INFORMATIQUE
2024 - 2025

Algorithmes sur les Graphes et Combinatoire

Laurent PHILIPPE

SUP FC

UNIVERSITÉ DE FRANCHE-COMTÉ

UNIVERSITÉ DE FRANCHE-COMTÉ

Module INFO-AGC

L. Philippe

Algorithmique sur les Graphes et Combinatoire

Masters Informatique DVL / I2A / ISL
1ère année

Cours

CTU Centre de Télé-enseignement Universitaire
UNIVERSITÉ DE FRANCHE-COMTÉ

Centre de télé enseignement
Filière Informatique
Domaine Universitaire de la Bouloie
25030 Besançon Cedex (France)

Table des matières

Introduction

Présentation du cours

A tous les niveaux de l'informatique, il est nécessaire de modéliser les données et le problème auxquels nous nous intéressons pour les traduire sous forme de structures de données et d'algorithmes. La modélisation vise à coder puis à traiter l'information considérée tout en garantissant la validité et l'efficacité du traitement. L'utilisation de structures de données connues permet alors de profiter de tous les acquis et outils liés à ces structures. Les structures de graphes sont ainsi très largement utilisées en informatique pour modéliser des problèmes aussi variés que la recherche d'un chemin entre deux ou un ensemble de sommets, la création d'emplois du temps, l'ordonnancement ou l'affectation de tâches, etc. Les domaines d'utilisation des graphes sont nombreux car il s'agit d'une structure de données ou une structure abstraite généraliste. Dans ce contexte, la première partie de ce module a pour objectif de donner des éléments, des outils ou des méthodes utiles à la modélisation à base de graphes et de présenter les principaux algorithmes sur les graphes. Une bonne compréhension des algorithmes de ce chapitre permettra l'application ou la réutilisation des principes vus dans de nombreux problèmes.

Beaucoup de problèmes posés aux informaticiens sont combinatoires, c'est-à-dire que la recherche d'une solution suppose d'énumérer tout ou partie des solutions valides. Dans le cas contraire, par exemple lorsque une solution peut-être trouvée à partir d'une équation mathématique, nous parlons de résolution directe. Nous parlons d'optimisation si l'objectif est de trouver une solution donnant une valeur maximale ou minimale pour un critère, une valeur caractéristique, parmi l'ensemble des combinaisons possibles. Par exemple, trouver un chemin entre deux points est simplement un problème d'algorithmique sur les graphes mais trouver un chemin de longueur minimale entre deux points est un problème combinatoire d'optimisation, qui utilise l'algorithmique sur les graphes, car il suppose d'explorer les chemins possibles pour trouver celui qui est de longueur minimale. Dans le cas de l'optimisation pour les problèmes de grande taille, les solution naïves sont soit rarement efficaces, avec une complexité algorithmique élevée, soit donnent des résultats non optimaux, la combinaison trouvée n'est pas la meilleure. Il est alors important de connaître les conditions de résolution de ces problèmes, leur complexité, et les techniques dont nous disposons pour apporter des solutions optimales, ou proches de l'optimal. La difficulté de ce domaine tient généralement au fait que l'efficacité de la résolution des problèmes peut être très différente d'un problème à l'autre. Par exemple, la recherche des plus courts chemins à partir d'une origine unique peut-être réalisée en un temps de calcul polynomial alors que la résolution du problème du voyageur de commerce, la recherche

du plus court chemin reliant une et une seule fois tous les sommets d'un graphe complet, engendre des temps de calcul exponentiels. Ce cours présente ainsi des techniques classiques d'optimisation, comme la programmation linéaire, la programmation dynamique ou la programmation gloutonne, et leurs conditions d'utilisation pour garantir la qualité de la solution produite.

L'algorithmique sur les graphes et optimisation sont des domaines très liés car, d'une part, certains problèmes d'optimisation peuvent se ramener à des problèmes sur les graphes, et d'autre part, certains problèmes sur les graphes sont des problèmes d'optimisation. Ceci sera illustré par le dernier chapitre qui traite de l'optimisation pour des problèmes de graphes.

Pré-requis

Ce cours est un cours d'algorithmique. Il s'appuie sur les compétences précédemment acquises dans les cours d'algorithmique pour développer des algorithmes reposant sur la théorie des graphes et pour proposer des approches algorithmiques à la résolution de problèmes d'optimisation combinatoire [1].

Comme nous l'avons dit précédemment ce cours est un cours d'algorithmique. Il ne revient pas sur les concepts fondamentaux de l'algorithmique : structures de données simples ou complexes et structures de contrôle. Il suppose donc acquis une maîtrise de l'écriture d'algorithmes avec des structures de données telles que des matrices (tableaux à 2 dimensions), les listes, les files ou les piles.

L'écriture des algorithmes se fait dans un langage "*algorithmique*" qui n'est pas formellement défini mais qui est largement utilisé dans la communauté informatique. Il n'y a donc aucun pré-requis quant à un langage de programmation particulier.

Objectifs du cours

L'objectif de ce cours est de donner à l'étudiant qui le suit une connaissance des possibilités offertes par la modélisation à base de graphes et par différentes techniques de recherche de solutions optimales ou proches de l'optimal dans des problèmes d'optimisation.

Les compétences acquises à l'issue de ce cours sont la définition d'algorithmes sur des structures de type graphe ou arbre et l'utilisation de techniques pour résoudre des problèmes d'optimisation.

Recommandations

Les éléments présentés doivent être mis en application à travers les exercices donnés dans la rubrique "*Exercices*" sur moodle mais aussi en exécutant les algorithmes présentés sur des exemples. Par exemple, la plupart des algorithmes sur les graphes prennent en entrée

1. Vous trouverez une définition formelle du terme *optimisation combinatoire* sur Wikipedia

des graphes quelconques. Il est alors facile d'exécuter l'algorithme sur un graphe simple dessiné sur une feuille. Ces exercices de déroulement permettent de mieux comprendre, donc de mieux maîtriser, les algorithmes.

Ne vous laissez pas impressionner par certains aspects mathématiques du cours, toutes les notions abordées sont à votre portée si vous êtes arrivés jusque là.

N'hésitez pas à consulter d'autres sources d'information si les définitions et explications données dans le cours vous paraissent difficiles à comprendre : une approche différente permet parfois de résoudre ce problème.

Il est également important que vous fassiez part de vos questions et remarques dans le forum. Cela permet à l'enseignant de voir vos difficultés et aux autres étudiants de profiter des réponses qui sont données.

Bibliographie et webographie

Une part non négligeable de ce cours a été rédigée grâce au "*Cormen*", livre incontournable en algorithmique dont voici la référence :

— Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction à l'algorithmique*. DUNOD. 2002.

Vous trouverez par ailleurs sur le web un grand nombre de supports de cours qui vous permettrons d'élargir votre connaissance des graphes si vous le souhaitez. Attention cependant à ne pas vous éloigner du périmètre du cours en cherchant une notion. J'ai essayé de simplifier les notations pour ne pas leur donner un aspect trop rébarbatif, ce qui n'est pas toujours le cas. Il peut être tout aussi efficace de poser la question dans le forum.

Pour ceux qui apprennent mieux en vidéo diverses ressources permettent de voir d'une autre manière les notions du cours.

Pour les graphes :

https://www.youtube.com/channel/UCHtJVeNLyR1yuJ1_xCK1WRg

Pour la programmation linéaire. Un exemple qui détaille la modélisation puis la résolution graphique du problème :

https://www.youtube.com/watch?v=60TiTsAJ22Y
https://www.youtube.com/watch?v=3kjTNEyjQ0E

https://www.youtube.com/watch?v=xKYPdYunois

Notations mathématiques

La définition des graphes, et des algorithmes qui les utilisent, repose en partie sur des notations mathématiques, principalement des notations ensemblistes. Nous rappelons ici la signification des notations utilisées dans le cours.

$\forall$: quelque soit
$\exists$: il existe

$\wedge$: et

$\vee$: ou

$\Rightarrow$: implique

$\neg$: négation

$\neq$: différent

∞ : infini

$\sum_{x=0}^{N}$: somme pour $x = 0$ jusqu'à $x = N$

$\{\ldots\}$: définition d'un ensemble

$\in$: appartient

$\notin$: n'appartient pas

$\emptyset$: ensemble vide

$|V|$: cardinal de l'ensemble V

$V \times V'$: produit cartésien des ensembles V et V', ensemble composé de tous les couples possibles constitués de deux éléments, un de V et un de V'.

