chemin.

**Théorème 12** (Validité de l'algorithme de Dijkstra). *Si l'on exécute l'algorithme de Dijkstra sur un graphe orienté pondéré  $G = (S, A)$  avec une fonction de pondération positive  $\omega$  et une origine  $s \in S$ , après exécution on a :  $\forall a \in S, d[a] = \delta(s, a)$ .*

PREUVE :

Pour la preuve de ce théorème, consulter par exemple [Cormen et al., 1994]. L'idée de la preuve est de montrer qu'à chaque étape de l'algorithme,  $\forall a \in E, d[a] = \delta(s, a)$ .  $\square$

#### Complexité de l'algorithme :

la complexité de l'algorithme de Dijkstra dépend de la manière dont est implantée la file de priorité.

La solution la plus simple consiste à implanter la file de priorité par un tableau de  $|S|$  cases : les sommets sont numérotés de 1 à  $|S|$ , et la valeur de  $d[s_i]$  est stockée dans la case  $i$ . Le coût d'une insertion dans la file est alors de  $\Theta(1)$ . Par contre, le coût pour extraire l'élément de coût minimum de la file est de  $\Theta(|S|)$ . La complexité de l'algorithme est alors de  $\Theta(|S|^2 + |A|) = \Theta(|S|^2)$ . Remarquons que cette complexité est meilleure que celle de l'algorithme de Bellman-Ford : en général  $|S| < |A|$ .

Il existe de meilleures implantations de files de priorités, en utilisant par exemple des tas binomiaux ou des tas de Fibonacci. La complexité de l'algorithme devient alors  $\Theta((|S| + |A|) \log |S|)$  dans le premier cas, et  $\Theta(|S| \log |S| + |A|)$  dans le second.

#### Remarque :

la structure de tas est une structure algorithmique qui ne sera pas décrite dans ce cours (il s'agit d'un arbre binaire possédant certaines propriétés supplémentaires). Pour plus d'informations à ce sujet, il est possible de consulter tout livre d'algorithmique décrivant les structures non élémentaires, comme par exemple [Cormen et al., 1994].

### 3.3.2 Plus court chemin pour tout couple de sommet

Considérons un graphe orienté pondéré  $G$ . Nous cherchons maintenant à obtenir le plus court chemin pour tout couple de sommets du graphe. Ceci peut être utile par exemple lors de l'élaboration d'un outil permettant de choisir l'itinéraire optimal dans le métro, ou encore pour l'élaboration d'une table des distances entre toutes les villes d'un atlas routier.

Si on se restreint au cas où tous les arcs ont des poids positifs, il est possible d'employer l'algorithme de Dijkstra pour chaque sommet, ce qui induit une complexité de  $\Theta(|S|^3)$  avec une implantation simple de la file de priorité, ou de  $\Theta(|S|^2 \log |S| + |S| \cdot |A|)$  en utilisant un tas de Fibonacci.

Si certains arcs peuvent avoir des poids négatifs, il faut alors employer l'algorithme de Bellman-Ford qui est encore plus coûteux ( $\Theta(|S|^2 \cdot |A|)$ ).

Il est donc intéressant d'utiliser des algorithmes spécifiques pour répondre à ce problème. Contrairement aux algorithmes de plus court chemin à origine unique, ces algorithmes utilisent une représentation des graphes par matrice d'adjacences. Un graphe orienté pondéré  $G = (S, A)$  avec  $|S| = n$  et une fonction de poids  $\omega : S \times S \rightarrow \mathbb{R}$  sera représenté par une matrice  $W = (\omega_{ij})$  définie par :

$$\omega_{ij} = \begin{cases} 0 & \text{si } i = j \\ \omega(i, j) & \text{si } i \neq j \text{ et } (i, j) \in A \\ \infty & \text{si } i \neq j \text{ et } (i, j) \notin A \end{cases}$$

La sortie des algorithmes sera une matrice  $D = (d_{ij})$  appelée distancier, chaque valeur  $d_{ij}$  représentant la distance entre les sommets  $i$  et  $j$ , c'est-à-dire, en utilisant la notation  $\delta$  introduite

dans la définition 45,  $d_{ij} = \delta(i, j)$ .

Comme dans le cas des algorithmes à origine unique, il est nécessaire de représenter le plus court chemin en utilisant les prédécesseurs de chaque sommet. Dans le cas du plus court chemin pour tout couple de sommets, les prédécesseurs seront stockés dans une matrice de liaison  $\Pi = (\pi_{ij})$  définie par :

$$\pi_{ij} = \begin{cases} null & \text{si } i = j \text{ ou } i \not\rightsquigarrow j \\ \text{le prédécesseur de } j \text{ sur le plus court chemin issu de } i & \text{sinon} \end{cases}$$

Une fois la matrice de liaison calculée, il est possible d'afficher le plus court chemin entre deux sommets du graphe en utilisant l'algorithme 11.

---

**Algorithme 11 : Affichage d'un plus court chemin**

---

**Entrées :** matrice de liaison  $\Pi$   
**Entrées :** sommet  $i$ , sommet  $j$   
**si**  $i = j$  **alors**  
    | afficher  $i$  ;  
**sinon si**  $\pi_{ij} = null$  **alors**  
    | afficher « pas de chemin de  $i$  à  $j$  » ;  
**sinon**  
    | affichagePcc( $\Pi, i, \pi_{ij}$ ) ;  
    | afficher  $j$  ;  
**fin**

---

**Solution récursive**

On définit récursivement une solution au problème de plus court chemin. Soit  $d_{ij}^{(m)}$  la longueur minimale d'un chemin d'au plus  $m$  arcs entre  $i$  et  $j$ . Pour  $m = 0$ , on pose :

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{si } i = j \\ \infty & \text{sinon} \end{cases}$$

Dans ce cas,  $d$  vaut 0 s'il existe un chemin de taille 0 entre  $i$  et  $j$ , c'est-à-dire si  $i$  et  $j$  sont confondus, et  $d$  vaut  $\infty$  sinon. Pour  $m \geq 1$ , on définit  $d_{ij}^{(m)}$  en fonction de  $d_{ij}^{(m-1)}$  :

$$\begin{aligned} d_{ij}^{(m)} &= \min \left( d_{ij}^{(m-1)}, \min_{1 \leq k \leq n} \{d_{ik}^{(m-1)} + \omega_{kj}\} \right) \\ &= \min_{1 \leq k \leq n} \{d_{ik}^{(m-1)} + \omega_{kj}\} \end{aligned}$$

Autrement dit, on cherche à diminuer la distance entre  $i$  et  $j$  en utilisant l'arc  $(k, j)$ .

**Algorithme de plus court chemin**

Prenons en entrée la matrice  $W = (\omega_{ij})$ . L'algorithme de plus court chemin consiste à calculer une suite de matrices  $D^{(1)}, D^{(2)}, D^{(n-1)}$ , chaque matrice  $D^{(m)} = (d_{ij}^{(m)})$ . La matrice  $D^{(n-1)}$  contient alors les longueurs des plus courts chemins pour tous les couples de sommets du graphe.

La coeur de l'algorithme est la procédure *extension* qui a pour rôle de passer des matrices  $D^{(m-1)}$  et  $W$  à la matrice  $D^{(m)}$ . Elle étend donc le résultat temporaire pour des chemins de taille  $m - 1$  à des chemins de taille  $m$ . Cette algorithme défini par 12 suppose que les matrices utilisées ont pour taille  $n \times n$ .

La complexité de la procédure *extension* de 12 est  $\Theta(n^3)$ .

L'algorithme de calcul de plus court chemin revient alors à initialiser les matrices et itérer la procédure *extension*.

**Algorithme 12** : Extension des chemins: extension(matrice  $D$ , matrice  $W$ )

---

```

pour  $i \in [1, n]$  faire
  pour  $j \in [1, n]$  faire
     $d'_{ij} \leftarrow \infty$  ;
    pour  $k \in [1, n]$  faire
       $d'_{ij} \leftarrow \min(d'_{ij}, d_{ik} + \omega_{kj})$  ;
    fin
  fin
fin
return  $D'$  ;

```

---

**Algorithme 13** : Plus court chemin pour tout couple de sommets

---

```

Entrées : matrice  $W$ 
Données :  $\forall m \in [1, n-1] : D^{(m)}$  matrice de taille  $n \times n$ 
 $D^{(1)} \leftarrow W$  ;
pour  $m \in [2, n-1]$  faire
   $D^{(m)} \leftarrow \text{extension}(D^{(m-1)}, W)$  ;
fin
return  $D^{(n-1)}$  ;

```

---

La complexité de l'algorithme 13 est  $\Theta(n^4)$  : il utilise  $n$  fois la procédure *extension*.

On peut montrer que l'opération *extension* est associative. Le résultat cherché est une matrice  $D^{(m)}$  pour  $m \geq n-1$  ( $\forall m \geq n, D^{(m)} = D^{(m-1)}$ ). Il est donc plus rapide de multiplier les matrices  $D^{(m)}$  entre elles au lieu de les multiplier par  $W$ . On calcule alors la suite  $D^{(1)}, D^{(2)}, D^{(4)}, \dots$ .

On obtient alors l'algorithme 14 dont la complexité est  $\Theta(n^3 \log n)$  :  $\log n$  appels à l'algorithme 12.

**Algorithme 14** : Plus court chemin pour tout couple de sommets version 2

---

```

Entrées : matrice  $W$ 
Données :  $\forall m \in [1, n-1] : D^{(m)}$  matrice de taille  $n \times n$ 
Données :  $m$  entier
 $D^{(1)} \leftarrow W$  ;
 $m \leftarrow 1$  ;
tant que  $m < n-1$  faire
   $D^{(2m)} \leftarrow \text{extension}(D^{(m)}, D^{(m)})$  ;
   $m \leftarrow 2m$  ;
fin
return  $D^{(m)}$  ;

```

---

**Algorithme de Floyd-Warshall**

L'algorithme de **Floyd-Warshall** est un algorithme de plus court chemin pour tout couple de sommet de complexité  $\Theta(|S|^3)$ . Cet algorithme a été découvert simultanément par Demoucron en France et par Floyd aux Etats-unis.

Il s'appuie sur la notion de *sommet intermédiaire*. Soit un chemin  $p = (a_1, \dots, a_n)$ . On appelle sommet intermédiaire de  $p$  tout sommet de  $p$  différent de  $a_1$  et de  $a_n$ .

L'algorithme de Floyd-Warshall s'appuie sur la constatation suivante : un plus court chemin ne contient chaque sommet qu'au plus une fois. On considère un chemin  $p$  entre  $i$  et  $j$  dont

les sommets intermédiaires sont compris dans  $\{1, 2, \dots, k\}$ . On note sa distance  $d_{ij}^{(k)}$ . On étudie l'ajout de  $k$  au chemin. Il y a deux possibilités :

- $k \in p$ , alors  $d_{ij}^{(k)} = d_{ij}^{(k-1)}$
- $k \notin p$ , alors  $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$

On en déduit la récursivité suivante :

$$d_{ij}^{(k)} = \begin{cases} \omega_{ij} & \text{si } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{si } k \geq 1 \end{cases}$$

---

**Algorithme 15 : Algorithme de Floyd-Warshall**


---

**Entrées :** matrice  $W$   
**Données :**  $\forall k \in [0, n] : D^{(k)}$  matrice de taille  $n \times n$   
 $D^{(0)} \leftarrow W$  ;  
**pour**  $k \in [1, n]$  **faire**  
  **pour**  $i \in [1, n]$  **faire**  
    **pour**  $j \in [1, n]$  **faire**  
       $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$  ;  
    **fin**  
  **fin**  
**fin**  
**return**  $D^{(n)}$  ;

---

**Construction du plus court chemin**

Pour représenter le plus court chemin entre tous les couples de sommets, on utilise la matrice de liaison  $\Pi$ . Cette matrice est calculée en même temps que les distances. Elle est définie par la formule récursive suivante :

$$\pi_{ij}^{(0)} = \begin{cases} \text{null} & \text{si } i = j \text{ ou } \omega_{ij} = \infty \\ i & \text{sinon} \end{cases}$$

$$\forall k \geq 1, \pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{si } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{sinon} \end{cases}$$

### 3.4 Ordonnancement

La conception et la gestion de projet posent des problèmes de planification (déroulement des tâches) et de contrôle (suivi et réajustement du planning initial). Ces problèmes (dits problèmes d'ordonnancement) peuvent se résoudre par application directe des méthodes de recherche de chemins optimaux (voir section 3.3 p. 48). Résoudre un problème d'ordonnancement revient à choisir l'ordre dans lequel les tâches devront être exécutées de manière à optimiser une certaine fonction objectif<sup>1</sup> tout en respectant des contraintes. Il peut s'agir :

- de contraintes de type *potentiel*, qui sont des contraintes de *localisation temporelle* (la tâche  $i$  doit commencer ou finir avant la date  $t$ ), de *succession* (la tâche  $j$  doit commencer après la tâche  $i$ ) ;
- de contraintes de type *disjonctif*, imposant que deux tâche  $i$  et  $j$  ne soient pas simultanées ;

---

1. Généralement associé au coût du projet, ce qui se traduit souvent par sa durée totale.



- de contraintes de type *cumulatif* concernant l'évolution dans le temps des moyens humains et matériels utilisés. Ces contraintes sont généralement modélisées de manière approchée par des heuristiques.

**Définition 46** (Ordonnancement simple). On dit d'un problème d'ordonnancement qu'il est simple s'il s'agit de trouver un calendrier pour la réalisation d'un ensemble de tâches de durées déterminées, indépendantes des dates de début et des moyens disponibles. On demande à ce calendrier de satisfaire un ensemble de contraintes d'antériorité qui doivent toutes être vérifiées simultanément.

Parmi les techniques d'ordonnancement, on pourra citer :

- la méthode des diagrammes de **Gantt**<sup>2</sup>. Cette technique permet de visualiser les diverses tâches du projet au cours du temps. Elle présente l'avantage d'une représentation visuelle pour de petits projets. Cependant, le diagramme devient impossible à manipuler pour de gros projets.
- la méthode PERT<sup>3</sup> qui consiste à créer un graphe de dépendance des tâches. Cette méthode permet d'identifier facilement les chemins critiques ainsi que les dates au plus tôt et au plus tard pour chaque tâche. Cependant, les graphes produits deviennent rapidement impossible à afficher et à imprimer.
- la méthode MPM, développée par Bernard **Roy**<sup>4</sup> qui est semblable à la méthode PERT, tout en étant parfois jugée plus souple.

En utilisant uniquement des contraintes qui se ramènent au type potentiel, et à condition que les contraintes soient compatibles, les méthodes PERT et MPM permettent :

- d'établir un ordonnancement ;
- de minimiser le temps total de réalisation de l'objectif ;
- de déterminer les tâches critiques ;
- d'évaluer les marges sur les tâches non critiques.

La méthode *PERT* s'appuie sur un graphe orienté, sans boucle, dont les sommets constituent les événements et les arcs les opérations. Les arcs sont pondérés par la durée des tâches. Une fois le graphe construit, il est possible d'obtenir le chemin critique par une recherche de plus long chemin<sup>5</sup>.

La méthode *MPM* utilise un graphe dont les sommets représentent les opérations et les arcs les contraintes. Le poids d'un arc  $u = (a, b)$  représente le temps minimum devant s'écouler entre le début de la tâche représentée par  $a$  et le début de la tâche représentée par  $b$ .

### 3.4.1 Méthode PERT

**Définition 47** (Réseau PERT). Soit un problème d'ordonnancement simple. On lui associe un réseau PERT  $R = (S, A, d)$  dans lequel  $S$  correspond aux étapes du projet,  $A$  aux tâches, et  $d$  aux durées des tâches. Ce réseau devra toujours comprendre un sommet de départ, noté  $D$ , et un sommet de fin, noté  $F$ . Les tâches pourront être réelles ou fictives (dans ce dernier cas, elle servent uniquement à marquer des contraintes d'antériorité, et leur durée pourra être nulle).

**Définition 48** (Date au plus tôt, date au plus tard). À chaque sommet  $x$  d'un réseau PERT  $R = (S, A, d)$ , on associe une date au plus tôt, notée  $\pi(x)$ , et une date au plus tard, notée  $\eta(x)$ , définies par :

2. Développé entre 1910 et 1915 par l'ingénieur Américain Henry Laurence Gantt.

3. Program (ou Project) Evaluation and Review Technique, développée aux États-Unis dans les années 1950.

4. Méthode des Potentiels et antécédents Métra, développée en 1958.

5. Cette recherche est possible en temps polynomial car le graphe ne contient pas de cycle ; il est donc possible d'adapter l'algorithme de Bellman-Ford.

- $\pi(x)$  est la date minimum à laquelle la tâche  $x$  peut être atteinte, sachant que  $\pi(D) = 0$  ;
- $\eta(x)$  est la date maximum à laquelle la tâche  $x$  peut être atteinte sans que cela n'induisse un retard dans le projet.

**Définition 49** (Tâche critique, chemin critique). Une tâche est dite critique si tout retard lors de son exécution entraîne un retard égal dans la réalisation du projet. Un chemin dont tous les arcs correspondent à des tâches critiques est dit critique.

**Théorème 13.** Soit un réseau de PERT  $R = (S, A, d)$ .

- $\forall x \in S, \pi(x)$  est la longueur du plus long chemin de  $D$  à  $x$ .
- Soient  $x \in S$  et  $\Upsilon$  un plus long chemin de  $x$  à  $F$ . Soit  $d(\Upsilon) = \sum_{a \in \Upsilon} d(a)$ . Alors  $\eta(x) = \pi(F) - d(\Upsilon)$ .
- Une tâche correspondant à l'arc  $u = (x, y)$  peut voir sa durée reprogrammée d'un délai  $\delta(u) = \eta(y) - \pi(x) - d(u)$  sans que ce retard ne se répercute sur la durée du projet. Les tâches critiques vérifient  $\delta(u) = 0$ .
- Un chemin de  $D$  à  $F$  est critique si et seulement si il s'agit d'un plus long chemin de  $D$  à  $F$  dans  $R$ .