$E' \subseteq E$: l'ensemble E' est un sous-ensemble, inclus dans E ou égal

$\bigcup$: union

$V \setminus \{u\}$: ensemble V moins l'élément u

Objets algorithmiques

Les algorithmes donnés dans le cours et les exercices s'appuient sur quelques objets classiques en programmation. Nous donnons ici la syntaxe utilisée (inspirée de Java) dans le cours et qui doit être utilisée dans les rendus.

Pile<type> :
- *boolean empty*() : pile vide
- *type peek*() : lit le premier élément sans le dépiler
- *type pop*() : dépile
- *push*(*type*) : empile

File<type> :
- *add*(*type*) : ajoute un élément à la fin de la file
- *boolean empty*() : file vide
- *type peek*() : lit le premier élément de la file sans le supprimer
- *type poll*() : lit et supprimer le premier élément

Liste<type> :
- *add*(*type*) : ajoute un élément à la fin de la liste
- *addAll*(*List*) : concatène la liste en paramètre à la fin de la liste
- *boolean contains*(*type*) : l'élément est dans la liste
- *boolean empty*() : liste est vide
- *type get*(*int*) : lit, sans le supprimer, l'élément dont l'index est donné en paramètre
- *type remove*(*index*) : supprime l'élément dont l'index est donné en paramètre
- *set*(*index*, *type*) : change l'élément dont l'index est donné en paramètre

— *int size*() : taille de la liste
Pour les listes les indexes commencent en 0.

Rappels sur les preuves

Dans certains exercices du cours il est demandé de prouver un résultat, par exemple qu'un algorithme glouton est optimal ou non.

Prouver une assertion ou une propriété revient à montrer qu'elle est vraie quelque soit le contexte. Si nous reprenons le cas d'un algorithme glouton, prouver qu'il est optimal (l'assertion est alors "l'algorithme glouton trouve toujours une solution optimale au problème") revient à prouver que, quelque soit l'instance considérée, l'algorithme nous donne une solution et qu'aucune autre solution ne peut pas donner un meilleur résultat.

À noter qu'il suffit de trouver une instance fausse pour prouver qu'une assertion est fausse mais qu'il faut prouver que toutes les instances sont justes pour prouver qu'elle est vraie.

Plusieurs approches de preuve peuvent être utilisées. Nous développons ici rapidement la preuve directe, la preuve par l'absurde ou la preuve par récurrence. Nous les illustrons par l'exemple du problème suivant : un explorateur veut remplir son sac qui peut contenir un volume donné V, à partir d'objets qui ont tous le même volume, mais pas le même poids, de manière à ce que le sac soit le plus léger possible. L'algorithme qui choisit d'abord les objets les plus légers est optimal.

Preuve directe

Dans une preuve directe, il faut prouver que quelque soit l'instance, le résultat est optimal. Une preuve directe repose, à partir de propriétés vraies, sur une suite de déductions logiques pour arriver à l'assertion finale.

Dans le cas du sac de l'explorateur il est possible de calculer toutes les sommes possibles pour constater que la somme qui contient les objets les plus légers est optimale.

Preuve par l'absurde

Dans une preuve par l'absurde nous commençons par supposer que l'assertion à prouver est fausse et nous montrons, par une suite de déductions logiques, que nous arrivons à une contradiction.

Dans le cas du sac de l'explorateur, nous prouvons l'optimalité de la solution en supposant qu'il existe une solution optimale qui contient des objets qui ne sont pas parmi les objets les plus légers. Or, dans cette solution, il est possible de remplacer un objet plus lourd par un plus léger puisqu'ils font la même taille. Nous obtenons ainsi un sac dont le poids est inférieur à la solution initiale, donc cette solution n'est pas optimale. Nous pouvons en déduire que seule une solution avec les objets les plus légers est optimale.

Preuve par récurrence

La preuve pas récurrence fonctionne sur les problèmes qui disposent d'une dimension. Nous vérifions alors que l'assertion est correcte pour la dimension 0 ou 1, nous supposons qu'elle est vraie pour une dimension n et nous utilisons cette propriété pour montrer qu'elle est vraie pour une dimension $n + 1$.

Si l'explorateur n'a la possibilité de ne mettre qu'un seul objet dans son sac alors il est évident qu'en prenant le plus léger il aura le sac le plus léger (dimension 1). Nous supposons que l'explorateur a mis les n objets les plus légers dans son sac et que celui-ci est le plus léger qu'il puisse obtenir (dimension n). Si nous considérons un sac avec une place de plus (dimension $n + 1$), alors ajouter l'objet le plus léger permet d'augmenter le moins possible le poids du sac. Puisque le sac de dimension n était le plus léger alors le sac de dimension $n + 1$ sera le plus léger puisque nous avons ajouté le moins de poids possible.

Chapitre 1

Graphes et algorithmes

Le concept de graphe sert à modéliser des entités en relation. Il existe de très nombreuses situations où nous avons des entités en relation. Par exemple, les utilisateurs d'un réseau social sont des entités en relation avec leurs contacts : les entités sont les utilisateurs et la relation est "l'utilisateur X est en contact avec l'utilisateur Y". D'autres exemples d'entités en relation sont les pages web et les hyperliens qui les lient, les réseaux de communication avec les routeurs et les câbles qui les relient, les villes et les routes qui les relient, les états d'un système et les transitions entre états, etc.

Tous ces exemples peuvent être modélisés sous la forme de graphes. Du point de vue informatique, l'intérêt de modéliser un système sous la forme d'un graphe est qu'il existe de nombreux algorithmes ou bibliothèques qui savent apporter des solutions aux problèmes généraux comme vous l'avez déjà vu pour d'autres structures de données : tableaux, listes, etc.

Dans cette première partie du cours nous introduisons les principales définitions et propriétés classiquement utilisées sur les graphes en même temps que les algorithmes qui y sont associés. Ce chapitre utilise un formalisme mathématique, nous y parlons d'ensembles, de cardinal, etc. mais nous avons essayé pour chaque définition ou propriété, de donner une signification plus concrète de manière à ce que ces dernières soient facilement compréhensibles. Nous donnons le formalisme mathématique sous l'appellation "*Précision*" ou "*Formellement*". Ces parties, mise en vert, peuvent être considérées comme optionnelles.

1.1 Introduction aux graphes

En général, la définition d'un graphe repose sur deux structures : l'ensemble des sommets et la famille des arêtes. Les sommets sont les entités et les arêtes sont les relations. Dans le cas de notre exemple de réseau social, nous avons donc un ensemble de sommets qui sont les utilisateurs et une famille d'arêtes qui sont les contacts entre les utilisateurs. Modéliser un système sous la forme d'un graphe revient donc à définir ces deux structures.

1.1.1 Graphe

Un *graphe* est une structure constituée d'un ensemble de *sommets*[1] (*Vertex* en anglais) et d'une famille E d'*arêtes* (*Edge* en anglais). L'ensemble des sommets est habituellement noté V et la famille des arêtes est habituellement notée E. Un graphe G est donc défini à partir de son ensemble de sommets et de son ensemble d'arêtes, ce que nous notons sous la forme $G(V, E)$.

Supposons que nous avons un réseau social composé de six personnes, Laura, Paul, Naddir, Thomas, Amine et Gautier, tel que :

- Laura et Paul sont en contact
- Thomas et Paul sont en contact
- Thomas et Gautier sont en contact
- Paul et Naddir sont en contact
- Naddir et Laura sont en contact
- Naddir et Amine sont en contact
- Gautier et Paul sont en contact
- Gautier et Paul sont en contact

L'ensemble V est alors constitué de $\{Laura, Paul, Naddir, Thomas, Amine, Gautier\}$ et la famille E est constituée des couples $\{\{Laura, Paul\}, \{Thomas, Paul\}, \{Thomas, Gautier\}, \{Paul, Naddir\}, ...\}$.

Lorsque nous avons besoin de généraliser, nous notons $\{x_1, x_2, \ldots, x_n\}$ les sommets du graphe, les éléments de l'ensemble V, ou $\{y_1, y_2, \ldots, y_n\}$, ou $\{z_1, z_2, \ldots, z_n\}$.

Précision *: les arêtes sont des éléments du produit cartésien $V \times V$, c'est-à-dire qu'ils sont constitués de couples dont chacun des deux éléments est pris dans l'ensemble V, puisqu'ils relient deux sommets de V. A noter qu'un élément de $V \times V$ peut apparaître plusieurs fois dans une même famille E, qui n'est donc pas appelée ensemble ici.*

1.1.2 Diagramme sagittal d'un graphe

Savoir comment représenter les entités et les relations d'un graphe est très utile. Par exemple, la figure 1.1 donne deux représentations possibles des relations au sein d'un réseau social. Les deux représentations ne donnent pas le même niveau de compréhension, en particulier il est plus facile d'avoir un aperçu global des relations si nous regardons la figure 1.1b que si nous regardons la figure 1.1a.