L'algorithme de recherche des dates au plus tôt et des dates au plus tard est une variante de l'algorithme de Bellman-Ford. Il est représenté par l'algorithme 16.

---

**Algorithme 16 :** Algorithme de PERT: Pert(Réseau  $R = (S, A, d)$ )

---

```

 $E \leftarrow \{D\}$  ;
 $\pi(D) \leftarrow 0$  ;
tant que  $\exists x \notin E / \forall y \in \text{Pred}(x), y \in E$  faire
     $\pi(x) \leftarrow \max_{u/T(u)=x} (\pi(I(u)) + d(u))$  ;
     $E \leftarrow E \cup \{x\}$  ;
fin
 $E \leftarrow \{F\}$  ;
 $\eta(F) \leftarrow \pi(F)$  ;
tant que  $\exists x \notin E / \forall y \in \text{Succ}(x), y \in E$  faire
     $\eta(x) \leftarrow \min_{u/I(u)=x} (\eta(T(u)) - d(u))$  ;
     $E \leftarrow E \cup \{x\}$  ;
fin
pour tous  $u \in A$  faire
     $\delta(u) \leftarrow \eta(T(u)) - \pi(I(u)) - d(u)$  ;
fin

```

---

### 3.4.2 Méthode MPM

La méthode MPM s'appuie également sur un réseau  $R = (S, A, w)$  où  $S$  correspond aux opérations,  $A$  aux relations d'antériorité entre les opérations, et  $w$  à la durée de l'opération antérieure.

Un algorithme similaire à 16 peut être utilisé pour la résolution. On suppose que chaque sommet  $s$  possède un ensemble de successeurs  $\text{Succ}$  et un ensemble de prédécesseurs  $\text{Pred}$ . On notera  $\pi_y(x)$  le temps au plus tôt du prédécesseur  $y$  de  $x$ .

L'algorithme 18 s'applique après 17. Le principe est de parcourir le graphe en partant de la tâche finale  $F$ . Parmi les prédécesseurs d'un sommet  $x$ , on marque tous les sommets  $y$  tels que  $\pi(y) + w(y, x) = \pi(x)$ .

**Algorithme 17 :** Algorithme de MPM: Mpm(Réseau  $R = (S, A, d)$ )

---

```

 $E \leftarrow \{D\}, \pi(D) \leftarrow 0, \forall x \in S, \forall y \in Pred(x), \pi_y(x) \leftarrow +\infty ;$ 
tant que  $E \neq \emptyset$  faire
     $s \leftarrow E.pop ;$ 
    pour tous  $x \in Succ(s)$  faire
         $\pi_s(x) \leftarrow \pi(s) + w(s, x) ;$ 
         $\pi(x) = \max_{y \in Pred(x)} \pi_y(x) ;$ 
        si  $\pi(x) \neq +\infty$  alors
             $E.add(x) ;$ 
        fin
    fin
fin

```

---

**Algorithme 18 :** Chemin critique MPM: CrMpm(Réseau  $R = (S, A, d)$ )

---

```

 $E \leftarrow \{F\}, \text{Marquer } F ;$ 
tant que  $E \neq \emptyset$  faire
     $s \leftarrow E.pop ;$ 
    pour tous  $x \in Pred(s)$  faire
        si  $\pi_x(s) + w(x, s) = \pi(s)$  alors
            Marquer  $x$  ,  $E.add(x) ;$ 
        fin
    fin
fin

```

---

**3.4.3 Exemple**

L'exemple traité dans cette section est tiré de [Faure *et al.*, 2009]. Un éditeur veut passer commande d'un ouvrage à un auteur scientifique. Les étapes à suivre sont indiquées dans le tableau 3.1, avec leur durée (en quinzaines) et la mention des opérations qui doivent précéder chacune d'entre elles.

TABLE 3.1: Opérations à effectuer

	Opérations	Durée	Opérations antérieures
a	Approbation du plan d'ouvrage	1	/
b	Signature du contrat	1	a
c	Remises du manuscrit	12	b
d	Approbation du comité de lecture	2	c
e	Composition du texte	3	d
f	Correction par les correcteurs de l'imprimerie	1	e
g	Clichage et tirage hors-texte	3	d
h	Exécution des dessins, des figures	4	d
i	Révision des dessins par l'auteur	1	h
j	Correction de dessins ; clichage des figures	2	i
k	Première correction des épreuves par l'auteur	2	f
l	Exécution des premières corrections à l'imprimerie	1	k
...			

...			
m	Seconde correction des épreuves par l'auteur ; indication de l'emplacement des clichés ; approbation des hors-texte	2	g,j,l
n	Exécution des secondes corrections à l'imprimerie ; mise en page	1	m
o	Tirage du livre	2	n
p	Établissement de la prière d'insérer, des listes d'exemplaires presse de d'hommage	1	m
q	Pliage	1	o
r	Brochage	1	q
s	Reliure de certains exemplaires	2	q
t	Impression de la prière d'insérer	1/2	p
u	Envoi des exemplaires de presse	1/4	r, t
v	Envoi des hommages	1/8	s, t
w	Envoi des contingents aux librairies	1/2	r, s

### Résolution par la méthode PERT

Afin de résoudre ce problème d'ordonnancement par la méthode PERT, il faut tout d'abord créer le graphe de PERT. Celui-ci sera composé de sommets représentant les événements (inconnus au départ) et d'arcs représentant les tâches. Il est généralement nécessaire d'introduire des tâches fictives (inconnues au départ) afin de traduire les contraintes.

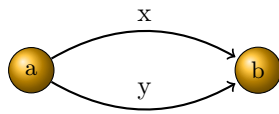


FIGURE 3.9 – Graphe multiple

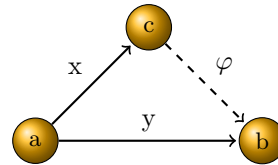


FIGURE 3.10 – Ajout de tâche fictive

En traduisant directement la liste d'opérations en graphe, on aboutit généralement à un graphe multiple (voir figure 3.9), c'est à dire un graphe dans lequel un couple de sommets peut être relié par plusieurs arêtes. Pour palier ce problème, il convient d'insérer un sommet supplémentaire, comme représenté par la figure 3.10. La valeur de  $\varphi$  sera nulle si  $c$  sert uniquement à casser la multiplicité du graphe. Elle pourra être non nulle pour représenter la contrainte « A ne peut débuter qu'au moins 5 jours après le début du projet ».

Bien que la construction du graphe soit aisée dans cet exemple, elle est souvent délicate dans les cas plus complexes et représente une des faiblesses de la méthode PERT.

À partir du tableau 3.1, il est possible de créer la liste des événements :

TABLE 3.2: Évènements

N°	Évènement
0	Début des opérations
1	Plan approuvé
2	Contrat signé
3	Manuscrit remis
4	Manuscrit approuvé
...	

...	
5	Texte composé
6	Texte corrigé par l'imprimerie
7	Première épreuves corrigées par l'auteur
8	Dessins exécutés
9	Dessins revus par l'auteur
10	Premières corrections exécutées par l'imprimerie, secondes épreuves tirées ; clichage terminé pour les figures ; impression achevée pour les hors-texte ; envoi à l'auteur des secondes épreuves, après exécution des premières corrections, de spécimens des hors-texte et des épreuves des clichés des dessins
11	Retour des secondes épreuves
12	Secondes corrections exécutées
13	Tirage exécuté
14	Pliage exécuté
15	Brochage exécuté
16	Reliure exécutée
17	Prière d'insérer, liste des services de presse et des hommages remises
18	Prière d'insérer imprimée
19	Début de l'envoi des contingents aux libraires
20	Début de l'envoi des hommages
21	Début de l'envoi des exemplaires de presse
22	Achèvement des opérations

Le graphe correspondant aux tables 3.1 et 3.2 est représenté figure 3.11. On remarquera les tâches fictives représentées en pointillés.

L'application de l'algorithme 16 permet d'aboutir au tableau 3.3 représentant les temps au plus tôt et au plus tard de toutes les tâches, ainsi qu'au tableau 3.4 représentant la marge associée à chaque opération. Notons que les opérations dont la marge est nulle sont les tâches critiques.

La résolution d'un problème d'ordonnancement à partir d'un réseau de PERT est donc très simple. L'algorithme 16 étant équivalent à celui de Bellman-Ford, il permettra de résoudre rapidement des problèmes avec des dizaines de milliers d'éléments. Par contre, la difficulté de cette méthode est dans l'obtention du réseau. En particulier, l'ajout de tâches fictives qui ne modifient pas le problème de départ s'avère très difficile lorsqu'il faut manipuler plusieurs milliers d'opérations.

#### Résolution par la méthode MPM

À la différence de la méthode PERT, la méthode MPM ne repose pas sur une définition d'événements et ne nécessite pas l'introduction de tâches fictives. De plus, la représentation du graphe n'est pas indispensable et une représentation des prédécesseurs suffit.

Un telle représentation est définie par la table 3.5, et la représentation équivalente sous forme de graphe est définie figure 3.12. Le graphe est uniquement donné à titre d'information <sup>6</sup>.

La résolution du problème peut s'effectuer directement sur le tableau 3.5 grâce à l'algorithme 17. Les valeurs des  $\pi_y(x)$  sont remplies au fur et à mesure ; lorsque les valeurs sont connues pour tous les  $y$  pour un  $x$  donné, on calcule  $\pi(x) = \max_{y \in \text{Pred}(x)} \pi_y(x)$ .

Appliquons la méthode MPM sur le tableau 3.5 :

- Le temps de  $D$  est 0, ce qui nous permet de calculer le temps de  $a$  :  $0+0=0$
- La première case du tableau devient :

0	a
0	D : 0

6. On rappelle qu'il n'est pas indispensable au bon déroulement de l'algorithme.

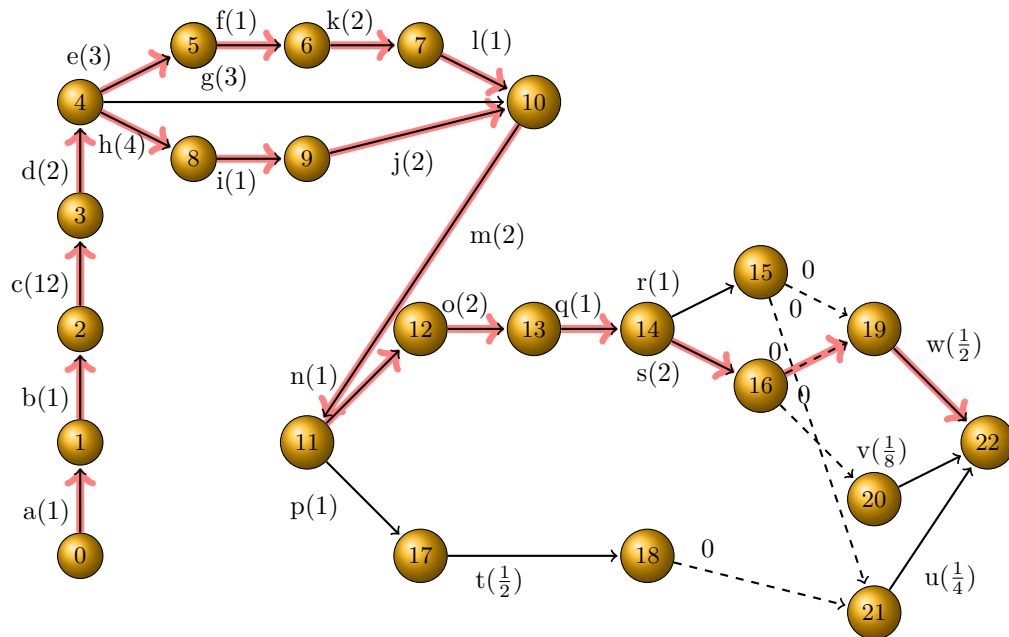


FIGURE 3.11 – Graphe PERT

TABLE 3.3 – Résolution du PERT

N°	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$\pi$	0	1	2	14	16	19	20	22	20	21	23	25	26	28
$\sigma$	0	1	2	14	26	29	20	22	20	21	23	25	26	28
N°	14	15	16	17	18	19	20	21	22					
$\pi$	29	30	31	26	26.5	31	31	30	31.5					
$\sigma$	29	30	31	30.75	31.25	31	31.25	31.25	31.5					

TABLE 3.4 – Marges des différentes opérations

Nom	a	b	c	d	e	f	g	h	i	j	k	l	m	n
$\delta$	0	0	0	0	0	0	4	0	0	0	0	0	0	0
Nom	o	p	q	r	s	t	u	v	w					
$\delta$	0	4.75	0	1	0	4.75	1.25	0.125	0					

Opérations	a		b		c		d		e		f	
Préalables		D : 0		a : 1		b : 1		c : 12		d : 2		e : 3
Opérations	g		h		i		j		k		l	
Préalables		d : 2		d : 2		h : 4		i : 1		f : 1		k : 2
Opérations	m		n		o		p		q		r	
Préalables		g : 3 j : 2 l : 1		m : 2		n : 1		m : 2		o : 2		q : 1
Opérations	s		t		u		v		w		F	
Préalables		q : 1		p : 1		r : 1 t : 0.5		t : 0.5 s : 2		r : 1 s : 2		u : 0.25 v : 0.125 w : 0.5

TABLE 3.5 – Prédécesseurs MPM

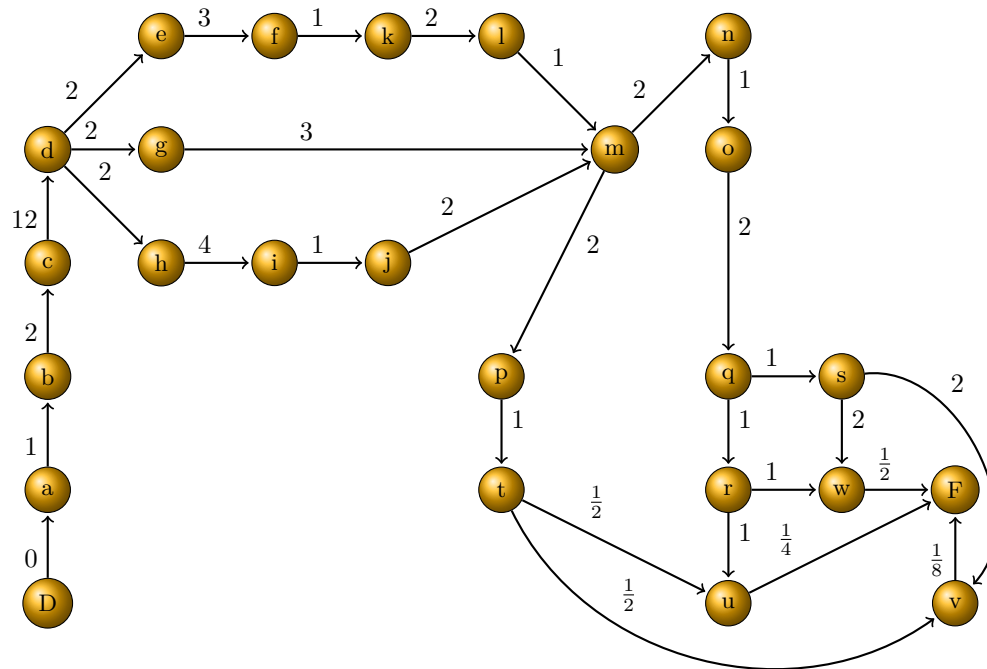


FIGURE 3.12 – Graphe MPM

- On répercute alors la date au plus tôt de  $a$  dans la case suivante, ce qui nous permet de calculer la date au plus tôt de  $b$  :

1	b
0	a : 1

- Les dépendances se poursuivent linéairement jusqu'à l'évènement  $m$  qui dépend de  $g$ ,  $j$  et  $l$ . Il faut donc avoir calculé ces trois dates au plus tôt avant de calculer celle de  $m$ . On constate que la date au plus tôt de  $m$  provient à la fois de  $j$  et de  $l$ .

23	m
16	g : 3
21	j : 2
22	l : 1

À la fin du parcours, on obtient le tableau 3.6.

Pour obtenir les tâches critiques, on applique l'algorithme 18. Les tâches ainsi marquées sont soulignées dans le tableau.

On remarque que les dates au plus tôt fournies par la méthode PERT et la méthode MPM sont identiques.

#### 3.4.4 Intervention de modifications

Dans cette section, nous allons étudier la prise en compte de modifications dans les méthodes PERT et MPM. Nous reprenons l'exemple décrit en section 3.4.3 en prenant en compte les modifications suivantes :

- le tirage  $o$  peut commencer dès que la tâche  $n$  est réalisée à 50% ;
- le pliage  $q$  peut commencer avec un décalage d'une quinzaine uniquement sur le début du tirage  $o$ , mais alors la durée de  $q$  passe à 1.5 ; la tâche  $q$  reste précédée par  $n$  ;

Opérations	0	a	1	b	2	c	14	d	16	e	19	f
Préalables	0	<u>D</u> : 0	0	<u>a</u> : 1	1	<u>b</u> : 1	2	<u>c</u> : 12	14	<u>d</u> : 2	16	<u>e</u> : 3
Opérations	16	g	16	h	20	i	21	j	20	k	22	l
Préalables	14	<u>d</u> : 2	14	<u>d</u> : 2	16	<u>h</u> : 4	20	<u>i</u> : 1	19	<u>f</u> : 1	20	<u>k</u> : 2
Opérations	23	m	25	n	26	o	25	p	28	q	29	r
Préalables	16	<u>g</u> : 3	23	<u>m</u> : 2	25	<u>n</u> : 1	23	<u>m</u> : 2	26	<u>o</u> : 2	28	<u>q</u> : 1
	21	<u>j</u> : 2										
	22	<u>i</u> : 1										
Opérations	29	s	26	t	30	u	31	v	31	w	31.5	F
Préalables	28	<u>q</u> : 1	25	<u>p</u> : 1	29	<u>r</u> : 1	26	<u>t</u> : 0.5	29	<u>r</u> : 1	30	<u>u</u> : 0.25
					26	<u>t</u> : 0.5	29	<u>s</u> : 2	29	<u>s</u> : 2	31	<u>v</u> : 0.125
											31	<u>w</u> : 0.5

TABLE 3.6 – Résolution MPM

- on ne désire demander à l’auteur le prière d’insérer et les listes d’envoi des exemplaires de presse et des hommages que lorsque le pliage est achevé :  $q$  précède  $p$  ;
- le brochage  $r$  et la reliure  $s$  sont toujours précédés par  $o$ .

#### Prise en compte de modifications : PERT

Afin de prendre en compte ces modifications dans le réseau PERT, il faut effacer des arcs, en créer d’autres et introduire de nouveaux événements. Le graphe résultant est représenté figure 3.13. Nous n’avons représenté ici que la partie suivant l’évènement 11.

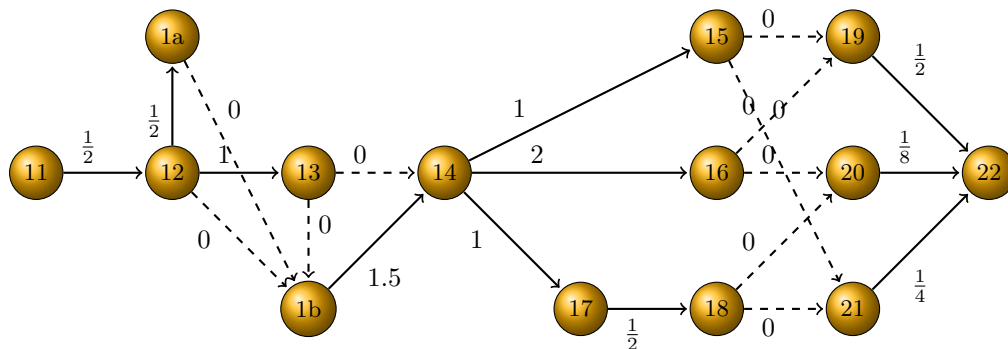


FIGURE 3.13 – Modification du PERT initial

On constate que la prise en compte des modifications nécessite une restructuration assez lourde du graphe.

#### Prise en compte de modifications : MPM

Afin de prendre en compte ces modifications avec la méthode MPM, il suffit de modifier quelques valeurs dans le tableau 3.5. On obtient alors le tableau 3.7 (on ne prend en compte que la tâches postérieures à  $n$ ).

On remarque qu’il n’a pas été nécessaire d’ajouter de sommet. La nouvelle représentation du graphe ne sera pas fournie ici (elle est très simple à obtenir et ne sert pas à la résolution du problème).

La prise en compte de modifications avec la méthode MPM n’implique pas une réécriture profonde du graphe, et il est assez rapide de mettre la solution à jour. La souplesse de la méthode MPM en fait donc une méthode à privilégier en recherche opérationnelle.



25	n	25.5	o	28	p	26.5	q	28	r	28	s
23	<u>m</u> : 2	25	<u>n</u> : $\frac{1}{2}$	26.5	q : 1.5	25	n : 1	25.5	o : 2	25.5	o : 2
				23	m : 2	25.5	<u>o</u> : 1	26.5	q : 1.5	26.5	<u>q</u> : 1.5
29	t	29.5	u	30	v	30	w	30.5	F		
28	p : 1	28	r : 1	28	s : 2	28	r : 1	29.5	u : $\frac{1}{4}$		
		29	t : $\frac{1}{2}$	29	t : $\frac{1}{2}$	28	<u>s</u> : 2	30	v : $\frac{1}{8}$		
								30	<u>w</u> : $\frac{1}{2}$		

TABLE 3.7 – Prédecesseurs MPM après modification



## Graphes non orientés

### Sommaire

<b>4.1 Généralités</b>	<b>67</b>
<b>4.2 Arbres couvrants</b>	<b>67</b>
4.2.1 Algorithme de Kruskal	69
4.2.2 Algorithme de Prim	71
<b>4.3 Graphes planaires</b>	<b>74</b>
<b>4.4 Coloration de graphes</b>	<b>76</b>
4.4.1 Quelques propriétés des colorations de graphe	77
4.4.2 Exemple d'application	78
4.4.3 Théorème des quatre couleurs	78
4.4.4 Algorithme de coloration de graphe	80

### 4.1 Généralités

DANS ce chapitre, nous nous intéresserons aux propriétés des graphes non orientés, c'est-à-dire de graphes dans lesquels la relation d'adjacence est symétrique. Quelques problèmes classiques se posent pour ce type de graphe, et nous étudierons les algorithmes permettant d'y répondre.

### 4.2 Arbres couvrants

Dans cette section, nous allons définir formellement la notion d'arbre et étudier différentes manières d'extraire un arbre d'un graphe.

**Définition 50** (Arbre). Un arbre est un graphe connexe acyclique.

**Théorème 14.** Soit  $G = (S, A)$  un graphe sur  $m = |S|$  sommets.

1. Si  $G$  est connexe,  $|A| \geq m - 1$
2. Si  $G$  est sans cycle,  $|A| \leq m - 1$

PREUVE :

Supposons que le graphe  $G = (S, A)$  soit construit par ajout successifs d'arcs à partir du graphe  $(S, \emptyset)$ .

1. À chaque itération, le nombre de composantes connexes décroît au plus de 1. Le graphe initial possédant  $m$  composantes connexes, il faut au minimum  $m - 1$  étapes pour rendre le graphe connexe.

2. Si  $G$  est sans cycle, alors tout graphe partiel de  $G$  est sans cycle. À chaque itération, le nombre de composantes connexes a été diminué de un (il ne peut rester stable que par l'ajout d'un cycle). Comme  $G$  possède au moins une composante connexe, il ne peut y avoir eu qu'au plus  $m - 1$  étapes.

□

**Théorème 15.** Soit  $G = (S, A)$  un graphe sur  $m = |S|$  sommets. Les propriétés suivantes sont équivalentes et caractérisent un arbre :

1.  $G$  est connexe et sans cycle ;
2.  $G$  est connexe et minimal pour cette propriété (si on supprime un arc de  $G$  il n'est plus connexe) ;
3.  $G$  est connexe et possède  $m - 1$  arcs ;
4.  $G$  est sans cycle et maximal pour cette propriété (si on ajoute un arc à  $G$  il possède un cycle) ;
5.  $G$  est sans cycle et possède  $m - 1$  arcs ;
6. il existe dans  $G$  un chaîne et une seule joignant tout couple de sommets.

PREUVE :

- $1 \Rightarrow 3, 5$  selon le théorème 14.
- $3 \Rightarrow 2$  selon le théorème 14.
- $5 \Rightarrow 4$  selon le théorème 14.
- $4 \Rightarrow 3$  : si  $G$  n'était pas connexe, il serait possible d'ajouter un arc à  $G$  sans créer de cycle.  $G$  est donc connexe et sans cycle, et selon le théorème 14 il possède donc  $m - 1$  arcs.
- $2 \Rightarrow 6$  :  $G$  étant connexe, il existe au moins une chaîne reliant tout couple de sommets de  $G$ . S'il existait un couple de sommets reliés par deux chaînes, alors la suppression d'un arc le long d'une de ces chaînes ne détruirait pas la connexité de  $G$ , et  $G$  ne serait pas connexe minimal.
- $6 \Rightarrow 1$  : S'il existe une chaîne reliant tout couple de sommets de  $G$ , le graphe est connexe. Si  $G$  possédait un cycle, il existerait au moins deux chaînes reliant tout couple de sommets de ce cycle.

□

**Définition 51** (Arbre couvrant). Soit  $G = (S, A)$  un graphe non orienté pondéré. On appelle arbre couvrant de  $G$  tout sous-graphe de  $G$  qui est un arbre. On appelle poids de l'arbre couvrant la somme des poids de ses arcs.

La notion d'arbre couvrant est étudiée par exemple dans [Fournier, 2007a; Sakarovitch, 1984a; Cormen *et al.*, 1994].

**Définition 52** (Poids d'un graphe partiel). Soient  $G = (S, A)$  un graphe non orienté pondéré (de pondération  $\omega$ ) et  $A' \subset A$ . On appelle poids le graphe partiel  $G' = (S, A')$  la quantité suivante :

$$\omega(A') = \sum_{u \in A'} \omega(u).$$

Soit  $G$  un graphe pondéré non orienté. Il est souvent intéressant de chercher un arbre couvrant de poids minimum ou maximum. On se ramène trivialement d'un problème à l'autre en prenant l'opposé du poids des arcs.

Par la suite, nous nous restreindrons au problème des arbres couvrants de poids minimum. On cherche alors un ensemble d'arcs  $A'$  de  $G$  minimisant la quantité  $\omega(A')$ .

**Remarque :**

tout graphe partiel de  $G$  qui est un arbre possède le même nombre d'arcs :  $m - 1$ . On peut ajouter une même quantité  $\alpha$  aux poids de tous les arcs sans changer le classement des arbres en terme de poids. Il est donc toujours possible de se ramener au cas  $\forall u \in A, \omega(u) \geq 0$ .

La figure 4.1 représente un graphe et un arbre couvrant minimum sur ce graphe.

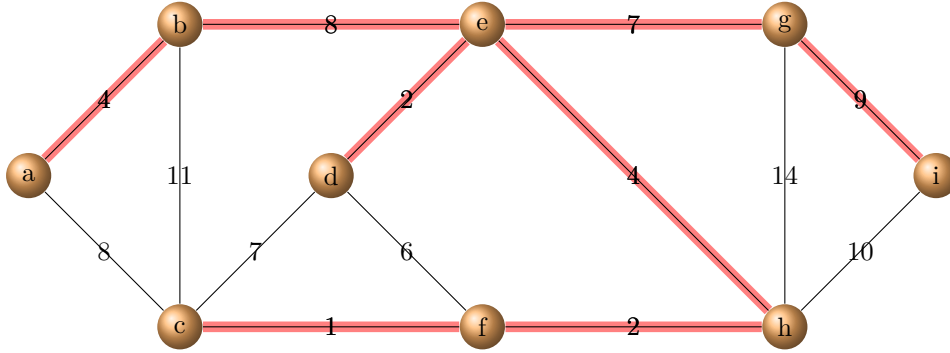


FIGURE 4.1 – Arbre couvrant minimum

Il existe plusieurs algorithmes permettant de construire un arbre couvrant minimum. Ces algorithmes utilisent des stratégies gloutonnes, c'est-à-dire effectuent ce qui semble être le meilleur choix à chaque itération sans jamais remettre ce choix en question. Une stratégie gloutonne n'aboutit en général pas à une solution optimale, mais dans le cas des arbres couvrant minimaux, certaines stratégies gloutonnes permettent de l'obtenir.

Ceci est en particulier le cas pour les algorithmes de **Kruskal** et de **Prim**.

**4.2.1 Algorithme de Kruskal**

**Théorème 16.** Soit  $G = (S, A)$  un graphe connexe sans boucle, et  $T \subset A$  tel que  $G' = (S, T)$  soit un arbre. Étant donné  $u \in A - T$ , on désigne par  $C_u$  l'ensemble des arcs de  $G'$  reliant les extrémités de  $u$  (un exemple est présenté figure 4.2). Soient  $v \in T$  et  $G'' = (S, T - \{v\})$ .  $G''$  est composé de deux composantes connexes. On note  $\Omega_v$  l'ensemble des arcs de  $A - T$  dont chaque extrémité appartient à une des composantes connexes de  $G''$  (un exemple est présenté figure 4.3). Alors les deux conditions suivantes sont équivalentes et caractérisent un arbre couvrant minimum :

$$\forall u \in A - T, \omega(u) \geq \max_{w \in C_u} (\omega(w)) \quad (4.1)$$

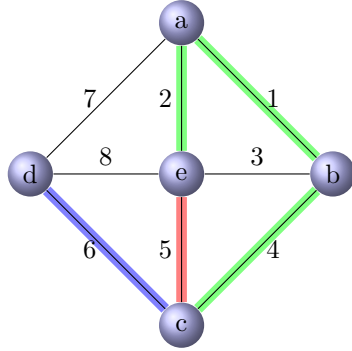
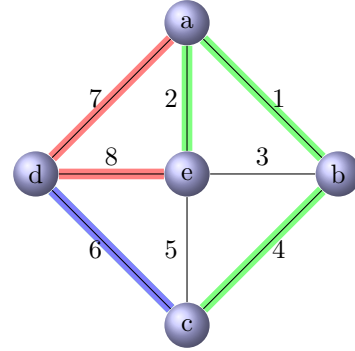
$$\forall v \in T, \omega(v) \leq \min_{w \in \Omega_v} (\omega(w)) \quad (4.2)$$

Autrement dit, si on considère un arc  $u$  hors de l'arbre couvrant, on appelle  $C_u$  le chemin (unique) de l'arbre couvrant reliant les extrémités de  $u$ , alors tous les poids des arcs de  $C_u$  sont inférieurs au poids de  $u$ . De même, si on considère un arc  $v$  de l'arbre couvrant, l'arbre privé de  $v$  est divisé en deux composantes connexes. Il existe des arcs dont chaque extrémité est dans une composante connexe (on note leur ensemble  $\Omega_v$ ). Tous les poids des arcs de  $\Omega_v$  sont alors supérieurs au poids de  $v$ .

*Exemple 22.* La figure 4.2 représente un graphe, un arbre couvrant ( $T = \{(e, a), (a, b), (b, c), (c, d)\}$ ), et un choix d'arc  $u = (c, e)$ . L'ensemble  $C(u) = \{(e, a), (a, b), (b, c)\}$ .

Dans la figure 4.3, on représente le même graphe et le même arbre couvrant. Maintenant, on

choisit l'arc  $v = (c, d)$ . Les deux composantes connexes sont  $\{e, a, b, c\}$  et  $\{d\}$ , et on a alors  $\Omega_v = \{(d, e), (d, a)\}$ .

FIGURE 4.2 – Arbre couvrant,  $u$  et  $C_u$ FIGURE 4.3 – Arbre couvrant,  $v$  et  $\Omega_v$ 


---

**Algorithme 19 :** Recherche d'arbre couvrant minimum par l'algorithme de Kruskal

---

```
//  $A = \{u_1, \dots, u_n\}$  tels que  $\omega(u_1) \leq \omega(u_2) \leq \dots \leq \omega(u_n)$ 
 $T \leftarrow \emptyset$ ;
pour tous  $i \in [1, n]$  faire
    si  $(S, T \cup \{u_i\})$  ne contient pas de cycle alors
        |  $T \leftarrow T \cup \{u_i\}$ ;
    fin
fin
```

---

**PREUVE** du théorème 16:

▷ Les propositions 4.1 et 4.2 sont des conditions nécessaires :

- 4.1 :** si  $u \in A - T, w \in C_u$ , alors  $(S, A \cup \{u\} - \{w\})$  est sans cycle et possède  $m - 1$  arcs ; il s'agit d'un arbre.
- 4.2 :** si  $v \in T, w \in \Omega_v$ , alors  $(S, T \cup \{w\} - \{v\})$  est sans connexe et possède  $m - 1$  arcs ; il s'agit d'un arbre.

▷ Montrons maintenant que la condition 4.1 est suffisante :

- Soit  $T_1$  tel que  $G_1 = (S, T_1)$  soit un arbre et que  $G_1$  satisfasse 4.1.
- Soit  $T_2$  tel que  $G_2 = (S, T_2)$  soit un arbre couvrant minimal.
- On suppose de plus que  $\omega(T_2) < \omega(T_1)$  (on peut supposer sans perte de généralité que  $G_2$  est un arbre couvrant minimum).
- Soient  $u \in T_2 - T_1$  et  $C_u$  (dans  $G_1$ ) défini dans le théorème 16.

Le graphe  $(S, T_2 - \{u\})$  est non connexe : il s'agit d'un arbre auquel on a retiré un arc. Il est formé de deux composantes connexes qu'on note  $(S', T'_2)$  et  $(S'', T''_2)$ .

Alors,

$$\exists v \in C_u / I(v) \in S' \text{ et } T(v) \in S'' \text{ (remarque : } v \in T_1)$$

Selon l'hypothèse 4.1,  $\omega(u) \leq \omega(v)$  et selon la condition nécessaire 4.2,  $\omega(u) \geq \omega(v)$ .

Donc,  $\omega(u) = \omega(v)$ .

On considère alors  $T_3 = T_2 \cup \{v\} - \{u\}$ .  $(S, T_3)$  est un arbre et  $\omega(T_3) = \omega(T_2)$ .

On recommence alors la même opération tant que  $T_2 - T_1 \neq \emptyset$ . On obtient alors un ensemble

d'arc  $T_k$  tel que  $T_k = T_1$  et  $\omega(T_k) < \omega(T_1)$  ce qui est une contradiction.

Le caractère suffisant de la condition 4.1 se démontre de manière similaire : on choisit  $v \in T_1 - T_2$  et  $u \in \Omega_v$  dans  $G_2$ .  $\square$

L'algorithme de Kruskal s'appuie sur le théorème 16. Les arcs du graphe sont numérotés par ordre de poids croissants,  $(u_1, \dots, u_n)$  et on construit alors progressivement le graphe partiel  $(S, T)$  qui est un arbre de poids minimum en partant de  $T_0 = \emptyset$ .  $T_i$  est construit en ajoutant l'arc  $u_i$  à  $T_{i-1}$  si le graphe  $(S, T_{i-1} \cup \{u_i\})$  ne contient pas de cycle.  $(X, T_n)$  sera alors un arbre, et selon le théorème 16, il sera de poids minimum.

#### Remarques :

- les valeurs des poids des arcs n'ont aucune importance pour la construction de l'arbre couvrant minimum. Seul leur ordre importe ;
- l'algorithme 19 reste superficiel, et en particulier il ne détaille pas la fonctionnalité « détection de cycle ».

#### Variante :

une variante de l'algorithme de Kruskal consiste à trier les arcs dans l'ordre décroissant, de partir du graphe  $G$  et de supprimer tous les arcs  $u_1, \dots, u_n$  si la suppression de  $u_i$  conserve la connexité du graphe.