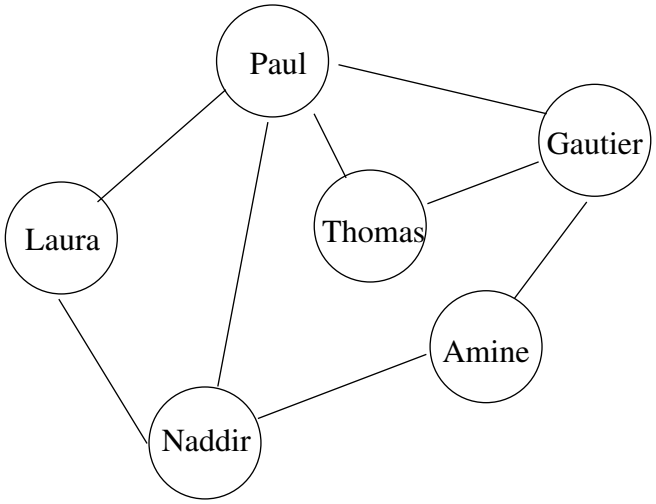
La représentation donnée sur la figure 1.1b est appelée diagramme sagittal. C'est une représentation d'un graphe où les sommets sont des points ou des cercles et les arêtes des liens. C'est la représentation classique d'un graphe sous une forme visuelle.

Dans la suite nous représentons souvent les graphes sous cette forme sans forcément utiliser l'appellation diagramme sagittal et en les appelant simplement graphe.

1. L'ensemble est toujours fini, c'est à dire qu'il est possible de compter le nombre de sommets

Laura et Paul sont en contact
Thomas et Paul sont en contact
Thomas et Gautier sont en contact
Paul et Naddir sont en contact
Naddir et Laura sont en contact
Naddir et Amine sont en contact
Gautier et Paul sont en contact
Gautier et Amine sont en contact

(a) Représentation textuelle



(b) Représentation graphique

FIGURE 1.1 – Représentations des contacts dans un réseau social

1.1.3 Boucles et P-Graphes

Les boucles et les arêtes dupliquées sont deux particularités qui peuvent exister dans les graphes.

Dans certaines relations une entité peut-être en relation avec elle-même. Dans le graphe, cela se traduit par une *boucle* qui est une arête d'un sommet vers lui-même. Par exemple, l'arête de a vers a de la figure 1.2 est un exemple de boucle.

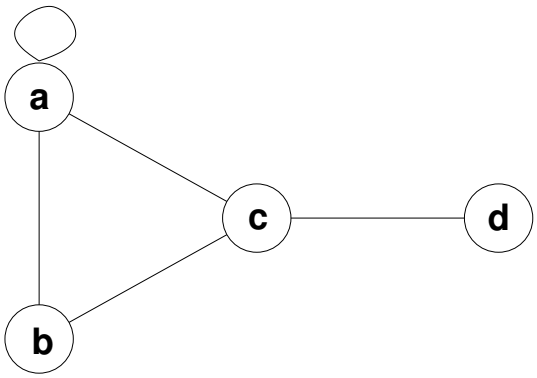


FIGURE 1.2 – Exemple de graphe avec une boucle

Deux sommets d'un graphe peuvent être reliés par plus d'une arête s'il y a plus d'une relation qui les lient, par exemple deux relations possibles entre un sommet x et un sommet y. Un *p-graphe* est alors défini comme un graphe qui n'admet pas plus de p arêtes entre deux sommets quelconques de V. Dans les *1-graphes* chaque sommet n'a pas plus d'une seule arête qui le relie à un autre sommet.

La figure 1.3 donne un exemple de 2-graphe. La valeur 2 représente bien ici le nombre maximal d'arêtes entre deux sommets.

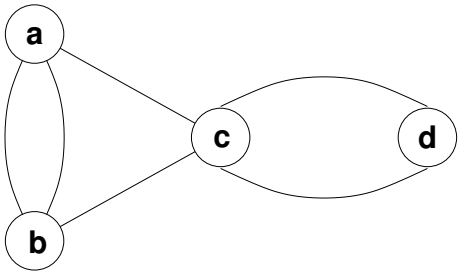


FIGURE 1.3 – Exemple de 2-graphe

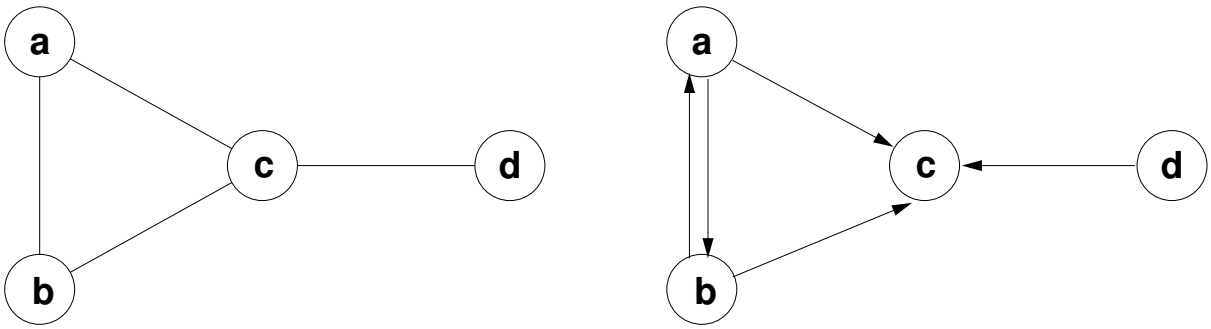
Précisions *:*
Définition $G(V, E)$ *est un p-graphe s'il est tel que :*

$$\begin{cases} V = \{x_1, x_2, \ldots, x_n\} \text{ avec } |V| = n \text{ le nombre de sommets du graphe } G. \\ E : \text{famille de couples } (x, y) \in V \times V, \text{ formant les arêtes de } G \text{ avec } |E| = m. \\ \forall (x, y) \in V \times V, |\{(x, y) \in E\}| \leq p \end{cases}$$

Dans un 1-graphe, la famille E des arêtes, de G devient un ensemble car il n'y pas deux fois le même élément

Dans la suite du cours, on appelle *graphe* tout 1-graphe sans boucle et nous ne considérons plus que ce type de graphe.

1.1.4 Orientation d'un graphe



(a) Graphe non orienté (b) Graphe orienté

FIGURE 1.4 – Exemples de graphes

Dans un graphe, chaque arête possède un sommet initial et un sommet terminal. Lorsque cette distinction n'est pas pertinente, le graphe est *non orienté*. Par exemple, le graphe des relations est un graphe non orienté car si une personne a est en contact avec une personne b alors la personne b est aussi en contact avec la personne a et il n'y a pas lieu de distinguer un sommet initial d'un sommet final. Dans la suite les arêtes sont notées $\{x, y\}$.

La relation entre deux sommets peut également modéliser une relation où il y a lieu de distinguer le sommet initial du sommet terminal. Nous pourrions, à partir des sommets de notre réseau social, représenter les appels téléphoniques au sein du groupe. La relation traduit alors la personne a a appelé la personne b. Dans ce cas il y a bien un sommet initial, la personne a, et un sommet final, la personne b. On parle alors de graphe *orienté* et les liens sont des arcs, notés (x, y). Sur le diagramme sagittal, les arcs sont notés avec une flèche donnant leur sens.

La figure 1.4 donne des exemples de graphe, respectivement orienté et non orienté. Pour ces figures, nous avons $V = \{a, b, c, d\}$ pour les sommets. Sur la figure 1.4a, l'ensemble des arêtes est $E = \{\{a, b\}, \{a, c\}, \{b, c\}, \{c, d\}\}$. Sur la figure 1.4b, l'ensemble des arcs est $E = \{(a, b), (a, c), (b, a), (b, c), (d, c)\}$ et l'ordre des couples a un sens. Par exemple, sur la figure 1.4b, (d, c) est dans E mais (c, d) n'y est pas et l'arc (a, b) et différent de l'arc (b, a).

Dans le cas des graphes orientés, on peut définir deux propriétés : la symétrie et l'anti-symétrie.

Définition 1.1 : Graphe symétrique

Un graphe orienté $G(V, E)$ est *symétrique* si, pour tout arc (x, y) de E, il existe l'arc inverse (y, x) dans E .

Formellement : $(x, y) \in E \Rightarrow (y, x) \in E$

La figure 1.5 donne un exemple de graphe symétrique puisque tous les arcs sont doublés. À noter qu'un graphe symétrique peut être plus simplement représenté par un graphe non-orienté.

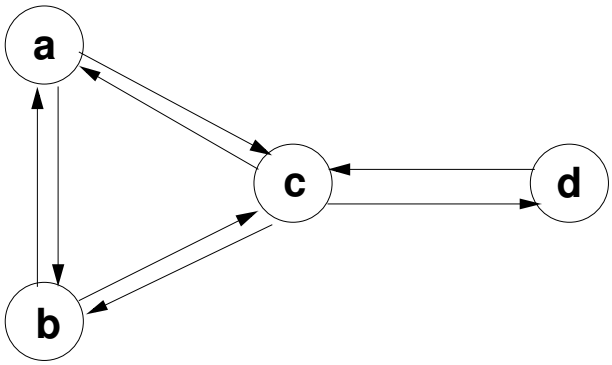


FIGURE 1.5 – Exemple de graphe symétrique

Définition 1.2 : Graphe antisymétrique

Un graphe orienté $G(V, E)$ est *antisymétrique* si, pour tout arc (x, y) de E, il n'existe pas l'arc inverse (y, x) dans E .

Formellement : $(x, y) \in E \Rightarrow (y, x) \notin E$

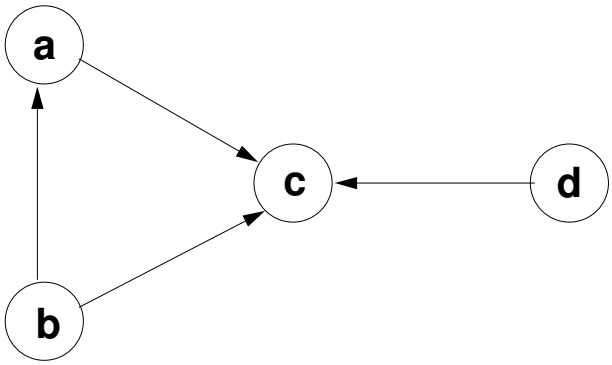


FIGURE 1.6 – Exemple de graphe antisymétrique

La figure 1.6 donne un exemple de graphe antisymétrique puisque aucun arc n'est doublé. La notion de symétrie/antisymétrique ne s'applique par aux graphes non-orientés.

Remarque : Un graphe orienté sans circuit (un Directed Acyclic Graph ou `DAG`) est antisymétrique.

Dans la suite de cette partie, nous ne ferons la distinction entre arc et arête que lorsque cela est nécessaire et nous utilisons le terme d'arête de manière générique [2]. Les définitions s'appliquent donc aussi bien à l'une ou l'autre forme de relation.

1.1.5 Sous-graphe et graphe partiel

Un sous-graphe $G'(V', E')$ d'un graphe $G(V, E)$ est un graphe dont les ensembles de sommets et d'arêtes constituent des sous-ensembles des ensembles des ensemble de sommets et d'arêtes du graphe G. Le sous-graphe $G'(V', E')$ est donc la partie du graphe G qui est constituée des sommets de V' et des arêtes qui sont adjacentes à deux sommets de V'.

Formellement : *soit $G(V, E)$ un graphe. Le graphe $G'(V', E')$, pour lequel $V' \subseteq V$ et E' est une restriction de E à $V' \times V'$, est appelé* sous-graphe *de G engendré par V'.*

Si $V' = V$ et $E' \subseteq E$ alors G' est un *graphe partiel* de G. C'est le graphe G auquel des arêtes ont été retirées. La notion de sous-graphe partiel se déduit alors de la combinaison des deux précédentes : le graphe partiel d'un sous-graphe, donc un sous-graphe dont on a retiré des arêtes.

1.1.6 Vocabulaire

Nous donnons dans cette partie quelques éléments de vocabulaires indispensables à la compréhension des algorithmes sur les graphes.

2. De la même manière qu'en anglais on utilise le terme *edge*

Définition 1.3 : Incidence

Une arête est dite incidente à un sommet si une de ses extrémités est ce sommet. Par exemple, sur la figure 1.4a, l'arête $\{a, b\}$ est incidente aux sommets a et b et sur la figure 1.4b, l'arc (d, c) est incident aux sommets c et d.

Définition 1.4 : Adjacence des arêtes et arcs

Deux arêtes sont *adjacentes* (ou *voisines*) si au moins une de leurs extrémités est commune, les deux arêtes sont donc incidentes au même sommet. Par exemple, sur la figure 1.4a, les arêtes marquées $\{a, b\}$ et $\{b, c\}$ sont adjacentes.

Formellement : $\{x, y\} \in E$ *a une arête adjacente si* $\exists z \in V | \{x, z\} \in E \vee \{y, z\} \in E \vee \{z, x\} \in E \vee \{z, y)\} \in E$.

Définition 1.5 : Adjacence des sommets

Deux sommets sont *adjacents* (ou *voisins*) si au moins une arête relie les deux sommets. Si le graphe est non-orienté il pourra s'agir de l'arête $\{x, y\}$ ou $\{y, x\}$. Par exemple, sur la figure 1.4a, les sommets a et c sont adjacents au sommet b.

Formellement : $x \in V$ *a un sommet adjacent si* $\exists y \in V | \{x, y\} \in E$.

Définition 1.6 : Successeur/Prédécesseur

Soit $(x, y) \in E$, un arc du graphe orienté G. Le sommet y est le *successeur* du sommet x et x est le *prédécesseur* de y. La définition ne s'applique pas aux graphes non-orientés. Par exemple, sur la figure 1.4b, le sommet c est le successeur du sommet d et le sommet b est le prédécesseur du sommet c.

On note habituellement $\Gamma(x)$ l'application qui donne tous les sommets adjacents (ou voisins) de x pour un graphe non orienté et les successeurs de x pour un graphe orienté. A noter que l'application $\Gamma(x)$ peut associer plusieurs images à un même élément.

Sur la figure 1.4b, nous avons $\Gamma(c) = \emptyset$ et $\Gamma(b) = \{a, c\}$

À noter qu'il est également possible de définir l'ensemble des successeurs (ou prédécesseurs) d'un sous-ensemble A de sommets comme l'ensemble des successeurs des sommets de A.

Formellement : $\Gamma(A) = \bigcup_{x \in A} \Gamma(x)$

1.2 Représentation informatique

La structure de graphe est donc un modèle qui formalise les relations entre des objets d'un ensemble. Jusque là nous avons représenté cette structure soit sous la forme de deux ensembles, sommets et arêtes, soit sous la forme d'un diagramme sagittal. Sur cette base il est possible d'écrire des algorithmes génériques tels que l'algorithme 1.

Algorithme 1 : Exemple d'algorithme générique

Données :
l : liste des voisins de s ayant 2 voisins **initialisée à** liste vide
$nbVoisins$: entier

```
1 Fonction listVoisins2Voisin(G, s) :
2 début
3     pour chaque sommet u voisin de s faire
4         nbVoisins ← 0
5         pour chaque sommet v voisin de u faire
6             nbVoisins ← nbVoisins + 1
7         si nbVoisins = 2 alors l.add(v)
8     retourner l
```

Néanmoins, si les représentations sous forme d'ensembles ou de diagramme sagittal conviennent bien à l'interprétation humaine, elles ne sont pas adaptées à une utilisation informatique.

Pour exploiter ce modèle en informatique, il est donc nécessaire de définir des structures de données permettant, d'une part, la mémorisation de la structure, c'est-à-dire de l'ensemble des sommets et des arêtes, et d'autre part son exploitation au sein d'algorithmes, sachant que ce choix a une incidence sur la complexité des algorithmes qui l'utilise et donc sur leurs performances.

Nous commençons par présenter les structures de données de bas niveau, basées sur des tableaux à une ou deux dimensions avant des présenter des structures de plus haut niveau, basées siur des listes ou des objets. Pour chacune de ces structures nous expliquons les arguments qui peuvent conduire à les choisir pour représenter un graphe.