*Exemple 23* (Algorithme de Kruskal). La figure 4.4 représente un graphe dont nous voulons extraire un arbre couvrant minimum en utilisant l'algorithme de Kruskal. Les arcs sont triés par ordre croissant, puis insérés dans le graphe partiel s'ils n'induisent pas de cycle :  $(a, b)(e, d)(a, d)(d, c)$ . Le graphe possédant 5 sommets, l'arbre couvrant ne peut contenir que 4 arcs. L'arc  $(a, e)$  génère le cycle  $a, d, e$  et il n'est donc pas ajouté. Le résultat obtenu est représenté figure 4.5.

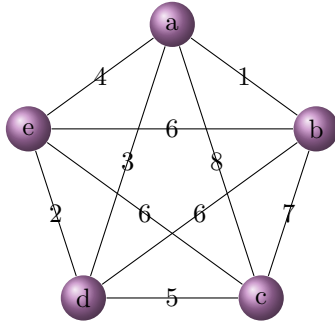


FIGURE 4.4 – Graphe

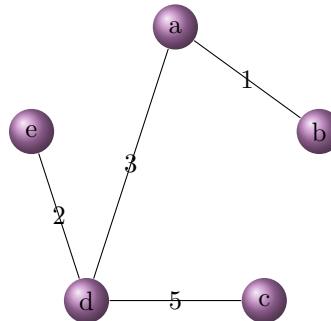


FIGURE 4.5 – Arbre couvrant minimum

#### Implantation de l'algorithme de Kruskal

Pour implanter l'algorithme de Kruskal, il est nécessaire de pouvoir déterminer si l'ajout d'un arc au graphe partiel introduit un cycle.

Pour le déterminer, il est possible d'utiliser un marquage des sommets. À chaque sommet est associé une classe  $\eta_i$  (initialement  $\eta_i = i$ ). Un arc peut être ajouté entre  $i$  et  $j$  si  $\eta_i \neq \eta_j$ . La classe de tous les sommets  $\eta_j$  devient alors  $\eta_i : \forall k \in \{x/\eta_x = \eta_j\}, \eta_k \leftarrow \eta_i$ .

On obtient alors l'algorithme 20.

#### 4.2.2 Algorithme de Prim

**Théorème 17.** Soient  $G = (S, A)$  un graphe connexe, et  $x$  un sommet. On pose

$$A_x = \{u \in A / u \text{ adjacent à } x, u \text{ n'est pas une boucle}\}$$

**Algorithme 20** : Recherche d'arbre couvrant minimum par l'algorithme de Kruskal v2

---

**Entrées** : graphe  $G$   
 //  $A = \{u_1, \dots, u_n\}$  tels que  $\omega(u_1) \leq \omega(u_2) \leq \dots \leq \omega(u_n)$   
**Données** :  $\forall i \in [1, n], \eta_i = i$   
 $T \leftarrow \emptyset$  ;  
**pour tous**  $i \in [1, n]$  **faire**  
     **si**  $u_i = (s_j, s_k)$  est non traitée et  $\eta_j \neq \eta_k$  **alors**  
          $T \leftarrow T \cup \{u_i\}$  ;  
         **pour tous**  $l$  tel que  $\eta_l = \eta_k$  **faire**  
              $\eta_l \leftarrow \eta_j$  ;  
         **fin**  
     **fin**  
**fin**

---

et  $u(x)$  défini par :

$$\omega(u(x)) = \min_{u \in U_x} (\omega(u))$$

Alors il existe  $T \subseteq A$  tel que  $(S, T)$  soit un arbre de poids minimum et  $u(x) \in T$ .

**Algorithme 21** : Recherche d'arbre couvrant minimum par l'algorithme de Prim

---

**Entrées** : graphe  $G$   
 $T \leftarrow \emptyset$  ;  
**tant que**  $G$  a deux sommets ou plus **faire**  
     choisir  $x \in S$  ;  
     choisir  $u$  de poids minimum adjacent à  $x$  ;  
      $T \leftarrow T \cup \{u\}$  ;  
      $G \leftarrow C_u(G)$  ;  
**fin**

---

**Définition 53** (Contraction d'un graphe). Soit  $G = (S, A)$  un graphe connexe et  $u = (x, y)$  un arc de  $G$ . Le graphe  $C_u(G)$  est le résultat de la contraction de l'arc  $u$  à partir de  $G$  par identification des extrémités de  $x$  et  $y$  de  $u$ . C'est-à-dire que les sommets  $x$  et  $y$  sont remplacés par un unique sommet  $z$ .

*Exemple 24* (Contraction d'un graphe). La figure 4.6 représente la contraction  $C_5(G)$ ,  $G$  étant le graphe représenté figure 4.4.

**Théorème 18.** Soient  $G$  un graphe, et  $u$  un arc de  $G$ .  $G$  est un arbre si et seulement si  $C_u(G)$  est un arbre.

PREUVE :

Le théorème est une conséquence directe du théorème 15 :  $G$  est connexe si et seulement si  $C_u(G)$  l'est et  $C_u(G)$  possède un arc de moins que  $G$ .  $\square$

Selon les théorèmes 17 et 18, l'algorithme de Prim est valide.

*Exemple 25* (Algorithme de Prim). L'algorithme de Prim appliqué au graphe de la figure 4.4 en choisissant  $x = c$  donne le graphe de la figure 4.6. Puis en choisissant successivement les sommets  $x_1 = (c, d)$ ,  $x_2 = (c, d, e)$  et  $x_3 = (a, c, d, e)$ , on obtient les graphes de la figure 4.7. Le graphe résultant de la dernière étape n'est pas représenté. L'arbre couvrant obtenu est celui de la figure 4.5.



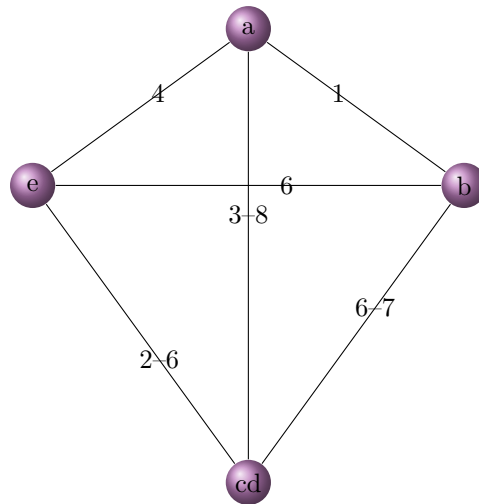


FIGURE 4.6 – Graphe contracté

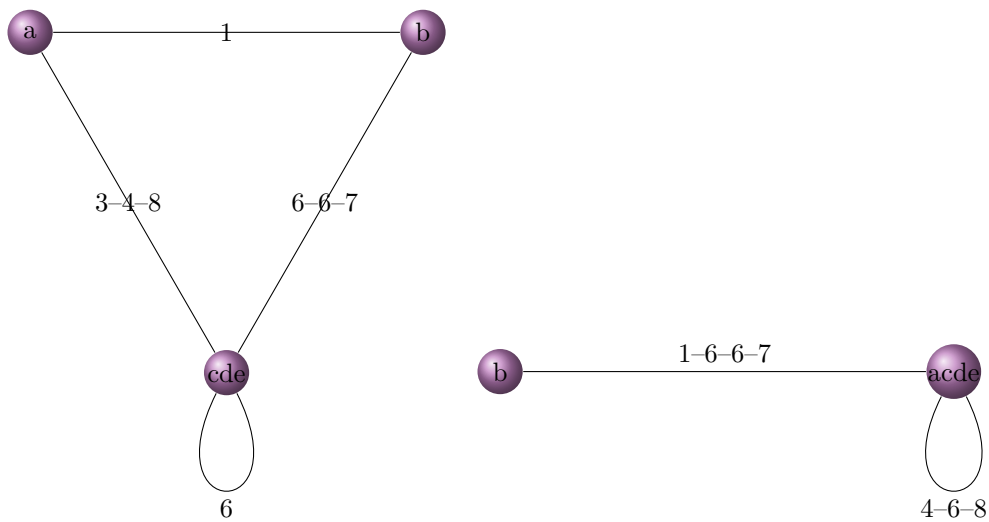


FIGURE 4.7 – Graphe contracté

### Implantation de l'algorithme de Prim

---

**Algorithme 22** : Recherche d'arbre couvrant minimum par l'algorithme de Prim

---

```

Entrées : graphe  $G$ 
pour tous  $x \in S$  faire
     $\sigma(x) \leftarrow \infty$  ;
     $\pi(x) \leftarrow NULL$  ;
fin
choisir une racine  $r \in S$  ;
 $\sigma(r) \leftarrow 0$  ;
 $F \leftarrow S$  ;
tant que  $F \neq \emptyset$  faire
     $x \leftarrow \text{extraireMin}(F)$  ;
    pour tous  $y \in \text{Adj}(x)$  faire
        si  $y \in F$  et  $\omega(x, y) < \sigma(y)$  alors
             $\pi(y) \leftarrow x$  ;
             $\sigma(y) \leftarrow \omega(x, y)$  ;
        fin
    fin
fin

```

---

Pour implanter efficacement l'algorithme de Prim il est important de faciliter la sélection de la nouvelle arête à ajouter au graphe partiel. Dans l'algorithme 22, tous les sommets n'étant pas dans le graphe partiel sont stockés dans une file de priorité  $F$ . Pour chaque sommet  $x$ , la clé de la file  $\sigma(x)$  est le poids minimal d'une arête reliant  $x$  à un sommet du graphe partiel. S'il n'existe pas une telle arête, on pose  $\sigma(x) = \infty$ .

L'arbre sera représenté par un vecteur de prédécesseurs  $\pi$ . L'ajout de sommets à l'arbre se traduit par l'ajout d'éléments dans le vecteur de prédécesseurs, et la contraction du graphe se traduit par la suppression de sommets de la file  $F$ .

### 4.3 Graphes planaires

La notion de graphe planaire est très simple à exprimer, mais beaucoup plus difficile à manipuler. Nous allons voir dans cette partie la définition d'un graphe planaire, quelques caractéristiques de ces graphes, et quelques classes de graphes simples planaires ou non. Il s'agit donc ici uniquement de survoler cette notion et en aucun cas de l'aborder en détail.

**Définition 54** (Graphe planaire). Un graphe est dit planaire s'il existe au moins une façon de le dessiner dans le plan sans que deux arêtes ne se croisent.

**Définition 55** (Graphe biparti). Un graphe est biparti si ses sommets peuvent être divisés en deux ensembles  $S_1$  et  $S_2$ , de sorte que toutes les arêtes du graphe relient un sommet dans  $S_1$  à un sommet dans  $S_2$ .

**Définition 56** (Graphe biparti complet). Un graphe biparti est complet si chaque sommet de  $S_1$  est en relation avec tous les sommets de  $S_2$ . Si  $|S_1| = n$  et  $|S_2| = m$ , on note le graphe  $K_{n,m}$ .

*Exemple 26* (Graphes bipartis). Les arbres sont des graphes bipartis.  $K_{3,3}$  est un graphe biparti.

**Théorème 19** (Caractérisation des graphes bipartis). *Un graphe est biparti si et seulement s'il n'a pas de cycle impair.*

PREUVE :

**Condition nécessaire** : il suffit de raisonner par l'absurde.

On suppose que le graphe possède un cycle impair  $\Gamma = (x_0, \dots, x_n)$  et que la classe de  $x_0$  est 0. Alors, la classe de  $x_1$  est 1, et le cycle étant impair, la classe de  $x_n$  est 0. Donc la classe de  $x_o$  est 1.

**Condition suffisante** : cette condition se prouve par construction, c'est-à-dire en produisant une bipartition adéquate. Le principe est le suivant :

Choisir un sommet  $x_0$  arbitrairement, le marquer à 0, puis marquer ses voisins à 1. Continuer en marquant les voisins des voisins à 0, et ainsi de suite.

Le point crucial de la démonstration est que si deux voisins  $x_i$  et  $x_j$  reçoivent la même marque, alors il existe un cycle impair. En effet, si la marque de  $x_i$  et de  $x_j$  est 0, alors partant de  $x_0$ , il existe un chemin de longueur paire  $\Upsilon_1$  menant à  $x_i$  et un chemin de longueur paire  $\Upsilon_2$  menant à  $x_j$ . Si la marque est de 1, les chemins sont de longueur impaire. Dans les deux cas,  $\{\Upsilon_1, (x_i, x_j), \Upsilon_2\}$  est un cycle, et sa longueur est impaire.  $\square$

*Exemple 27* (Graphes planaires). Les graphes suivant sont planaires :

- les arbres ;
- les cycles ;
- les graphes complets  $K_1, K_2, K_3, K_4$  ;
- les graphes bipartis  $K_{0,n}, K_{1,n}, K_{2,n}$ .

*Exemple 28* (Graphes non planaires). Les graphes suivant ne sont pas planaires :

- $K_5$
- $K_{3,3}$
- le Petersen (voir ex. 21)

**Remarque :**

il est souvent beaucoup plus simple de prouver la planarité que la non planarité d'un graphe. En effet, il suffit dans le premier cas d'exhiber une représentation sans arcs qui se croisent. Dans le deuxième cas, il faut trouver une propriété du graphe interdisant une telle représentation.

**Définition 57** (Face, région). Une face ou région est une région de plan limitée par des arêtes et qui ne contient ni sommets, ni arêtes. Le contour d'une face est le cycle formé par des arêtes frontière de la face. Deux faces sont dites adjacentes si elle possède au moins une frontière commune ; deux faces ne se touchant que par un sommet ne sont pas adjacentes.

**Théorème 20** (Formule d'Euler). Dans un graphe planaire connexe possédant  $n$  sommets,  $m$  arêtes et  $f$  faces on a :  $f = m - n + 2$

PREUVE par récurrence sur  $f$ :

- ▷ Montrons que la propriété est vraie pour  $f = 1$  :
  - dans ce cas,  $G$  ne peut pas posséder de cycle (sinon, il aurait au moins deux faces : une face intérieure et une face extérieure).
  - $G$  est donc connexe et acyclique, et il s'agit donc d'un arbre.
  - Donc,  $m = n - 1$ , et  $m - n + 2 = (n - 1) - n + 2 = 1$ .
- ▷ Supposons que la propriété soit vérifiée pour tous les graphes planaires possédant strictement moins de  $f$  faces.
  - Soit  $G = (S, A)$  un graphe possédant  $f$  faces.
  - Comme  $f > 1$ ,  $G$  possède au moins un cycle  $\Gamma$ .
  - Soit  $u \in \Gamma$  un arc du cycle.
  - Une face ne peut pas être de part et d'autre de  $u$ , et  $u$  sépare donc deux faces.
  - Donc, en supprimant  $u$ , on fusionne deux faces.

- Soit  $G' = (S, A - \{u\})$ .  $G'$  possède  $n$  sommets,  $m - 1$  arcs, et  $f - 1$  faces et respecte l'hypothèse de récurrence.
- Donc,  $f - 1 = m - 1 - n + 2$  et  $f = m - n + 2$ .

□

**Définition 58** (Graphe planaire maximal). Un graphe planaire est dit maximal si l'ajout d'un arc quelconque le rend non planaire. Il est également dit maximal s'il est impossible de lui ajouter un arc (dans le cas d'un graphe complet par exemple).

**Théorème 21** (Faces d'un graphe planaire). *Un graphe planaire possédant  $n \geq 3$  sommets est maximal ssi chaque face de chaque représentation plane est un triangle.*

PREUVE :

- ⇐ Si une face n'est pas un triangle, il est possible de la séparer en deux en ajoutant un nouvel arc tout en conservant la planarité du graphe. Le graphe n'était donc pas maximal.
- ⇒ Si  $G = (S, A)$  est planaire mais pas planaire maximal, il est alors possible d'ajouter un arc  $u$  à  $G$  tel que le graphe  $G' = (S, A \cup \{u\})$  soit toujours planaire. Soit une représentation planaire de  $G'$  dont on supprime  $u$ . Alors les deux faces séparées par  $u$  sont fusionnées en une seule face délimitée par un cycle de longueur supérieure ou égale à 4. Donc, la face en question n'est pas un triangle.

□

**Théorème 22** (Graphe planaire maximal). *Un graphe planaire maximal avec  $n \geq 3$  sommets possède  $m = 3n - 6$  arcs.*

PREUVE :

Construisons un graphe planaire maximal et comptons son nombre d'arcs. Chacune des  $f$  face est délimitée par 3 arcs. De plus, chaque face est comptée 2 fois (une fois pour chaque face dont il fait la frontière). Donc,  $3f = 2m$ . En substituant dans la formule d'Euler, on obtient  $m = 3n - 6$ . □

**Théorème 23** (Graphe planaire). *Un graphe planaire avec  $n \geq 3$  sommets possède  $m \leq 3n - 6$  arcs.*

PREUVE :

On considère un graphe planaire  $G$  possédant  $n \geq 3$  sommets. Si on complète  $G$  de manière à le rendre planaire maximal, il possède  $m = 3n - 6$  arcs. Le graphe  $G$  possède donc un nombre d'arcs  $m \leq 3n - 6$ . □

**Corollaire 3.** *Un graphe  $G$  possédant  $n$  sommets et  $n > 3n - 6$  arcs est non planaire.*

**Remarque :**

la réciproque du théorème 23 est fausse. Il existe des graphes vérifiant  $m \leq 3n - 6$  non planaires, comme par exemple  $K_{3,3}$  ( $n = 6, m = 9$ ).

**Définition 59** (Subdivision). Une subdivision d'un graphe est obtenue en ajoutant des sommets (éventuellement 0) de degré 2 au graphe, chacun séparant un arc en deux parties.

**Théorème 24** (Kuratowski). *Un graphe  $G$  est planaire ssi  $G$  ne contient aucun sous graphe partiel étant une subdivision de  $K_5$  ou de  $K_{3,3}$ .*

## 4.4 Coloration de graphes

Colorer un graphe signifie affecter une couleur à certains de ses éléments. Il existe plusieurs types de colorations de graphes : il est par exemple possible d'affecter une couleur aux sommets ou aux arcs. Il est également possible d'ajouter des contraintes dans le choix des couleurs.