1.2.1 Les matrices d'adjacence

Les matrices sont adaptées pour représenter des graphes, on parle de matrice d'adjacence. En particulier les matrices à deux dimensions, qui sont implémentées sous la forme de tableaux à deux dimensions, sont adaptées pour représenter les graphes simples (1-graphes) qu'ils soient orientés ou non. Dans ce cas chacun des sommets du graphe est associé à un indice. Pour les graphes orientés, un arc simple (i, j) se traduit par une valeur 1 pour l'élément (i, j) de la matrice : $A[i][j] = 1$. Pour les graphes non orientés, une arête $\{i, j\}$ est représentée par deux valeurs à 1 dans la matrice d'adjacence : $Ai][j] = 1$ et $A[j][i] = 1$ alors que si le graphe est orienté

Un graphe $G(V, E)$ est donc représenté par une matrice d'adjacence A définie comme suit :

$$A = \begin{pmatrix} m_{1,1} & \cdots & m_{1,j} & \cdots & m_{1,n} \\ \vdots & \ddots & \vdots & & \vdots \\ m_{i,1} & \cdots & m_{i,j} & \cdots & m_{i,n} \\ \vdots & & \vdots & \ddots & \vdots \\ m_{n,1} & \cdots & m_{n,j} & \cdots & m_{n,n} \end{pmatrix} \text{ avec } m_{i,j} = \begin{cases} 1 \text{ si } (i,j) \text{ est une arête du graphe} \\ 0 \text{ sinon} \end{cases}$$

Par exemple, le graphe exemple donné à la figure 1.4a, en prenant les sommets dans l'ordre a, b, c, d, peut être représenté par la matrice d'adjancence $_{no}$ suivante. Le graphe exemple donné à la figure 1.4b peut être représenté par la matrice d'adjancence $_{o}$.

$$A_{no} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \qquad A_{o} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

La lecture de la matrice se fait à partir du côté gauche, chacune des lignes correspondant à un sommet de départ pour une arête, et du dessus, chacune des colonnes correspondant à un sommet d'arrivée. Ainsi, dans la ligne 4 de la matrice $_{no}$ nous pouvons voir qu'il n'y qu'une seule valeur à 1, celle de la colonne 3. Cette valeur traduit le fait qu'il y a une arête qui part du sommet d et qu'il arrive au sommet c.

Cette matrice peut être implémentée de manière simple, par exemple en langage Java, sous la forme suivante :

$$intA[][] = \{\{0,1,1,0\},\{1,0,1,0\},\{1,1,0,1\},\{0,0,1,0\}\}$$

Les matrices d'adjacence sont aussi bien adaptées pour représenter les graphes orientés, dans ce cas l'élément (i, j) de la matrice sera différent de l'élément (j, i), que des graphes non-orientés, dans ce cas l'élément (i, j) de la matrice sera identique à l'élément (j, i). Nous pouvons remarquer que la matrice obtenue pour un graphe orienté n'est pas symétrique, sauf pour le cas où le graphe est lui-même symétrique, alors qu'elle le sera systématiquement lorsque le graphe n'est pas orienté.

Avantages et inconvénients

Plusieurs critères entrent en compte dans le choix d'une structure de données pour représenter un graphe : le coût d'accès aux arêtes et sommets, le coût de stockage et la possibilité d'ajouter/supprimer facilement des arêtes ou des sommets si le graphe change, que nous appelons la dynamicité.

La représentation d'un graphe sous forme de matrice d'adjacence offre l'avantage de nous permettre de savoir immédiatement s'il existe, ou non, une arête entre les sommets x et y à partir de l'élément $[x][y]$. Le coût de recherche d'une arête est de $O(1)$, ce qui peut être intéressant pour l'efficacité d'un algorithme qui utilise principalement les arêtes. C'est,

par exemple le cas de l'algorithme 6 qui utilise une matrice pour calculer les plus courts chemins entre les sommets du graphe.

Le coût de stockage de cette struture est en $O(n^2)$, avec n est le nombre de sommets. Si le graphe est peu dense, c'est-à-dire que le nombre d'arêtes est proportionnel au nombre de sommets ($|E| \sim |V|$), cette représentation n'est pas satisfaisante puisque la matrice A contient dans ce cas majoritairement des zéros. Elle consomme donc inutilement de la place mémoire. Par exemple, la matrice du graphe donné à la figure 1.4b a un tiers des ses valeurs à 0.

Si la structure du graphe doit être changée, elle peut l'être de deux manières : en ajoutant ou supprimant des arêtes ou en ajoutant ou supprimant des sommets. La représentation sous forme de matrice d'adjacence est bien adaptée pour la modification des arêtes puisqu'il suffit d'accéder à l'élément $[x][y]$ pour le mettre à 0 ou 1, en fonction du besoin. Comme pour l'accès, l'ajout d'une arête a un coût en $O(1)$. Par contre, pour l'ajout ou la suppression d'un sommets, puisque la structure est entièrement statique, il est nécessaire de refaire complètement la matrice. La structure de matrice d'adjacence n'est donc pas adaptée si le graphe est appelé à être modifié au cours de son utilisation. C'est, par exemple, le cas lorsqu'un graphe est généré incrémentalement à partir de données. À noter que les algorithmes vu dans ce cours sont principalement basés sur des ensembles statiques de sommets.

Une manière de représenter les graphes, en limitant l'occupation mémoire, est de ne traduire que les arêtes existantes dans le graphe, donc les ensembles d'adjacence de chaque sommet. Pour l'exemple de la figure 1.4a, il suffit de traduire le fait qu'il existe des arêtes depuis le sommet a vers les sommets b et c, des arêtes depuis le sommet b vers les sommets a et c, etc.

1.2.2 Successeurs/Premier Successeur

Une solution, basée sur des structures statiques, pour utiliser le mode de représentation basé sur les ensembles d'adjacence est de lister les voisins de chaque sommet sous la forme d'une énumération stockée dans un tableau d'entiers à une dimension. Les sommets du graphe sont numérotés de 0 à $n-1$ et nous enregistrons dans un tableau, nommé FS (File des Successeurs), successivement le numéro des sommets voisins du sommet 0, puis du sommet 1, etc. Le passage de la suite formée des successeurs du sommet i à celle des voisins du sommet $i+1$ est marqué par un séparateur. Par convention, le séparateur est -1 mais on peut aussi utiliser 0 si les sommets sont numérotés de 1 à n.

Pour accéder au voisinage du sommet i, sans avoir à passer en revue l'ensemble des voisins des sommets de numéro inférieur à i, on utilise un deuxième tableau, appelé APS (Adresse du Premier Successeur), dans lequel la $ième$ case désigne l'indice, dans le tableau des voisins FS, du premier voisin du sommet i.

À noter qu'avec ce tableau la présence du séparateur dans le tableau des voisins n'est pas indispensable. Il est maintenu car cela permet d'accélérer certains calculs. Par exemple le parcours des voisins d'un sommet peut se faire jusqu'à trouver le séparateur, sans avoir à utiliser la valeur contenue dans le tableau des premiers successeurs.

En associant les valeurs 0, 1, 2, 3 successivement aux sommets a, b, c, d, l'exemple du

graphe 1.4a conduit aux tableaux FS et APS suivants :

FS :	1	2	−1	0	2	−1	0	1	3	−1	2	−1

APS :	0	3	6	10

Avantages et inconvénients

La représentation FS/APS possède une bonne efficacité pour l'accès aux arêtes. En effet le coût d'accès à une arête se décompose entre le coût d'accès au sommet dans le tableau APS, pour trouver l'adresse de la liste des successeurs du sommet, et le coût d'accès à l'arête dans le tableau FS. Le premier est en $O(1)$ puisque c'est un accès direct dans un tableau. Le second nécessite le parours des cases du tableau FS jusqu'à trouver l'arête ou un délimiteur ce qui, au pire cas, est en $O(n)$ avec n le nombre de sommets.

Le stockage des structures se fait dans deux tableaux de taille n pour APS et $n+m$ pour FS, avec n le nombre de sommets et m le nombre d'arêtes. Le coût global en mémoire est donc réduit au minimum.

Par contre, si ce type de représentation permet une manipulation simple et une faible occupation mémoire elle manque de dynamisme si le graphe doit changer, que ce soit pour ajouter/supprimer un sommet ou une arête. En effet, l'insertion ou la suppression d'éléments est coûteuse dans ces tableaux dans la mesure où elle implique souvent de créer de nouveaux tableaux.

Voir dans les exercices un algorithme permettant de passer d'une représentation du type *Matrice d'Adjacence* à la représentation FS et APS. Un exercice demandant le calcul du graphe inverse travaille également sur cette représentation du graphe.

1.2.3 Listes chaînées

Les listes chaînées permettent de prendre en compte des structures de données qui évoluent au cours du temps. Elles sont donc adaptées si le graphe à représenter est succeptible d'évoluer au cours de son utilisation.

La représentation peut changer suivant que seules les arêtes peuvent être modifiées ou que les arêtes et les sommets peuvent l'être. Si seules les arêtes peuvent être modifiées, la structure envisagée se décompose en un tableau de sommets et, pour chaque case correspondant à un sommet, une liste des sommets voisins. Si les sommets et les arêtes peuvent être modifiés alors le tableau des sommets peut transformé en liste des sommets.

La représentation du graphe de la figure 1.4b sous la forme d'un tableau de listes de voisins est donnée par la figure 1.7, avec une numérotation de 1 à 4 des sommets.

En fonction de l'implémentation de la liste (liste simplement chaînée, doublement chaînée, doublement chaînée circulaire, doublement chaînée avec sentinelle, etc) les parcours des sommets et des voisins peuvent se faire de différente manières avec des parcours à simple, à double sens, etc.

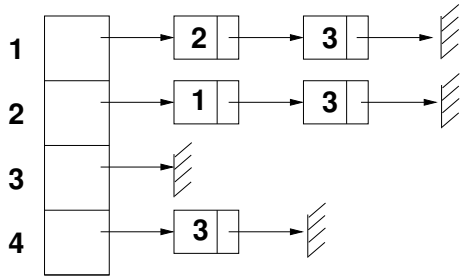


FIGURE 1.7 – Tableau des listes des voisins (ou des suivants) du graphe de la figure 1.4b

Avantages et inconvénients

Les listes chaînées cumulent beaucoup d'avantage dans la représentation d'un graphe et l'implémentation des algorithmes sur les graphes. Leur coût mémoire a une complexité réduite en $O(n+m)$, bien que la structure des listes prenne plus de place que les tableaux FS et APS. Les accès aux arêtes sont également rapides en $O(n)$, où n est le nombre de sommets et m est le nombre d'arêtes. Elles permettent un ajout rapide d'arêtes, voire de sommets si ces derniers sont stockés dans une liste. À cela vient s'ajouter les possibilités offertent par les différentes bibliothèques ou package de listes qui facilitent grandement leur manipulation.

1.2.4 Packages et biliothèques pour les graphes

La structure de graphe étant largement utilisée il existe des bibliothèques qui proposent des objets et fonctions qui facilitent grandement la manipulation des graphes pour des langages courants tels que Java ou Python. Nous recommandons la bibliothèque JGraphT [3] permettant de manipuler facilement les graphes en Java ou Python, avec une syntaxe proche des notations utilisées dans le cours, et nous donnons quelques exemples de programmes en accompagnement des exercices. D'autres bibliothèques peuvent également être utilisées telles que Guava [4] ou python-graph [5].

Remarque *: l'implémentation des algorithmes sur les graphes dépend fortement de la structure de données utilisée pour mémoriser le graphe. Comme nous l'avons évoqué certaines structures de données facilitent l'accès aux informations nécessaires pour atteindre notre objectif, d'autres sont plus adaptées à un autre objectif (consommation mémoire, temps d'exécution, etc.). Il est cependant possible de raisonner sur les algorithmes de manière générique en utilisant les structures d'ensemble qui définissent un graphe. Par exemple, une itération sur l'ensemble des sommets d'un graphe peut se faire de la manière suivante :*

pour chaque *sommet u du graphe* **faire**

Dans la suite du cours nous utilisons généralement ce formalisme "générique" pour définir

3. https://jgrapht.org/
4. https://github.com/google/guava/
5. https://github.com/pmatiello/python-grap

les algorithmes en restant indépendant des structures de données. Dans les exercices associés au cours nous voyons quelques exemples de transformation d'algorithmes génériques en algorithme utilisant une structure de données spécifique.

1.3 Chemins et parcours

La recherche de lien entre deux sommets ou entre un sommet et les autres sommets est importante pour savoir les sommets qui sont en relation.

1.3.1 Chemins et chaînes

Les chemins et les chaînes sont, comme leur nom l'indique, des suites de sommets que l'on parcourt dans l'ordre dans lequel ils sont donnés. La recherche de chemin ou de chaîne dans un graphe est utile par exemple pour savoir s'il est possible de passer d'un état à un autre pour un automate. Nous avons fait le choix de parler, dans le cours, de chemin pour les graphes orientés et de chaîne pour les graphes non orientés [6].

Avant d'aborder les algorithmes nous donnons les définitions des termes nécessaires à la compréhension.

Définition 1.7 : Chemin

Dans un graphe orienté G, un chemin est une suite de sommets tels chaque sommet de cette suite est connecté à son suivant par un arc du graphe.

La longueur d'un chemin est le nombre d'arcs utilisés dans la suite des sommets. Un chemin composé de deux sommets est donc de longueur 1, 2 avec trois sommets, etc.

Formellement *: Un chemin de longueur p est une suite finie $(x_0, x_1, \ldots, x_p)$ de sommets du graphe orienté $G(V, E)$ telle que pour tout i de 0 à $p-1$, $(x_i, x_{i+1}) \in E$.*

Sur la figure 1.4b, les suites (a, b, c) et (a, b, a, b, c) sont des chemins. Les suites (c, d) et (a, b, c, a) ne sont pas des chemins.

Définition 1.8 : Chaîne

Dans un graphe non-orienté G, une chaîne est une suite de sommets tels chaque sommet de cette suite est connecté à son suivant par une arête du graphe.

6. Attention l'appellation chaîne pour les graphes non-orientés et chemin pour les graphes orientés n'est pas forcément reconnue comme telle et les deux appellations sont souvent utilisées pour l'un ou l'autre. Cette définition est néanmoins cohérente avec celle de wikipedia(http://fr.wikipedia.org/wiki/ChaÃőne_(thÃľorie_des_graphes))

Formellement *: Une chaîne de longueur p est donc une suite finie $(x_0, x_1, \ldots, x_p)$ de sommets du graphe non-orienté $G(V, E)$ telle que pour tout i de 0 à $p-1$, $\{x_i, x_{i+1}\} \in E$ ou $\{x_{i+1}, x_i\} \in E$.*

Définition 1.9 : Ascendance/Descendance

Un sommet y est descendant de x s'il est sur un chemin (resp. chaîne) issu de x et que x est *l'ascendant* (ou *l'ancêtre*) de y. Attention les circuits (ou cycles) sont possibles et y peut donc également être l'ascendant de x.

Définition 1.10 : Chemin et chaîne élémentaires

Un chemin (resp. chaîne), est *élémentaire* si les sommets de la suite sont deux à deux distincts. Par convention, la suite ne possède qu'un sommet ou que le dernier sommet est égal au premier alors le chemin (resp. chaîne) est encore considéré comme élémentaire.

Dans la pratique une chaîne élémentaire est une chaîne qui ne passe pas deux fois par le même sommet puisque les sommets qui la composent sont tous différents. Sur la figure 1.4a, les suites (a, b, c), (a, b, a) et (a) sont des chemins élémentaires mais $(a, b, a, a, , b, c)$ ne l'est pas.

Définition 1.11 : Chemin et chaîne distincts

Deux chemins (resp. chaînes) sont dites distincts si les sommets qui les composent n'ont en commun que les sommets initiaux et finals.

Formellement *: Si $c = (x_0, x_1, \ldots, x_p)$ et $c' = (x_0, x'_1, \ldots, x_p)$ sont des chemins (resp. chaînes) ayant les mêmes sommets initiaux et finals, alors c et c' sont distincts si $c \cap c' = x_0, x_P$.*

Définition 1.12 : Circuit et cycle

Un circuit (resp. cycle pour un graphe non orienté [7]) est un chemin (resp. chaîne) fermé de longueur non nulle.

Formellement *: Un chemin (resp. chaîne) de longueur $p, p \neq 0$, $c = (x_0, x_1, \ldots, x_p)$ tel que $x_0 = x_p$ est un circuit (resp. un cycle).*

Sur la figure 1.4b, le chemin (a, b, a) est un circuit. Cette figure permet également de montrer qu'il peut y avoir un nombre infini de chemins si au moins l'un des chemins est un circuit. Pour créer un nouveau chemin il suffit en effet de parcourir une fois de plus ce circuit. Sur la figure on peut ainsi ajouter autant de circuits (a, b, a) que l'on veut pour faire un nouveau chemin. Par contre, il n'existe qu'un nombre fini de chemins élémentaires puisque ces derniers ne permettent pas de repasser par un sommet qui fait déjà partie du chemin.