Lorsqu'il est utilisé sans qualificatif, le terme coloration de graphe se rapporte toujours à la coloration des sommets. De même, il s'agit toujours d'une bonne coloration, c'est-à-dire que deux sommets adjacents ne peuvent pas avoir la même couleur. Une coloration utilisant  $k$  couleurs est appelée une  $k$ -coloration et est équivalente au problème de partitionnement des sommets en  $k$  ensembles indépendants.

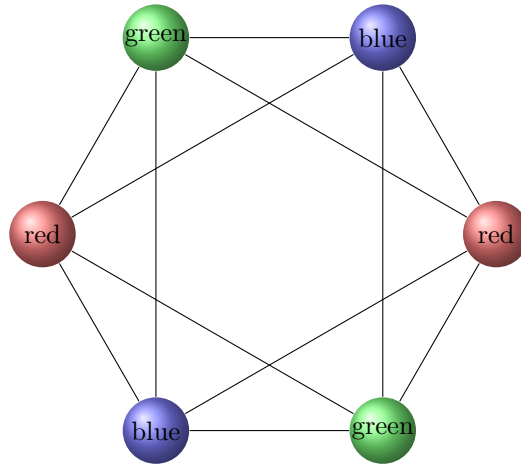


FIGURE 4.8 – Coloration

La figure 4.8 représente un exemple de graphe coloré.

Le problème de coloration d'un graphe a de nombreuses applications en ordonnancement, en allocation des registres de microprocesseurs, en affectation des fréquences radio.

Avant de déterminer une coloration pour un graphe, il est d'usage de déterminer le nombre de couleurs nécessaire pour cette coloration. Ce nombre, noté  $\chi$  est appelé nombre chromatique. Par exemple, le nombre chromatique du graphe complet  $K_n$  est  $\chi(K_n) = n$ . Le nombre de couleurs minimum nécessaire pour une coloration d'arcs est appelé *indice chromatique* et noté  $\Psi$ .

En général, le problème de détermination du nombre ou de l'indice chromatique d'un graphe est NP-difficile. Le problème de décision associé (étant donné un entier  $k$ , existe-t-il une  $k$ -coloration du graphe) est NP-complet. Il existe cependant des algorithmes d'approximation efficaces.

#### 4.4.1 Quelques propriétés des colorations de graphe

**Définition 60** (Clique). Soit  $G = (S, A)$  un graphe. Une clique de  $G$  est un sous-graphe complet de  $G$ . On note  $\omega(G)$  le degré maximum des cliques de  $G$ .

*Exemple 29* (Clique). Le graphe de la figure 4.9 contient deux cliques de degré 3 définies par  $\{d, g, h\}$  et  $\{d, e, h\}$ .

##### Quelques nombres chromatiques :

On note  $\Delta(G)$  le degré maximum de des sommets de  $G$ .

1.  $\chi(G) = 1$  ssi  $G$  est complètement déconnecté.
2.  $\chi(G) \geq 3$  ssi  $G$  possède un cycle impair (cycle de longueur impaire).
3.  $\chi(G) \geq \omega(G)$
4.  $\chi(G) \leq \Delta(G) + 1$
5.  $\chi(G) \leq 4$  pour un graphe planaire.

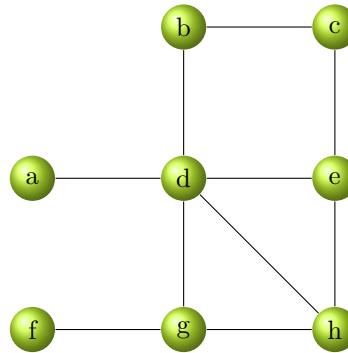


FIGURE 4.9 – Cliques

#### 4.4.2 Exemple d'application

Un exemple d'utilisation de la coloration de graphes est la résolution de problème d'emploi du temps.

Considérons un établissement dans lequel tous les cours ont une durée identique (par exemple 1h15). Certains cours ne peuvent pas avoir lieu en même temps pour diverses raisons :

- ils sont dispensés par le même enseignant
- ils s'adressent au même groupe d'élèves
- ils nécessitent une salle particulière

On souhaite trouver un emploi du temps minimisant le nombre de créneaux horaires utilisés.

##### Formalisation

Le problème se formalise par un graphe d'incompatibilités  $G = (S, A)$ .  $S$  représente l'ensemble des cours à assurer, et deux sommets  $x, y$  de  $S$  forment une arête de  $A$  s'ils sont incompatibles, c'est-à-dire s'ils ne peuvent pas avoir lieu en même temps. Le nombre chromatique de  $G$  représente le nombre de créneaux nécessaire pour assurer tous les cours, et une coloration du graphe détermine la répartition des cours.

Dans le cas où les cours peuvent avoir des durées différentes, il existe des extensions de la notion de coloration permettant de formuler le problème. De même, si le nombre de salles est limité, il existe d'autres méthodes permettant de traiter le problème.

#### 4.4.3 Théorème des quatre couleurs

Le théorème des quatre couleurs est l'un des résultats les plus célèbres de mathématiques combinatoires. Malgré un énoncé simple, ce théorème est resté une conjecture pendant plus d'un siècle, durant lequel il fut abordé par les plus grands mathématiciens, parfois avec des résultats faux. En outre, sa résolution en 1976 comportait un long calcul sur ordinateur, et était donc de ce fait très controversée, la plupart des mathématiciens n'étant pas capables de vérifier l'exactitude des programmes utilisés.

**Théorème 25** (Quatre couleurs). *Toute carte géographique peut être colorée de manière à ce que deux zones adjacentes n'aient pas la même couleur, en utilisant au plus 4 couleurs.*

Il est évident que trois couleurs sont insuffisantes (un contre-exemple est simple à construire), et il est relativement facile de prouver que 5 couleurs sont suffisantes.

Le théorème des quatre couleurs est le premier théorème prouvé en utilisant un ordinateur (grâce à un *theorem prover*), et la preuve n'est pas acceptée par de nombreux mathématiciens car elle n'est pas vérifiable par un être humain.

### Historique

Le problème a été posé en 1852 par Francis Guthrie, tout juste sorti du *University College of London*, qui remarquait que les régions du Royaume Uni pouvaient être colorées avec quatre couleurs de sorte que deux régions avec une frontière commune soient coloriées avec des couleurs différentes. Guthrie a pensé que quatre couleurs suffisaient pour toutes les cartes. Il en a discuté avec son frère qui a communiqué l'idée à son professeur Augustus De Morgan. De Morgan s'est intéressé au problème et a écrit la lettre suivante au mathématicien William Rowan Hamilton :

*A student of mine asked me today to give him a reason for a fact which I did not know was a fact — and do not yet. He says that if a figure be anyhow divided and the compartments differently coloured so that figures with any portion of common boundary line are differently colored — four colours may be wanted but not more — the following (figure 4.10) is a case in which four are needed. Query cannot be a necessity for five or more be invented.*

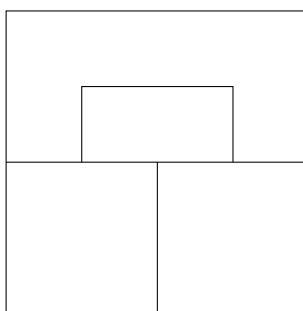


FIGURE 4.10 – Carte dont la coloration nécessite quatre couleurs

Hamilton lui répondit :

*I am not likely to attempt your “quaternion” of colours very soon.*

Bien que Hamilton ne se soit pas montré très enthousiaste, De Morgan continua à étudier le problème, qui est devenu une des plus grandes conjectures de l'époque sous le nom de *problème des quatre couleurs*. Le premier article s'y rapportant a été publié en 1878 par Arthur Cayley.

Par la suite, plusieurs tentatives de preuve ont vu le jour, par exemple en 1879 par Alfred Kempe, puis par Peter Tait en 1880. Ces deux preuves ont été réfutées en 1890 (par Heawood) et en 1891 (par Petersen), soit 11 ans après leurs publications.

En 1890, Heawood a utilisé le principe de la preuve de Kempe pour démontrer le théorème des cinq couleurs.

**Théorème 26** (Théorème de Heawood). *Tout graphe planaire admet une 5-coloration.*

Par la suite, de nombreux résultats relatifs au théorème des quatre couleurs ont été publiés par Danilo Blanuša. Dans les années 1960-1970, Heinrich Heesch a développé des méthodes d'utilisation d'ordinateurs pour la recherche de preuves.

Enfin, en 1977, le théorème des quatre couleurs a été prouvé par Kenneth Appel et Wolfgang Haken. La preuve réduit le problème à 1936 configurations (nombre réduit ultérieurement à 1476), traitées une par une par l'ordinateur. En 1996, une équipe de chercheurs a produit une preuve similaire nécessitant la vérification de 633 cas particuliers. Cependant, ces preuves demandent une entière confiance dans le programme de vérification, et la preuve générée par l'ordinateur n'est pas vérifiable par un être humain.

En 2004, une nouvelle approche s'appuyant sur l'assistant de preuve *Coq* a été utilisée ([Gonthier]). Cet outil ne génère pas une preuve automatiquement, mais il aide à la construire.

**Formalisation**

La carte peut être facilement transformée en graphe : chaque région est un sommet, et si deux régions sont voisines il existe un arc entre elles. En théorie des graphes, le problème des quatre couleurs peut se formuler ainsi : tout graphe planaire est 4-colorable.

**4.4.4 Algorithme de coloration de graphe****Algorithme de Welsh et Powell**

Cet algorithme permet d'obtenir une coloration de graphe valide.

1. Repérer le degré de chaque sommet.
2. Ranger les sommets par ordre de degrés décroissants. (dans certains cas plusieurs possibilités)
3. Attribuer au premier sommet (A) de la liste une couleur.
4. Suivre la liste en attribuant la même couleur au premier sommet (B) qui ne soit pas adjacent à (A).
5. Suivre (si possible) la liste jusqu'au prochain sommet (C) qui ne soit adjacent ni à A ni à B.
6. Continuer jusqu'à ce que la liste soit finie.
7. Prendre une deuxième couleur pour le premier sommet (D) non encore colorié de la liste.
8. Répéter les opérations 4 à 6.
9. Continuer jusqu'à avoir colorié tous les sommets.

Cette méthode n'aboutit pas forcément à une coloration minimale. Il faut donc observer si on peut faire mieux (c'est-à-dire avec moins de couleurs).

**Complexité de l'algorithme dans le pire des cas :**

le pire des cas est celui pour lequel il n'est possible de colorier qu'un sommet par parcours de liste. Lors de l'attribution de la première couleur, il y aura  $|S| - 1$  sommets à visiter, lors de l'attribution de la deuxième  $|S| - 2, \dots$ . La complexité de cet algorithme est donc de  $|S|^2$ .


D'autres algorithmes de coloration de graphes sont présentés dans [Fournier, 2007b].



## Sommaire

<b>5.1 Flots dans un graphe</b>	<b>81</b>
5.1.1 Réseaux à sources et à puits multiples	83
<b>5.2 Flots de valeur maximum</b>	<b>84</b>
5.2.1 Chemins améliorants	84
5.2.2 Coupe dans un réseau	85
5.2.3 Algorithme de Ford-Fulkerson	87
5.2.4 Algorithme d'Edmonds-Karp	88
5.2.5 Algorithmes de préflots	88
<b>5.3 Flots de valeur maximum de coût minimum</b>	<b>89</b>
5.3.1 Algorithme primal	91

## 5.1 Flots dans un graphe

USTAV Kirchhoff fut à l'origine de la théorie des flots grâce à ses travaux sur les réseaux électriques. En particulier, sa loi des nœuds stipule que la somme des intensités des courants entrant dans un nœud est égale à celle qui en sort. Un circuit électrique peut se voir comme un graphe, dans lequel les sommets sont les nœuds du circuit, et les arêtes correspondent aux connexions physiques entre ces nœuds. Pour modéliser les courants traversant le circuit, on considère que chaque arête peut être traversée par un flot.

Considérons un réseau de transport permettant d'acheminer des produits d'un point particulier du réseau, appelé source, vers un autre point, appelé puits. Le flot est la vitesse à laquelle le produit se déplace dans le réseau. Ce réseau peut représenter un réseau routier, un réseau d'eau, un réseau électrique, un réseau de communication, etc. Il sera représenté sous forme de graphe. La notion de flot est abordée dans de nombreux ouvrages, dont [Fournier, 2007a; Sakarovitch, 1984b].

**Définition 61** (Réseau de transport). Un réseau de transport est un graphe orienté  $G = (S, A)$  pour lequel chaque arc  $u = (x, y)$  se voit associé une capacité  $c(u)$  :

$$c : A \rightarrow \mathbb{R}^+ \cup \{+\infty\}$$

On distingue deux sommets du graphe : la source  $s$  et le puits  $t$ . On suppose que tous les sommets se trouvent sur un chemin entre la source et le puits, c'est-à-dire que  $\forall x \in S, s \rightsquigarrow x \rightsquigarrow t$ . On note le réseau  $R = (S, A, c)$

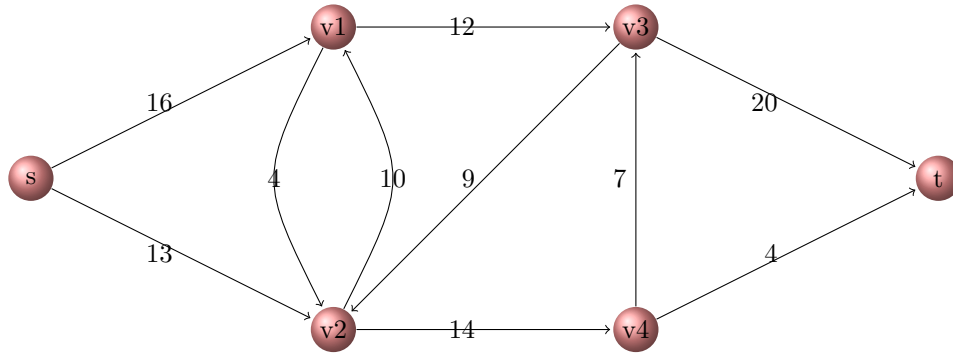


FIGURE 5.1 – Réseau de transport

**Notation :**

soit  $u = (a, b)$  un arc, on note  $-u = (b, a)$  l'arc inverse.

Dans un réseau de transport  $R = (S, A, c)$  il est alors possible de définir formellement la notion de flot :

**Définition 62** (Flot). Un flot de  $R$  est une fonction  $f : S \times S \rightarrow \mathbb{R}$  qui vérifie les trois propriétés suivantes :

1. Capacité :  $\forall u \in A, f(u) \leq c(u)$
2. Symétrie :  $\forall u \in A, f(u) = -f(-u)$
3. Conservation :  $\forall x \in S - \{s, p\}, \sum_{y \in S} f(x, y) = 0$

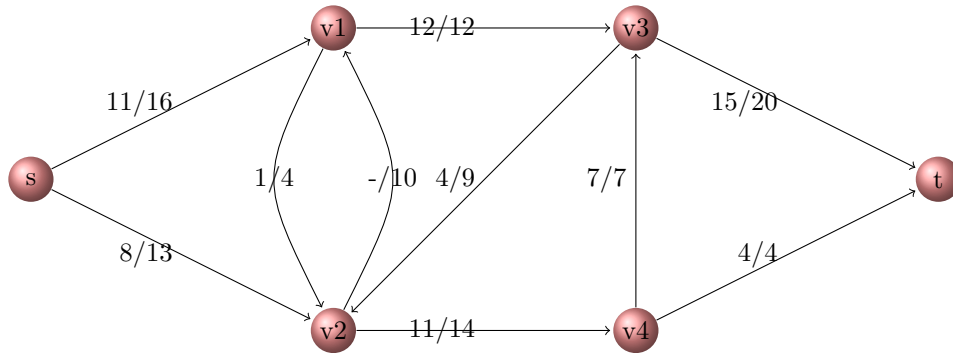


FIGURE 5.2 – Flot dans un réseau

La première propriété signifie que le flux au travers d'un arc ne doit pas dépasser la capacité de cet arc.

La seconde propriété est uniquement une commodité notationnelle.

La dernière propriété signifie que pour chaque sommet, le flot entrant est égal au flot sortant. Le flot vérifie donc la première loi de Kirchhoff.

Un exemple de réseau est donné figure 5.1, et un exemple de flot dans ce réseau figure 5.2.

**Définition 63** (Valeur d'un flot). Soit  $f$  un flot sur un réseau. On appelle valeur de  $f$ , et on note  $|f| = \sum_{x \in S} f(s, x)$ , c'est-à-dire le flot partant de la source.

**Définition 64** (Capacité résiduelle). Soit  $R = (S, A, c)$  un réseau. On note  $\delta(u)$  la capacité résiduelle d'un arc  $u$ , définie par :  $\forall u \in A, \delta(u) = c(u) - f(u)$ .