7. Comme pour les appellations chaînes et chemins, cette définition est cohérente avec celle donnée par wikipédia (http://fr.wikipedia.org/wiki/Cycle_(thÃľorie_des_graphes)).

Définition 1.13 : Circuit et cycle élémentaires

Un circuit (resp. cycle) est dit élémentaire si les sommets qui le composent sont deux à deux distincts, en dehors du sommet de initial/final. Un circuit élémentaire ne passe donc pas deux fois par le même sommet.

Attention, les appellations cycle et circuit peuvent parfois être employées indifféremment pour des graphes orientés ou non. Par exemple, lorsqu'un graphe orienté ne possède pas de circuit on parle de graphe orienté acyclique.

1.3.2 Mesures dans un graphe

Définition 1.14 : Distance entre deux sommets

La distance entre deux sommets est définie comme la longueur de la plus courte chaîne (ou chemin) reliant ces deux sommets, la longueur d'une chaîne étant le nombre d'arêtes utilisées dans la suite des sommets. On note $d(u, v)$, la distance entre les sommets u et v.

Il est possible de définir également la distance entre deux arêtes $\{x, y\}$ et $\{u, v\}$ comme la plus petite distance entre les sommets incidents des deux arêtes.

Formellement : $d(\{x, y\}, \{u, v\}) = \min\{d(x, u),\ d(x, v),\ d(y, u),\ d(y, v)\}$

Sur la figure 1.4a, la distance entre les sommets a et d est 2.

Définition 1.15 : Diamètre

Le *diamètre* d'un graphe G, noté $D(G)$, est la plus grande distance existant entre deux sommets du graphe. Il est défini comme étant la plus grande des plus courtes chaînes du graphe.

Formellement : $D(G) = \max_{x,y \in G} d(x, y)$

Dans le graphe de la figure 1.4a, le diamètre est de 2.

Définition 1.16 : Degré d'un sommet

Le *degré* d'un sommet x de G, noté $d_G(x)$, est le nombre d'arcs incidents à ce sommet. Par exemple, sur la figure 1.4a, le degré du sommet a est $d_G(a) = 2$, celui du sommet c est $d_G(a) = 3$.

Un sommet isolé, qui n'a pas d'arc incident, a donc un degré *nul*

Proposition 1.1 ***Somme des degrés des sommets d'un graphe***

La somme des degrés des sommets x d'un graphe $G(V, E)$ est paire. Cette somme est donnée par :

$$\sum_{x \in V} d_G(x) = 2|E|$$

Vous pouvez valider cette propriété expérimentalement sur les graphes donné dans le cours. Plus particulièrement la somme des degrés du graphe de la figure 1.4a est de 8, ce qui est effectivement pair. Intuitivement il est facile de comprendre qu'un arc ayant deux extrémitées, chacune comptant pour un degré, la somme des degrés est égale à deux fois le nombre d'arêtes.

1.3.3 Recherche de chemin ou de cycle

La recherche de chemins dans un graphe est un problème classique de l'algorithmique sur les graphes. Un grand nombre d'algorithmes existent avec des objectifs différents : recherche de chemins depuis un sommet vers tous les autres, recherche d'un plus court chemins entre deux sommets, etc.

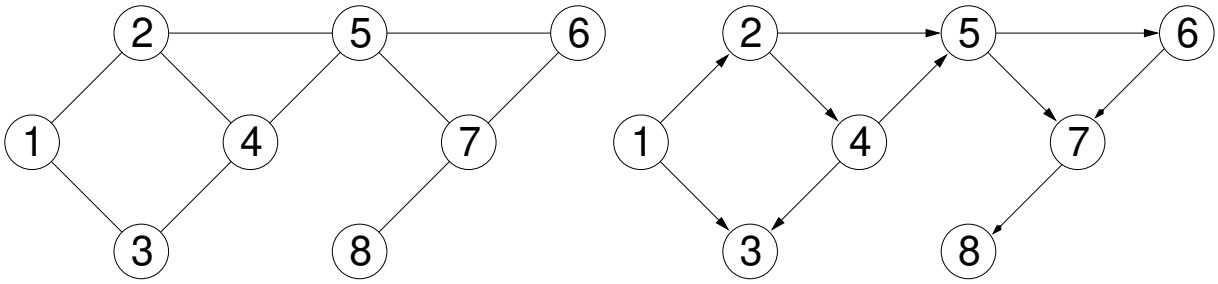
Le problème simple de trouver un chemin entre deux sommets d'un graphe repose sur l'idée simple de partir de l'un des deux sommets, de regarder si le sommet cible se trouve parmi ses voisins et, si cela n'est pas le cas, de regarder chez les voisins des voisins, etc. jusqu'à trouver le sommet cible. Cette idée repose donc sur le parcours d'un graphe, dont nous abordons les algorithmes dans la partie 1.3.4. Le parcours est simplement arrêté lorsque le sommet cible est trouvé.

La recherche de cycle est également importante en algorithmique sur les graphes car elle peut servir à calculer une tournée pour un véhicule de livraison, dans la mesure où ce dernier rentre à la fin de sa tournée. Les algorithmes de calcul de cycles sont abordés, comme des problèmes d'optimisation, au chapitre 5.

1.3.4 Parcours

L'objectif d'un algorithme de parcours est d'explorer les sommets du graphe de proche en proche. Un algorithme de parcours choisi, à partir d'une liste de sommets déjà visités, le prochain sommet parmi les voisins de ces sommets qui n'ont pas encore été visités.

Pour définir plus formellement la notion de parcours nous devons d'abord définir ce qu'est la bordure d'un graphe.



(a) Graphe G non orienté (b) Graphe G' orienté

FIGURE 1.8 – Graphes non orienté et orienté

Définition 1.17 : Bordure d'un sous-graphe

Pour un sous-graphe T d'un graphe $G(V, E)$, la *bordure*, notée $B(T, G)$, est l'ensemble des sommets n'appartenant pas à T et voisins (ou successeurs) d'au moins un sommet de T. À la différence de l'ensemble des voisins (ou des prédécesseurs) de T, la bordure $B(T, G)$ ne contient donc aucun sommet qui appartient déjà à T.

Par exemple, sur le graphe 1.8a, la bordure du sommet 4 est l'ensemble $\{2, 3, 5\}$, la bordure du sommet 5 est l'ensemble $\{2, 4, 6, 7\}$ et la bordure du sommet 6 est l'ensemble $\{5, 7\}$. La bordure du sous-graphe composé des sommets 4, 5 et 6 est donc l'ensemble $\{2, 3, 7\}$. Sur le graphe 1.8b, la bordure du sommet 4 est l'ensemble $\{3, 5\}$, la bordure du sommet 5 est l'ensemble $\{6, 7\}$ et la bordure du sommet 6 est l'ensemble $\{7\}$. La bordure du sous-graphe composé des sommets 4, 5 et 6 est donc l'ensemble $\{3, 7\}$.

Définition 1.18 : Parcours générique

Une permutation $L = (x_1, \ldots, x_n)$ [8] des sommets d'un graphe G est un *parcours générique* du graphe à partir du sommet s_1 si et seulement si, à partir du sommet courant, le sommet suivant est choisi dans la bordure des sommets déjà parcourus.

Formellement *: Pour un graphe $G(V, E)$, tel que $|V| = n$, si on note $L[1, j]$ les j premiers sommets parcourus et x_j le jième élément de cette liste, alors $\forall j \in \{2, \ldots, n\}$, $B(L[1, j-1], G) \neq \emptyset \Rightarrow x_j \in B(L[1, j-1], G)$*

En général on parle simplement de parcours pour les parcours génériques.

Un parcours doit donc inclure une et une seule fois tous les sommets du graphe. Ainsi, les permutations $L_1 = (1, 3, 4, 5, 7, 8, 6, 2)$ et $L_2 = (5, 6, 2, 7, 8, 1, 3, 4)$ sont des parcours possibles du graphe donné à la figure 1.8a. Par contre la permutation $L_3 = (5, 6, 3, 2, 1, 4, 8, 7)$ n'est pas un parcours étant donné que $s_3 \notin B(L[5, 6])$.

Dans un parcours, la manière de choisir le prochain sommet, détermine le type du parcours. Nous définissons dans la suite les parcours en largeur et en profondeur.

Définition 1.19 : Sommet ouvert ou fermé

Un sommet ouvert est un sommet ayant au moins un voisin qui n'a pas déjà été parcouru (qui n'appartient pas déjà au parcours). A l'inverse, un sommet fermé est un sommet dont tous les voisins ont déjà été parcourus.