**Définition 65** (Réseau résiduel). Soit  $R = (S, A, c)$  un réseau. Le réseau résiduel de  $R$  est  $R_f = (S, A_f, \delta)$  avec  $A_f = \{u \in A / \delta(u) > 0\}$

**Définition 66** (Vecteur représentatif). Soit un graphe  $G = (S, A)$  et  $\Gamma$  un cycle muni d'un sens de parcours de  $G$ . On définit le vecteur représentatif de  $\Gamma$  :  $\gamma \in \mathbb{R}^n$  défini par :

$$\gamma_i = \begin{cases} 1 & \text{si } u_i \in \Gamma^+ \\ -1 & \text{si } u_i \in \Gamma^- \\ 0 & \text{si } u_i \notin \Gamma \end{cases}$$

**Définition 67** (Décomposition conforme). Soit  $f$  un flot sur un graphe  $G = (S, A)$ . On effectue une décomposition conforme de  $f$  si on trouve  $q$  cycles  $\Gamma_1, \Gamma_2, \dots, \Gamma_q$  dont les vecteurs représentatifs  $\gamma_1, \dots, \gamma_q$  vérifient :

$$f = \sum_{j=1}^q \delta_j \gamma_j \text{ avec } \forall j \in \{1, \dots, q\}, \delta_j > 0$$

$$f(u) > 0 \text{ (resp. } < 0) \text{ et } u \in \Gamma_j \Rightarrow u \in \Gamma_j^+ \text{ (resp. } u \in \Gamma_j^-)$$

**Théorème 27** (Décomposition conforme). *Tout flot admet une décomposition conforme.*

### 5.1.1 Réseaux à sources et à puits multiples

La notion de flot se généralise facilement à un réseau à sources et puits multiples. Considérons le réseau de la figure 5.3. Ce réseau possède deux sources (a et b) et deux puits (c et d). Il est possible de se ramener au cas précédent en ajoutant un sommet  $s$  dit *supersource* en amont des sources et un sommet  $t$  dit *superpuits* en aval des puits. Les capacités des arcs reliant la supersource et le superpuits aux autres sommets sont choisies infinies. Le réseau de la figure 5.3 se transforme alors en celui de la figure 5.4.

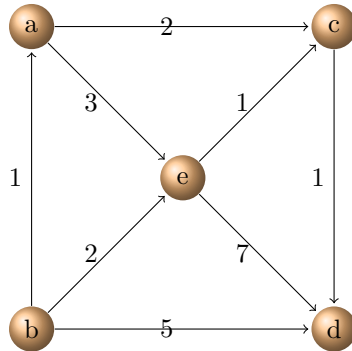


FIGURE 5.3 – Réseau à sources (a,b) et puits (c,d) multiples

**Lemme 3.** Soit  $R = (S, A, c)$  un réseau, et  $f$  un flot sur le réseau.

$$\forall X \subseteq S, f(X, X) = 0 \quad (5.1)$$

$$\forall X, Y \subseteq S, f(X, Y) = -f(Y, X) \quad (5.2)$$

$$\forall X, Y, Z \subseteq S \text{ avec } X \cap Y = \emptyset, f(X \cup Y, Z) = f(X, Z) + f(Y, Z) \quad (5.3)$$

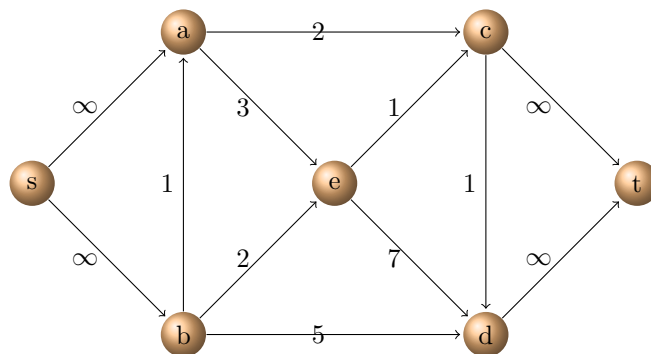


FIGURE 5.4 – Supersource et superpuits

## 5.2 Flots de valeur maximum

Un des problèmes intéressants à traiter dans les réseaux de transports est celui du flot maximal : quel est le flot  $f$  compatible (vérifiant les propriétés du flot) tel que  $|f|$  soit maximal ?

Il existe de nombreux algorithmes permettant de répondre à cette question, dont celui de **Ford-Fulkerson** auquel nous allons nous intéresser. Cet algorithme, extrêmement simple, est décrit dans la figure 23.

---

### Algorithme 23 : Méthode de Ford-Fulkerson

---

**Entrées :** graphe  $g$ , noeud *source*, noeud *puits*

**Données :** flot  $f$

initialiser  $f$  à 0 ;

**tant que**  $\exists p$  chemin améliorant **faire**

    augmenter  $f$  le long de  $p$  ;

**fin**

---

Toute la difficulté de l'algorithme consiste à déterminer les chemins améliorants.

### 5.2.1 Chemins améliorants

**Définition 68** (Chemin améliorant). Soit  $R = (S, A, c)$  un réseau de source  $s$  et de puits  $t$ . Un chemin  $p$  améliorant est un chemin de  $s$  à  $t$  le long duquel aucun arc dans le sens direct n'est saturé, et aucun arc dans le sens inverse n'est nul.

Soit un réseau  $R$  et un chemin  $p$  dans  $R$ . On note  $p^+$  (resp.  $p^-$ ) les arcs de  $p$  orientés de  $s$  vers  $t$  (resp. de  $t$  vers  $s$ ). On note alors :

$$\begin{aligned}\delta_p^+ &= \min_{u \in p^+} (\delta(u)) \\ \delta_p^- &= \min_{u \in p^-} (f(u))\end{aligned}$$

On considère alors  $\delta = \min(\delta_p^+, \delta_p^-)$ . Il est possible d'améliorer le flot en ajoutant  $\delta$  au flot des arcs de  $p^+$  et en soustrayant  $\delta$  au flot des arcs de  $p^-$ . Le résultat obtenu est toujours un flot réalisable grâce au choix de  $\delta$ .

*Exemple 30* (Amélioration de flot). Considérons l'exemple de la figure 5.5 qui représente un chemin dans un réseau. Soit  $p$  le chemin  $(a, b, c, d, e)$ .  $p^+ = \{(a, b), (b, c), (d, e)\}$  et  $p^- = \{(c, d)\}$ . On a donc  $\delta_p^+ = \min(3, 2, 5) = 2$  et  $\delta_p^- = 1$ .

$\delta$  vaut alors 1, et il est possible d'augmenter le flot sur ce chemin en ajoutant 1 à  $(a, b)(b, c)(d, e)$  et en retranchant 1 à  $(c, d)$ . On obtient alors le flot représenté figure 5.7. Le graphe résiduel est représenté figure 5.6.

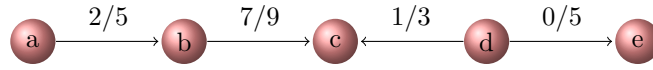


FIGURE 5.5 – Amélioration de flot

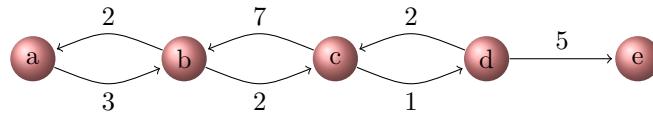


FIGURE 5.6 – Réseau résiduel

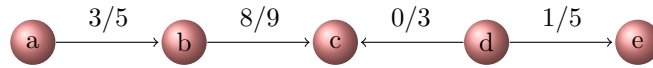


FIGURE 5.7 – Flot amélioré

### 5.2.2 Coupe dans un réseau

**Définition 69** (Coupe). Soit un réseau  $R = (S, A, c)$  de source  $s$  et de puits  $t$ . Un ensemble  $C$  est appelé coupe séparant  $t$  de  $s$  si on peut trouver  $X \subset S$  avec  $s \in X, t \notin X$  tel que :

$$C = \Omega^+(X) = \{u \in A / I(u) \in X, T(u) \notin X\}$$

Une coupe  $C$  est un ensemble d'arcs divisant le réseau en deux parties : une partie contenant  $s$  et une partie contenant  $t$ .

**Définition 70** (Capacité d'une coupe). On appelle capacité d'une coupe  $C$  séparant  $t$  de  $s$  la somme des capacités des arcs de  $C$ .

$$c(C) = \sum_{u \in C} c(u)$$

*Exemple 31* (Coupe). La figure 5.8 représente un exemple de coupe dans un réseau. L'ensemble  $X$  choisi est  $\{s, a, b\}$ . La coupe est  $C = \{(a, c); (b, d)\}$ . La valeur du flot au travers de la coupe est  $\delta(C) = 12 - 4 + 11 = 19$ , et sa capacité est de  $c(C) = 12 + 14 = 26$

**Lemme 4.** Pour tout réseau  $R = (S, A, c)$  de source  $s$  et de puits  $t$ , pour tout flot  $f$  réalisable sur  $R$ , pour toute coupe  $C = \Omega^+(E)$  séparant  $t$  de  $s$ , et pour  $T = S - E$ , on a :

$$\delta(C) = f(E, T) = |f| \quad (5.4)$$

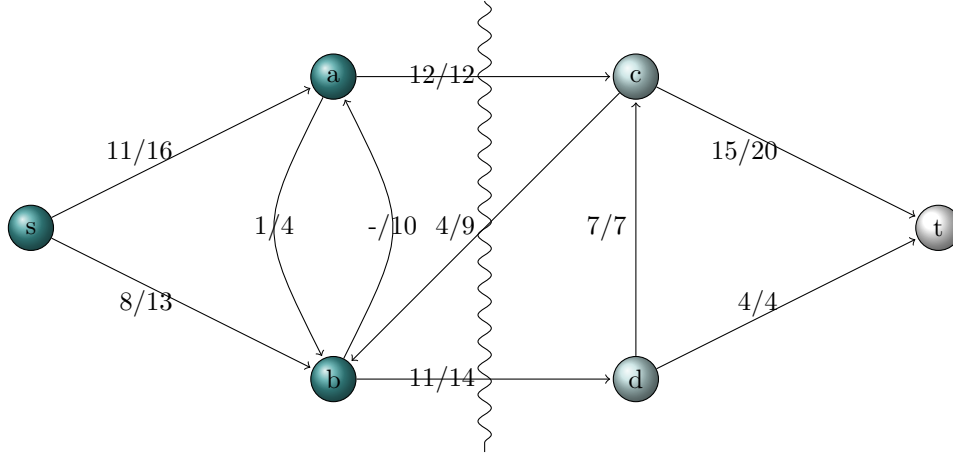


FIGURE 5.8 – Coupe

PREUVE :

$$\begin{aligned}
 \delta(C) &= f(E, T) \\
 &= f(E, S) - f(E, E) \\
 &= f(E, S) \\
 &= f(s, S) + f(E - s, S) \\
 &= f(s, S) \\
 &= |f|
 \end{aligned}$$

Par définition  
selon l'équation 5.3  
selon l'équation 5.1  
selon l'équation 5.3  
car  $f(E-s, S)=0$  selon la conservation du flot

□

**Corollaire 4.** La valeur d'un flot  $f$  dans un réseau  $R$  est majorée par la capacité d'une coupe quelconque de  $R$ .

PREUVE :

$$|f| = f(E, T) = \sum_{a \in E} \sum_{b \in T} f(a, b) \leq \sum_{a \in E} \sum_{b \in T} c(a, b) \leq c(E, T)$$

□

**Théorème 28** (flot maximum et coupe minimum). Soit  $R = (S, A, c)$  un réseau de sommet  $s$  et de puits  $t$ , et  $f$  un flot dans ce réseau. Alors les conditions suivantes sont équivalentes :

1.  $f$  est un flot maximal dans  $R$
2. Le réseau résiduel  $R_f$  ne contient aucun chemin améliorant
3.  $|f| = c(C)$  pour une certaine coupe  $C$  de  $R$ .

PREUVE :

- $1 \Rightarrow 2$  : raisonnons par l'absurde. Soit  $f$  un flot maximum de  $R$  et  $p$  un chemin améliorant de  $R_f$ . Alors  $f + f_p$  est un flot de  $R$  de valeur strictement supérieure à  $|f|$ .
- $2 \Rightarrow 3$  :  $R_f$  ne possède aucun chemin améliorant, c'est-à-dire aucun chemin de  $s$  vers  $t$ . On définit  $E = \{x \in S/s \xrightarrow{R_f} x\}$  et  $T = S - E$ .  $\Omega^+(E)$  est une coupe.  $\forall (a, b) \in E \times T$ ,  $f(a, b) = c(a, b)$  (sinon  $(a, b) \in R_f$ ). D'après le lemme 4 on a donc  $|f| = f(S, T) = c(C)$ .

— 3  $\Rightarrow$  1 : Selon le corollaire 4,  $|f| \leq \delta(\mathcal{C})$  pour toutes les coupes  $\mathcal{C}$ .  $|f| = c(\mathcal{C})$  implique donc que  $f$  est maximum.

□

### 5.2.3 Algorithme de Ford-Fulkerson

En utilisant tous les résultats vus précédemment, et en particulier en utilisant la notion de réseau résiduel, il est possible de réécrire la méthode de Ford-Fulkerson pour obtenir l'algorithme défini dans 24.

---

#### Algorithme 24 : Algorithme de Ford-Fulkerson

---

**Entrées :** réseau  $r$ , nœud *source*, nœud *puits*

**Données :** flot  $f$

**pour tous**  $(x, y) \in A$  **faire**

$f(x, y) \leftarrow 0$  ;

$f(y, x) \leftarrow 0$  ;

**fin**

**tant que**  $\exists p$  dans  $R_f$  **faire**

$\delta(p) \leftarrow \min_{u \in p} (\delta(u))$  ;

**pour tous**  $(x, y) \in p$  **faire**

$f(x, y) \leftarrow f(x, y) + \delta(p)$  ;

$f(y, x) \leftarrow -f(x, y)$  ;

**fin**

**fin**

---

Selon le théorème 28, lorsqu'il n'existe plus de chemin améliorant, le flot  $f$  est maximal.

#### Analyse de l'algorithme :

La terminaison de l'algorithme de *Ford-Fulkerson* n'est pas garantie, sauf dans le cas où les capacités des arcs sont dans  $\mathbb{Q}$ . Mais même dans ce cas, il n'y a pas de garantie sur le temps d'exécution.

Dans le cas où les capacités sont des rationnels, il est toujours possible de se ramener à des entiers naturels par une simple mise à l'échelle. Notons alors  $f^*$  le flot maximal trouvé par l'algorithme. Le temps d'exécution de *Ford-Fulkerson* dans le pire des cas est alors de  $\Theta(|A| \cdot |f^*|)$ .

La figure 5.9 représente un cas pour lequel l'algorithme 24 n'est pas efficace. Dans ce cas, le flot maximal vaut 2000, et il est possible de ne l'améliorer que de 1 à chaque itération, et donc de le trouver en 2000 itérations.

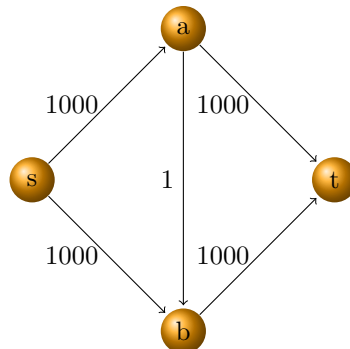


FIGURE 5.9 – Graphe piège pour l'algorithme de Ford-Fulkerson

### 5.2.4 Algorithme d'Edmonds-Karp

Il est possible d'améliorer l'algorithme de Ford-Fulkerson en recherchant les chemins améliorants par un parcours en largeur. Le chemin améliorant est alors choisi en tant que plus court chemin de  $s$  à  $t$  dans le réseau résiduel.

Cette implantation de la méthode est appelée algorithme d'**Edmonds-Karp**. Cet algorithme s'exécute alors en  $\Theta(|S| \cdot |A|^2)$ .

### 5.2.5 Algorithmes de préflots

La méthode des préflots permet de résoudre efficacement les problèmes de flot maximum. Elle sert de base aux algorithmes les plus rapides de flot maximum connus.

À la différence de la méthode de *Ford-Fulkerson*, cette méthode travaille de façon localisée. Au lieu de rechercher un chemin améliorant dans le réseau résiduel tout entier, on se concentre sur un sommet et ses voisins.

**Définition 71** (Préflot). On appelle préflot une fonction  $f : S \times S \rightarrow \mathbb{R}$  qui vérifie les propriétés de capacité et de symétrie. De plus, elle doit vérifier une version relâchée de la conservation :  $\forall x \in S - \{s\}, f(S, x) = e(x) \geq 0$ . On appelle cette quantité excédent de flot. On dit qu'un sommet  $x \in S - \{s\}$  déborde si  $e(x) > 0$ .

L'algorithme de préflot s'appuie sur des opérations élémentaires appelées *pousser* et *réétiqueter*. Pour fonctionner, ces opérations nécessitent la notion de *hauteur* d'un sommet.

**Définition 72** (Fonction de hauteur). Soit  $R = (S, A, c)$  un réseau de source  $s$  et de puits  $t$ , et soit  $f$  un préflot de  $R$ . On note  $R_f = (S, A_f, \delta)$  le réseau résiduel. On appelle fonction de hauteur une fonction  $h : S \rightarrow \mathbb{N}$  définie par :

$$\begin{cases} h(s) = |S| \\ h(t) = 0 \\ \forall (x, y) \in A_f, h(x) \leq h(y) + 1 \end{cases}$$

L'opération élémentaire *pousser*( $x, y$ ) peut s'appliquer si  $x$  est un sommet de débordement,  $\delta(x, y) > 0$  et si  $h(x) = h(y) + 1$ . La procédure 25 met à jour le préflot.

---

**Algorithme 25 :** Procédure pousser : pousser  $\delta(x, y)$  unités de flot de  $x$  vers  $y$

---

```
// Prérequis:  $e(x) > 0, \delta(x, y) > 0, h(x) = h(y) + 1$ 
 $d_f(x, y) \leftarrow \min(e(x), \delta(x, y))$ ;
 $f(x, y) \leftarrow f(x, y) + d_f(x, y)$ ;
 $f(y, x) \leftarrow -f(x, y)$ ;
 $e(x) \leftarrow e(x) - d_f(x, y)$ ;
 $e(y) \leftarrow e(y) + d_f(x, y)$ ;
```

---

L'opération *réétiqueter*( $x$ ) s'applique si  $x$  déborde et si  $h(x) \leq h(y)$  pour tout arc  $(x, y) \in A_f$ .

---

**Algorithme 26 :** Procédure ré-étiqueter : accroît la hauteur de  $x$

---

```
// Prérequis:  $e(x) > 0, \forall y \in S / (x, y) \in A_f, h(x) \leq h(y)$ 
 $h(x) \leftarrow 1 + \min_{(x, y) \in A_f} \{h(y)\}$ ;
```

---

**Remarque :**

tant qu'il existe un sommet débordant, il est possible d'appliquer un poussage ou un réétiquetage.

**Théorème 29** (Validité de l'algorithme de préflot). *L'algorithme de préflot défini en 28 se termine, et le préflot calculé est un flot maximal du réseau.*



---

**Algorithme 27** : Initialisation du préflot dans un réseau  $R = (S, A, c)$ 


---

```

pour tous  $x \in S$  faire
     $h(x) \leftarrow 0$  ;
     $e(x) \leftarrow 0$  ;
fin
pour tous  $(x, y) \in A$  faire
     $f(x, y) \leftarrow 0$  ;
     $f(y, x) \leftarrow 0$  ;
fin
 $h(S) \leftarrow |S|$  ;
pour tous  $x \in \text{succ}(s)$  faire
     $f(s, x) \leftarrow c(s, x)$  ;
     $f(x, s) \leftarrow -f(s, x)$  ;
     $e(x) \leftarrow c(s, x)$  ;
     $e(s) \leftarrow e(s) - c(s, x)$  ;
fin

```

---



---

**Algorithme 28** : Procédure préflot dans un réseau  $R = (S, A, c)$ 


---

```

InitPreflot(R, s) ;
tant que poussage ou réétiquetage possible faire
    choisir un poussage ou un réétiquetage et l'appliquer ;
fin

```

---

**Complexité de l'algorithme :**

le nombre d'opérations élémentaires nécessaire à l'exécution de l'algorithme de préflot défini en 28 sur un réseau  $R = (S, A, c)$  est de  $\Theta(|S|^2 \cdot |A|)$ .

### 5.3 Flots de valeur maximum de coût minimum

Soit un réseau  $R = (S, A, a, b, c)$  avec :

$$\begin{cases} a : A \rightarrow \mathbb{R} \\ b : A \rightarrow \mathbb{R} \cup \{-\infty\} \\ c : A \rightarrow \mathbb{R} \cup \{+\infty\} \end{cases}$$

et  $\forall u \in A, b(u) \leq c(u)$ . Le problème du flot de coût minimal sur  $R$  consiste à chercher un vecteur  $f \in \mathbb{R}^n$  tel que :

1.  $f$  soit un flot sur  $R$
2.  $\forall u \in A, b(u) \leq f(u) \leq c(u)$
3.  $\sum_{u \in A} a(u)f(u)$  soit minimal

De nombreux problèmes peuvent s'écrire en tant que problèmes de flots de coût minimum.

*Exemple 32* (Plus court chemin). Le problème de recherche du plus court chemin de  $s$  à  $t$  dans un réseau  $R = (S, A, d)$  peut s'écrire comme une recherche de flot de coût min de  $s$  à  $t$  dans le réseau  $R' = (S, A, a, b, c)$  avec :

$$\forall u \in A \begin{cases} a(u) = d(u) \\ b(u) = 0 \\ c(u) = 1 \end{cases}$$

**Théorème 30.** Une condition nécessaire et suffisante pour qu'un flot  $f$  réalisable sur un réseau  $R = (S, A, a, b, c)$  soit une solution optimale du problème du flot de coût minimum est que pour tout cycle  $\Gamma$  tel que

$$\delta = \min\left\{\min_{u \in \Gamma^+} (c(u) - f(u)), \min_{u \in \Gamma^-} (f(u) - b(u))\right\} > 0 \quad (5.5)$$

on ait

$$\sum_{u \in \Gamma^+} a(u) - \sum_{u \in \Gamma^-} a(u) \geq 0 \quad (5.6)$$

PREUVE :

**Sens direct :**

Soit  $\Gamma$  un cycle qui satisfait l'équation 5.5 et pas l'équation 5.6.

Soit  $f'$  un flot tel que

$$f'(u) = \begin{cases} f(u) + \delta & \text{si } u \in \Gamma^+ \\ f(u) - \delta & \text{si } u \in \Gamma^- \\ f(u) & \text{si } u \notin \Gamma \end{cases}$$

Alors  $f'$  est un flot réalisable et  $af' < af$

**Réciproque :**

Si  $f$  n'est pas optimal, soit  $f'$  un flot réalisable tel que  $af' - af < 0$ .

Alors,  $f'' = f' - f$  est un flot réalisable. D'après le théorème de décomposition confirme 27, on peut trouver des cycles  $\Gamma_i$  de  $f''$  de vecteurs représentatifs  $\gamma_i$  tels que :

$$f'' = \sum_{j=1}^q \delta_j \gamma_j \text{ avec } \forall j \in 1..q \delta_j > 0$$

$$u \in \Gamma_j, f''(u) > 0 \rightarrow u \in \Gamma_j^+$$

$$u \in \Gamma_j, f''(u) < 0 \rightarrow u \in \Gamma_j^-$$

Et comme  $af'' = af' - af < 0$  il existe au moins un  $j$  tel que

$$a\gamma_j = \sum_{u \in \Gamma_j^+} a(u) - \sum_{u \in \Gamma_j^-} a(u) < 0$$

□

**Corollaire 5.** Au flot  $f$  réalisable sur  $R = (S, A, a, b, c)$  on associe le réseau  $\hat{R}(f) = (S, \hat{A}, \hat{d})$  avec  $\hat{A} = \hat{A}' \cup \hat{A}''$  et :

si  $u = (x, y) \in A$  et  $f(u) < c(u)$  alors  $u' = (x, y) \in \hat{A}'$  et  $\hat{d}(u') = a(u)$

si  $u = (x, y) \in A$  et  $f(u) > b(u)$  alors  $u'' = (x, y) \in \hat{A}''$  et  $\hat{d}(u'') = -a(u)$

Une condition nécessaire et suffisante pour que  $f$  soit une solution optimale du problème du flot de coût minimal est que le réseau associé  $\hat{R}(f)$  ne contienne pas de cycle absorbant.

**Corollaire 6.** Une condition nécessaire et suffisante pour qu'un flot  $f$  réalisable sur un réseau  $R = (S, A, a, b, c)$  soit une solution optimale du problème du flot de coût minimal est qu'il existe  $A' \subset A$  et un système de potentiels  $\pi$  tels que :

- $(S, A')$  soit connexe
- $\pi$  satisfait :

$$\begin{aligned} u = (x, y) \in A' & \Rightarrow \pi(y) - \pi(x) = a(u) \\ u = (x, y) \text{ et } \pi(y) - \pi(x) < a(u) & \Rightarrow f(u) = b(u) \\ u = (x, y) \text{ et } \pi(y) - \pi(x) > a(u) & \Rightarrow f(u) = c(u) \end{aligned}$$

### 5.3.1 Algorithme primal

Le principe de l'algorithme primal de résolution du problème de flot de coût minimal est une conséquence directe du corollaire 6 du théorème 30. Étant donné un flot réalisable  $f$  et un ensemble d'arcs  $\tilde{A}$  tel que  $(S, \tilde{A})$  soit un arbre, on calcule  $\pi$  par le corollaire 6.

Si le flot n'est pas une solution optimale, il existe un arc  $\tilde{u} = (x, y)$  avec :

$$\begin{aligned} \pi(y) - \pi(x) &> a(\tilde{u}) \quad \text{et} \quad f(\tilde{u}) < c(\tilde{u}) \\ (\text{resp. } \pi(y) - \pi(x) &< a(\tilde{u}) \quad \text{et} \quad f(\tilde{u}) > b(\tilde{u})) \end{aligned}$$

Soit  $\Gamma$  le cycle unique de  $(S, \tilde{U} \cup \{\tilde{u}\})$ , soit  $\Gamma^+$  l'ensemble des arcs orientés comme  $\tilde{u}$  dans  $\Gamma$ , et  $\Gamma^- = \Gamma - \Gamma^+$ . On pose

$$\begin{aligned} \delta^+ &= \min_{u \in \Gamma^+} (c(u) - f(u)) \\ \delta^- &= \min_{u \in \Gamma^-} (f(u) - b(u)) \end{aligned}$$

- Si  $\delta^+ = 0$  (resp. si  $\delta^- = 0$ ) soit  $u'$  un arc de  $\Gamma^+$  (resp.  $\Gamma^-$ ) tel que  $f(u') = c(u)$  (resp.  $f(u') = b(u)$ ). L'ensemble des arcs

$$\tilde{U}' = \tilde{U} \cup \{\tilde{u}\} - \{u'\}$$

définit un graphe partiel qui est un arbre. On calcule  $\pi$  grâce au corollaire 6 et on recommence.

- Si  $\delta = \min(\delta^+, \delta^-) > 0$ , on définit  $f'$  :

$$f'(u) = \begin{cases} f(u) + \delta & \text{si } u \in \Gamma^+ \\ f(u) - \delta & \text{si } u \in \Gamma^- \\ f(u) & \text{si } u \notin \Gamma \end{cases}$$

Il existe alors un arc  $u'$  tel que l'une des conditions suivantes soit réalisée :

$$\begin{cases} u' \in \Gamma^+ \text{ et } f'(u') = c(u') \\ u' \in \Gamma^- \text{ et } f'(u') = b(u') \end{cases}$$

L'ensemble des arcs

$$\tilde{U}' = \tilde{U} \cup \{\tilde{u}\} - \{u'\}$$

définit un graphe partiel qui est un arbre. On calcule  $\pi$  grâce au corollaire 6 et on recommence.

*Exemple 33* (Algorithme primal). Considérons le réseau de la figure 5.10, chaque arc contenant  $a, b, c$ . Supposons que l'on ait déterminé le flot  $f$  représenté par la figure 5.11, et  $U' = \{(s, b), (b, t), (a, t)\}$ .

Posant  $\pi(s) = 0$ , et en utilisant le corollaire 6, on obtient :  $\pi(b) = 4, \pi(t) = 7, \pi(a) = 1$ , et pour l'arc  $(a, b)$  :  $\pi(b) - \pi(a) = 3 > a(ab) = 1$ .

Donc,  $\Gamma^+ = \{(a, b), (b, t)\}, \Gamma^- = \{(a, t)\}, \delta = 1$ . On obtient alors la solution représentée figure 5.12.

L'arbre  $(S, A')$  et les valeurs de  $\pi$  sont représentées par la figure 5.13.

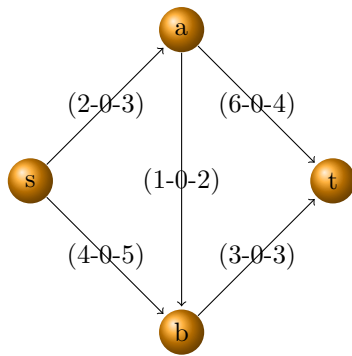


FIGURE 5.10 – Algorithme primal 1

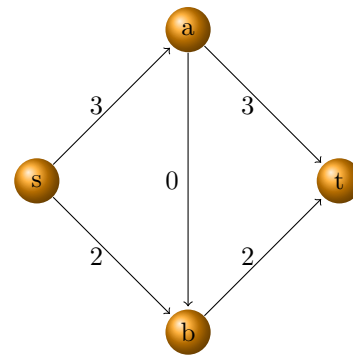


FIGURE 5.11 – Algorithme primal 2

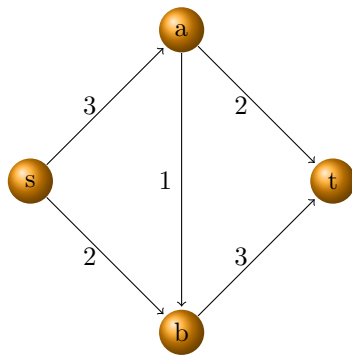


FIGURE 5.12 – Algorithme primal 3

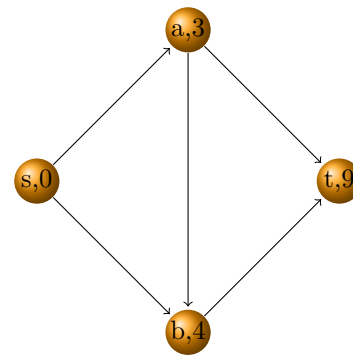



FIGURE 5.13 – Algorithme primal 4

## Problèmes NP-Complets

### Sommaire

<b>6.1</b>	<b>Problème du voyageur de commerce</b>	<b>93</b>
6.1.1	Algorithmes d'approximation	94
6.1.2	Algorithme de Christofides	95
<b>6.2</b>	<b>Problème du cycle Hamiltonien</b>	<b>96</b>
<b>6.3</b>	<b>Variations autour de l'arbre couvrant minimum</b>	<b>97</b>
<b>6.4</b>	<b>Problème de la clique maximum</b>	<b>97</b>
<b>6.5</b>	<b>Isomorphisme de graphes</b>	<b>97</b>

 E chapitre recense quelques problèmes NP-complets classiques issus de la théorie des graphes. Pour tous ces problèmes, on ne cherchera pas à obtenir une solution optimale mais on se contentera d'une solution approchée.

### 6.1 Problème du voyageur de commerce

L'énoncé du problème du voyageur de commerce est le suivant : étant donné  $n$  points (des « villes ») et les distances séparant chaque point, trouver un chemin de longueur totale minimale qui passe exactement une fois par chaque point (et revienne au point de départ).

Ce problème est plus compliqué qu'il n'y paraît ; on ne connaît pas de méthode de résolution permettant d'obtenir des solutions exactes en un temps raisonnable pour de grandes instances (grand nombre de villes) du problème. Pour ces grandes instances, on devra donc souvent se contenter de solutions approchées, car on se retrouve face à une explosion combinatoire : le nombre de chemins possibles passant par 69 villes est déjà un nombre de 100 chiffres. Pour comparaison, un nombre de 80 chiffres permettrait déjà de représenter le nombre d'atomes dans tout l'univers connu !).

Ce problème peut servir tel quel à l'optimisation de trajectoires de machines-outils par exemple, pour minimiser le temps total que met une fraiseuse à commande numérique pour percer  $n$  points dans une plaque de tôle.

Plus généralement, divers problèmes de recherche opérationnelle se ramènent au voyageur de commerce. Un voyageur de commerce peu scrupuleux serait intéressé par le double problème du chemin le plus court (pour son trajet réel) et du chemin le plus long (pour sa note de frais).

### 6.1.1 Algorithmes d'approximation

Comme pour la plupart des problèmes NP-difficiles, des algorithmes d'approximation de la solution optimale ont été recherchés. Ces algorithmes sont définis dans de nombreux ouvrages comme [Fournier, 2007b; Cormen *et al.*, 1994]. Nous étudierons ici deux de ces algorithmes.

**Définition 73** (Graphe Euclidien). Soit  $G = (S, A, \omega)$  un graphe non orienté pondéré.  $G$  est dit Euclidien s'il respecte l'inégalité triangulaire, autrement dit si :

$$\forall i, j, k : \omega_{i,k} \leq \omega_{i,j} + \omega_{j,k}$$

Lorsque le graphe est Euclidien, il est possible de calculer en temps polynomial une tournée de longueur inférieure à deux fois la longueur de la tournée optimale.

Par exemple, l'algorithme 29 est de complexité polynomiale (l'opération la plus coûteuse est la construction d'un arbre couvrant minimal), et il permet de déterminer un cycle de longueur inférieure au double de la longueur du cycle optimal.

---

**Algorithme 29 : Approximation de tournée minimale**


---

**Entrées :** Graphe  $K_n = G$  Euclidien

Calculer l'arbre couvrant minimum  $T$  de  $G$ ;

Choisir un chemin  $\Gamma$  dans  $T$  (pouvant passer plusieurs fois par le même sommet); chaque arc de l'arbre doit être traversé exactement deux fois ;

Réduire  $\Gamma$  en ne prenant chaque sommet que lors de sa première apparition;

---

La dernière étape a un sens uniquement si le graphe est Euclidien ; en effet, on remplace le chemin  $ABC$  par  $AC$  en sachant que  $\omega_{A,B} + \omega_{B,C} \leq \omega_{A,C}$ .

*Exemple 34* (Approximation de la tournée optimale). On considère le graphe  $K_5$  représenté figure 6.1, sur lequel on cherche une tournée optimale. On commence par calculer l'arbre couvrant minimal, représenté figure 6.2.

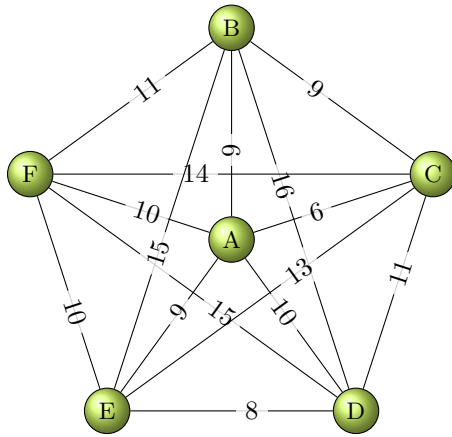


FIGURE 6.1 – Graphe initial

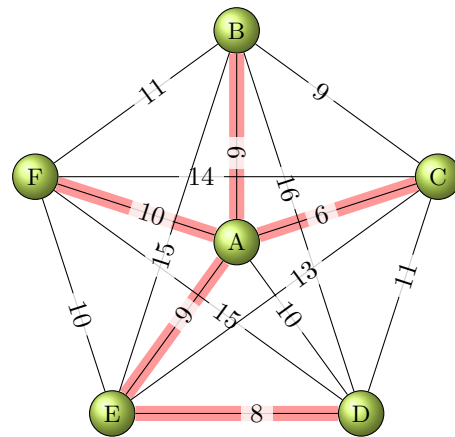


FIGURE 6.2 – Arbre couvrant

Un parcours en profondeur de l'arbre partant de A sera : ABACAEDEAF.

Le parcours donné par l'algorithme consiste à prendre les sommets de la liste précédente lors de leur première apparition. Le chemin, représenté figure 6.3, sera alors : ABCEDFA. Sa longueur est de 64.

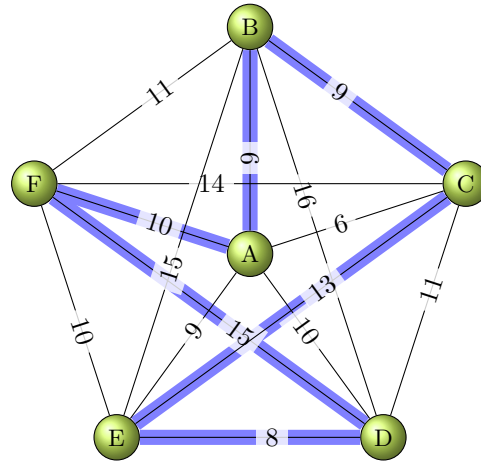


FIGURE 6.3 – Tournée calculée

### 6.1.2 Algorithme de Christofides

Une amélioration de l'algorithme présenté section 6.1.1 a été proposée par Nicos Christofides dans [Christofides, 1976]. Cet algorithme garantit l'obtention, en temps polynomial, d'une tournée de longueur inférieure à 1.5 fois la longueur optimale. Il utilise les résultats préliminaires définis par les propriétés 2.

*Propriétés 2.* L'algorithme de Christofides s'appuie sur les trois propriétés définies ci-dessous :

1. tout graphe possède un nombre pair de sommets de degré impair ;
2. un graphe dont tous les sommets sont de degré pair possède un cycle Eulérien ;
3. le problème du 2-mariage parfait de coût minimal se résout en temps polynomial (démontré dans [Edmonds, 1965]).

Grâce à ces propriétés, Christofides a découvert l'algorithme 30. L'opération la plus coûteuse de cet algorithme est le 2-mariage. Cette opération peut être réalisée en se ramenant à un problème de flot, et sa complexité est donc polynomiale.

---

#### Algorithme 30 : Algorithme de Christofides

---

**Entrées :** Graphe  $K_n = G$  Euclidien

Calculer l'arbre couvrant minimum  $T$  de  $G$ ;

Soit  $G^*$  le sous-graphe de  $G$  sur les sommets de degré impair de  $T$ ;

Déterminer un 2-mariage minimum  $M$  sur  $G^*$ ;

$T \cup M$  est un graphe dont tous les sommets sont de degré pair ; trouver un circuit Eulérien  $\Gamma$  dans ce graphe;

Réduire  $\Gamma$  grâce au même principe que dans l'algorithme 29.

---

*Exemple 35* (Algorithme de Christofides). Reprenons l'exemple 34, et appliquons lui l'algorithme de Christofides.

On obtient le même arbre couvrant  $T$  (figure 6.4), puis on extrait le sous-graphe  $G^*$  en s'appuyant sur les sommets de degré impair de  $T$ .

On calcule le 2-mariage parfait de coût minimal  $M$  de  $G^*$  (voir figure 6.5).

Le graphe  $G' = T \cup M$  (représenté figure 6.6) possède alors par construction uniquement des sommets de degré pair. Il existe donc un circuit Eulérien  $\Gamma$  sur  $G'$ . Par exemple :  $\Gamma = ACDEAFBA$ .

Il reste alors à réduire  $\Gamma$  en ne considérant un sommet que lors de sa première apparition. On obtient alors le cycle  $\Gamma' = ACDEFBA$  de longueur 55 (voir figure 6.7).

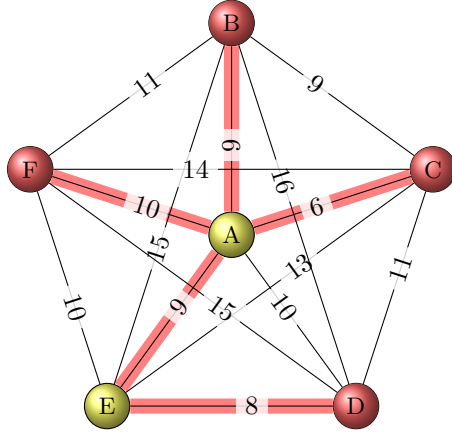


FIGURE 6.4 – Sommets de degré impair de l'arbre couvrant  $T$

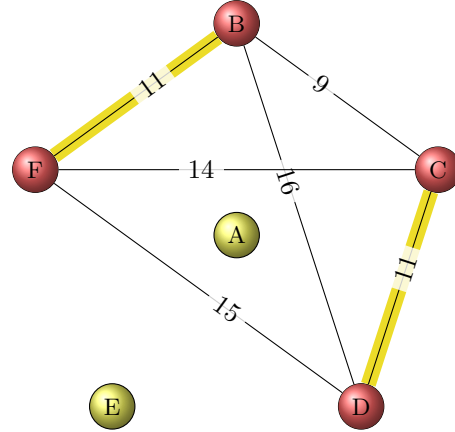


FIGURE 6.5 – Sous-graphe  $G^*$

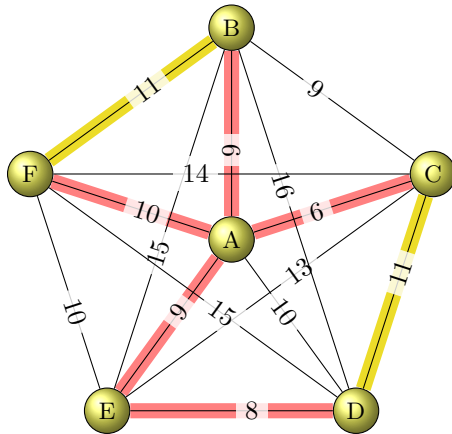


FIGURE 6.6 –  $T \cup \mathcal{M}$

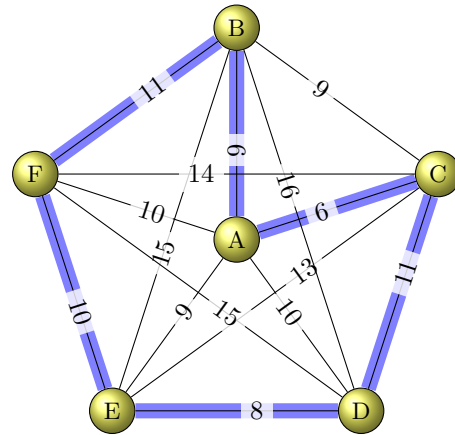


FIGURE 6.7 – Tournée calculée par Christofides

## 6.2 Problème du cycle Hamiltonien

Un cycle Hamiltonien est un chemin dans un graphe non orienté qui visite chaque sommet exactement une fois. Déterminer si un tel chemin existe est un problème NP-Complet appelé problème du cycle Hamiltonien.

Les cycles Hamiltoniens sont nommés d'après William Rowan **Hamilton**, inventeur du jeu icosien qui se joue sur un dodécaèdre régulier. À partir d'un sommet choisi et en suivant les arêtes, il s'agit de trouver un chemin passant une et une seule fois par chacun des 20 sommets pour revenir au point de départ. Comme le chemin était difficile à visualiser sur le dodécaèdre, Hamilton le remplaça, sur un plan, par un graphe isomorphe au dodécaèdre.

Hamilton a résolu ce problème en utilisant le calcul icosien. Malheureusement, cette solution ne se généralise pas à des graphes quelconques.

Un graphe contenant un cycle Hamiltonien est appelé graphe Hamiltonien.

*Exemple 36* (Graphes Hamiltoniens). Les graphes complets  $K_n$  pour  $n > 2$  sont Hamiltoniens.



### 6.3 Variations autour de l'arbre couvrant minimum

Chercher un arbre couvrant minimum se fait en  $\mathcal{O}(n^2)$  opérations par l'algorithme de Prim. Il est toutefois dangereux d'ajouter des contraintes sur le type d'arbre recherché :

- Chercher un arbre couvrant minimal dont les sommets sont de degré au plus 3 est NP-difficile.
- Si on se restreint à des sommets de degré au plus 2, on retrouve le problème du voyageur de commerce.
- Chercher un arbre couvrant ayant le plus possible de feuilles est NP-difficile.
- Le diamètre d'un graphe est la longueur maximale d'un plus court chemin entre deux sommets de ce graphe. Chercher un arbre couvrant de diamètre minimal est NP-difficile.

### 6.4 Problème de la clique maximum

Une clique de  $G$  est un sous-graphe complet de  $G$ . Ce terme a été introduit dans [Luce et Perry, 1949], qui utilise des sous-graphes complets dans les modèles sociaux pour modéliser des cliques de personnes, c'est-à-dire des groupes de personnes dans lequel chacun connaît tous les autres.

Le problème de la clique maximum est un problème d'optimisation consistant à trouver une clique de degré maximum dans un graphe. Le problème de décision associé consiste à chercher si une clique de taille  $k$  existe dans un graphe.

Ce problème est équivalent à la recherche d'un stable – sous-graphe dépourvu d'arête – maximum.

### 6.5 Isomorphisme de graphes

Soient deux graphes  $G_1$  et  $G_2$ . Le problème de l'isomorphisme de graphes consiste à déterminer si  $G_1$  est isomorphe à un sous-graphe de  $G_2$ . Ce problème est une généralisation à la fois du problème de la clique maximale et de la recherche de cycle Hamiltonien ; il est donc NP-Complet.



## Table des figures

1.1	Machine de Turing	12
1.2	Automate de la machine de Turing calculant $\lambda x[2x]$	13
1.3	Tableau représentant la fonction de transition de la machine de Turing	13
1.4	P et NP	17
2.1	Problème linéaire	21
2.2	Algorithme du simplexe	29
2.3	Problème sans solution	33
2.4	Problème non borné	34
3.1	Königsberg et ses ponts	38
3.2	Deux représentations du graphe de Petersen	39
3.3	Deux représentations supplémentaires du graphe de Petersen	39
3.4	Graphe orienté pondéré	42
3.5	Liste d'adjacences, graphe pondéré et non pondéré	43
3.6	Matrice d'adjacences, graphe pondéré et non pondéré	43
3.7	Graphe et décomposition en composantes fortement connexes	46
3.8	Plus court chemin dans un graphe	52
3.9	Graphe multiple	60
3.10	Ajout de tâche fictive	60
3.11	Graphe PERT	62
3.12	Graphe MPM	63
3.13	Modification du PERT initial	64
4.1	Arbre couvrant minimum	69
4.2	Arbre couvrant, $u$ et $C_u$	70
4.3	Arbre couvrant, $v$ et $\Omega_v$	70
4.4	Graphe	71
4.5	Arbre couvrant minimum	71
4.6	Graphe contracté	73
4.7	Graphe contracté	73
4.8	Coloration	77
4.9	Cliques	78
4.10	Carte dont la coloration nécessite quatre couleurs	79

5.1	Réseau de transport	82
5.2	Flot dans un réseau	82
5.3	Réseau à sources (a,b) et puits (c,d) multiples	83
5.4	Supersource et superpuits	84
5.5	Amélioration de flot	85
5.6	Réseau résiduel	85
5.7	Flot amélioré	85
5.8	Coupe	86
5.9	Graphe piège pour l'algorithme de Ford-Fulkerson	87
5.10	Algorithme primal 1	92
5.11	Algorithme primal 2	92
5.12	Algorithme primal 3	92
5.13	Algorithme primal 4	92
6.1	Graphe initial	94
6.2	Arbre couvrant	94
6.3	Tournée calculée	95
6.4	Sommets de degré impair de l'arbre couvrant $T$	96
6.5	Sous-graphe $G^*$	96
6.6	$T \cup \mathcal{M}$	96
6.7	Tournée calculée par Christofides	96

## Liste des Algorithmes

1	Algorithme du tri bulles . . . . .	8
2	Principe de l'algorithme du simplexe . . . . .	22
3	Parcours en largeur: bfs(graphe $G$ , nœud <i>depart</i> ) . . . . .	44
4	Parcours en profondeur: dfs(graphe $G$ , nœud $n$ ) . . . . .	45
5	Composantes fortement connexes: cfc(Graphe $G$ ) . . . . .	46
6	Algorithme de Tarjan: tarjan(Graphe $G$ ) . . . . .	47
-	Procédure desc(sommet $s$ ) . . . . .	47
7	Initialisation: init(graphe $G$ , sommet $s$ ) . . . . .	49
8	Algorithme de relâchement: relacher(sommet $a$ , sommet $b$ ) . . . . .	49
9	Algorithme de Bellman-Ford: bellmanFord(graphe $G$ , sommet $s$ ) . . . . .	50
10	Algorithme de Dijkstra: dijkstra(graphe $G$ , sommet $s$ ) . . . . .	52
11	Affichage d'un plus court chemin . . . . .	54
12	Extension des chemins: extension(matrice $D$ , matrice $W$ ) . . . . .	55
13	Plus court chemin pour tout couple de sommets . . . . .	55
14	Plus court chemin pour tout couple de sommets version 2 . . . . .	55
15	Algorithme de Floyd-Warshall . . . . .	56
16	Algorithme de PERT: Pert(Réseau $R = (S, A, d)$ ) . . . . .	58
17	Algorithme de MPM: Mpm(Réseau $R = (S, A, d)$ ) . . . . .	59
18	Chemin critique MPM: CrMpm(Réseau $R = (S, A, d)$ ) . . . . .	59
19	Recherche d'arbre couvrant minimum par l'algorithme de Kruskal . . . . .	70
20	Recherche d'arbre couvrant minimum par l'algorithme de Kruskal v2 . . . . .	72
21	Recherche d'arbre couvrant minimum par l'algorithme de Prim . . . . .	72
22	Recherche d'arbre couvrant minimum par l'algorithme de Prim . . . . .	74
23	Méthode de Ford-Fulkerson . . . . .	84
24	Algorithme de Ford-Fulkerson . . . . .	87
25	Procédure pousser : pousser $\delta(x, y)$ unités de flot de $x$ vers $y$ . . . . .	88
26	Procédure ré-étiqueter : accroît la hauteur de $x$ . . . . .	88
27	Initialisation du préflot dans un réseau $R = (S, A, c)$ . . . . .	89
28	Procédure préflot dans un réseau $R = (S, A, c)$ . . . . .	89

29	Approximation de tournée minimale . . . . .	94
30	Algorithme de Christofides . . . . .	95

## Bibliographie

- D. AVIS et V. CHVÁTAL : Notes on bland's pivoting rule. *Polyhedral Combinatorics*, pages 24–34, 1978.
- Claude BERGE : *Théorie des graphes et applications*. Dunod, 1967.
- Robert G. BLAND : New finite pivoting rules for the simplex method. *Mathematics of Operations Research*, pages 103–107, 1977.
- Karl H. BORGWARDT : The average number of pivot steps required by the simplex-method is polynomial. *Mathematical Methods of Operations Research*, 26:157–177, 1982. ISSN 1432-2994. 10.1007/BF01917108.
- Nicos CHRISTOFIDES : Worst-case analysis of a new heuristic for the travelling salesman problem. Rapport technique, DTIC Document, 1976. URL <http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA025602>.
- Stephen COOK : The complexity of theorem proving procedures. In *The third annual ACM symposium on Theory of computing*, pages 151–158, 1971. URL <http://www.comp.nus.edu.sg/~cs5234/2012/Lectures/Papers/Cook-1971-A4.pdf>.
- Thomas H. CORMEN, Charles E. LEISERSON, Ronald L. RIVEST et Clifford STEIN : *Introduction à l'algorithmique*. Dunod, 2<sup>e</sup> édition, 1994.
- George DANTZIG : *Linear Programming and Extensions*. Princeton University Press, 1963.
- D. DE WERRA, T. M. LIEBLING et JF. HÊCHE : *Recherche opérationnelle pour ingénieurs I*, volume 1. Presses polytechniques et universitaires romandes, 1<sup>re</sup> édition, 2003.
- Edsger Wybe DIJKSTRA : A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. ISSN 0029-599X. URL <http://dx.doi.org/10.1007/BF01386390>.
- Jack EDMONDS : Paths, trees, and flowers. *Canadian Journal of mathematics*, 17(3):449–467, 1965. URL <http://web.eecs.umich.edu/~pettie/matching/Edmonds-paths-trees-flowers.pdf>.
- Robert FAURE : *Précis de recherche opérationnelle*. Dunod, 1979.

- Robert FAURE, Bernard LEMAIRE, Bernard LEMAIRE et Christophe PICOULEAU : *Précis de recherche opérationnelle. Méthodes et exercices d'application*. Dunod, 6<sup>e</sup> édition, 2009.
- Jean-Claude FOURNIER : *Théorie des graphes et applications*. Hermes, 2006.
- Jean-Claude FOURNIER : *Graphes et applications*, volume 1. Hermes, 2007a.
- Jean-Claude FOURNIER : *Graphes et applications*, volume 2. Hermes, 2007b.
- Georges GONTHIER : A computer-checked proof of the four colour theorem. <http://research.microsoft.com/~gonthier/4colproof.pdf>.
- A. KAUFMAN : *Méthodes et modèles de la recherche opérationnelle*, volume 2. Dunod, 1964.
- Victor KLEE et George J. MINTY : How good is the simplex algorithm. In *Inequalities III*, pages 159–175. Academic Press, New York, 1972.
- Donald E. KNUTH : *The art of computer programming*, volume 1. Addison-Wesley, 1968.
- R Duncan LUCE et Albert D PERRY : A method of matrix analysis of group structure. *Psychometrika*, 14(2):95–116, 1949. URL <http://aris.ss.uci.edu/~lin/50.pdf>.
- Michel MINOUX : *Programmation mathématique*. Lavoisier, 2<sup>e</sup> édition, 2008.
- Jean-François REY : *Calculabilité, complexité et approximation*. Vuibert, 2004.
- Michel SAKAROVITCH : *Optimisation combinatoire, graphes et programmation linéaire*. Hermann, 1984a.
- Michel SAKAROVITCH : *Optimisation combinatoire, programmation discrète*. Hermann, 1984b.
- Jacques STERN : *Fondements mathématiques de l'informatique*. McGraw Hill, 1990.
- Robert E. TARJAN : Depth first search and linear graph algorithms. *SIAM journal on Computing*, pages 146–160, 1972. URL [http://dutta.csc.ncsu.edu/csc791\\_spring07/wrap/dfs.pdf](http://dutta.csc.ncsu.edu/csc791_spring07/wrap/dfs.pdf).