Formellement *: Un sommet x d'un parcours $L[1, i] = (x_1, \ldots, x_i)$ d'un graphe G est dit ouvert dans $L[1, i]$ s'il possède au moins un voisin (resp. successeur) dans G appartenant à $L[i+1, n]$. Dans le cas contraire, le sommet x sera dit fermé dans $L[1, i]$ pour G.*

Définition 1.20 : Parcours en largeur

Un parcours en largeur choisi, parmi la liste des sommets ouverts, le sommet voisin du premier sommet ouvert du parcours.

8. Une permutation est une liste complète des sommets du graphe, ordonnée de manière quelconque.

Formellement *: un parcours $L = (x_1, \ldots, x_n)$ est dit en en largeur si $\forall j \in \{2, \ldots, n\}$ tel que $B(L[1, j-1], G) \neq \emptyset$, le sommet x_j est voisin (resp. successeur) du premier sommet ouvert pour $L[1, j-1]$ dans G.*

Un parcours en largeur consiste donc à parcourir tous les sommets adjacents à un sommet avant de passer à l'exploration de ses fils, donc à toujours fermer les premiers sommets du parcours en premier. Cela revient aussi à parcourir les sommets à distance k d'un sommet x de départ avant les sommets placés à distance $k+1$.

Pour le graphe G donné en exemple à figure 1.8a, le parcours $L_5 = (1, 2, 3, 4, 5, 6, 7, 8)$ est un parcours en largeur, tout comme le parcours $L_6 = (6, 7, 5, 8, 4, 2, 3, 1)$. Par contre le parcours $L_{6bis} = (6, 5, 7, 8, 4, 2, 3, 1)$ n'est pas un parcours en largeur car $s_4 = 8$ n'est pas adjacent au premier sommet ouvert pour $L[1, 3]$ dans G (sommet 5).

Définition 1.21 : Parcours en profondeur

Un parcours en profondeur choisi, parmi la liste des sommets ouverts, le sommet voisin du dernier sommet ouvert du parcours.

Formellement *: un parcours $L = (x_1, \ldots, x_n)$ est dit en parcours en profondeur si $j\forall \in \{2, \ldots, n\}$ tel que $B(L[1, j-1], G) \neq \emptyset$, le sommet x_j est voisin (resp. successeur) du dernier sommet ouvert pour $L[1, j-1]$ dans G.*

Un parcours en profondeur consiste donc à parcourir d'abord les sommets fils avant de revenir aux sommets parents lorsque nous avons exploré tous les voisins du dernier fils, donc les fils sont fermés.

Pour le graphe G donné en exemple à figure 1.8a, le parcours $L_7 = (2, 5, 4, 3, 1, 6, 7, 8)$ est un parcours en profondeur de G. Par contre le parcours $L_8 = (5, 7, 8, 2, 4, 1, 3, 6)$ n'est pas un parcours en profondeur car $s_4 = 2$ n'est pas adjacent au dernier sommet ouvert pour $L[1, 3]$ dans G (sommet 7).

Un algorithme de parcours respecte les propriétés d'un parcours, c'est-à-dire que l'ordre des sommets est tel que l'exploration se fait de proche en proche, parmi les voisins qui n'ont pas encore été visités.

1.3.5 Algorithme de parcours en largeur

Nous voyons ici un algorithme possible pour ce parcours.

Principe de l'algorithme : on part d'un sommet s et on cherche à atteindre tous les sommets possibles en faisant la différence entre les sommets qui n'ont pas encore été parcourus, ceux qui ont déjà été visités mais dont les voisins n'ont pas encore été parcourus, et les sommets dont on a déjà marqué tous les voisins. Pour cela on utilise le code couleur suivant : les sommets est non visités sont $BLANC$, les sommets visités

sont $GRIS$ et les sommets dont tous les voisins ont été marqués sont $NOIR$. La limite $BLANC/GRIS$ marque donc la limite d'exploration. La couleur $NOIR$ n'est utilisée que pour illustrer la progression de l'algorithme comme sur la figure 1.10 mais, si elle n'est pas utilisée, il faut faire passer la couleur du sommet s à $GRIS$ au début.

Algorithme générique : l'algorithme 2 propose une implémentation itérative minimale du parcours en largeur sur la base de l'utilisation d'une file de sommets. Un algorithme légèrement différent, récursif, est donné dans les exercices.

Algorithme 2 : parcours en largeur d'un graphe $G(V, E)$ à partir du sommet s

Données : f : file des sommets en cours **init à** file vide
$couleur[u]$: $couleur[u] \in \{BLANC, GRIS, NOIR\}$, **init à** $BLANC$, $\forall u \in V$

```
1  Fonction parcoursLargeur(G, s) :
2  début
3      f.add(s)
4      couleur[s] ← GRIS
5      tant que ¬f.empty() faire
6          u ← f.poll()
7          pour chaque sommet v voisin de u faire
8              si couleur[v] = BLANC alors
9                  couleur[v] ← GRIS
10                 f.add(v)
11         couleur[u] ← NOIR // optionel
```

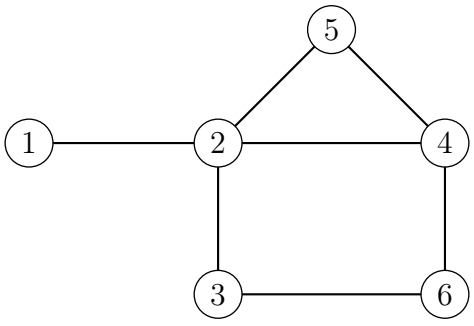


FIGURE 1.9 – Exemple de graphe

La figure 1.10 déroule[9] une exécution pas à pas de l'algorithme 3 sur l'exemple de graphe donné à la figure 1.9. Les sommets sont colorés au fur et à mesure de l'avancée de l'algorithme.

9. Il est recommandée de réaliser ce type de déroulement sur chacun des algorithmes qui est vu par la suite, même si celui-ci n'est pas détaillé dans le cours, pour bien en comprendre le fonctionnement.

L'algorithme 2 seul ne présente que peu d'intérêt puisqu'il n'a pas d'action sur le graphe et ne collecte pas de donnée. Il constitue par contre une base à adapter pour réaliser un grand nombre de traitements sur un graphe. Par exemple collecter des données, faire une recherche, un calcul, etc. Nous illustrons cette adaptation avec le calcul de la distance entre un sommet s et les autres sommets du graphe.

Algorithme de calcul de distance : l'algorithme utilise l'algorithme générique 3 pour calculer la distance du sommet s à tous les autres sommets. Il construit également une trace du parcours pour mémoriser le chemin entre le sommet s et tous les autres sommets. À chaque découverte d'un sommet v $BLANC$, issu du sommet u, l'arête (u, v) est ajoutée dans l'arborescence avec u *pere* de v ou u *prédécesseur* de v. Il permet ainsi de trouver le plus court chemin entre le sommet s et tout sommet du graphe.

Algorithme 3 : parcours en largeur d'un graphe avec calcul de distance

Données : f : file des sommets en cours **initialisé à** file vide
$couleur[u]$: $couleur[u] \in \{BLANC, GRIS, NOIR\}$, **initialisée à** $BLANC$, $\forall u \in V$
Résultat : $pere[u]$: le père de u, **initialisé à** $NULL$, $\forall u \in V$
$d[u]$: distance de s à u, **initialisée à** ∞, $\forall u \in V$

```
1  Fonction parcoursLargeur(G, s) :
2  début
3      d[s] ← 0
4      f.add(s)
5      tant que ¬f.empty() faire
6          u ← f.poll()
7          pour chaque sommet v voisin de u faire
8              si couleur[v] = BLANC alors
9                  couleur[v] ← GRIS
10                 d[v] ← d[u] + 1
11                 pere[v] ← u
12                 f.add(v)
13         couleur[u] ← NOIR
14     retourner pere, d
```

Dans la figure 1.10 Les distances trouvées sont affichées à côté des sommets au fur et à mesure du parcours.

Nous pouvons remarquer qu'avec cet algorithme seuls les sommets qui sont accessibles par une chaîne depuis le sommet s sont visités. De ce fait la distance à s des sommets non accessibles reste à ∞ ce qui est logique.

A noter également que l'arbre du parcours est conservé dans le sens *fils* $\rightarrow$ *père* et non dans le sens contraire comme il est d'usage pour le stockage des arbres de type *arbre de recherche*. Dans ce cas, la reconstruction d'un parcours peut se faire récursivement de la destination à la source, comme indiqué plus loin dans ce paragraphe.

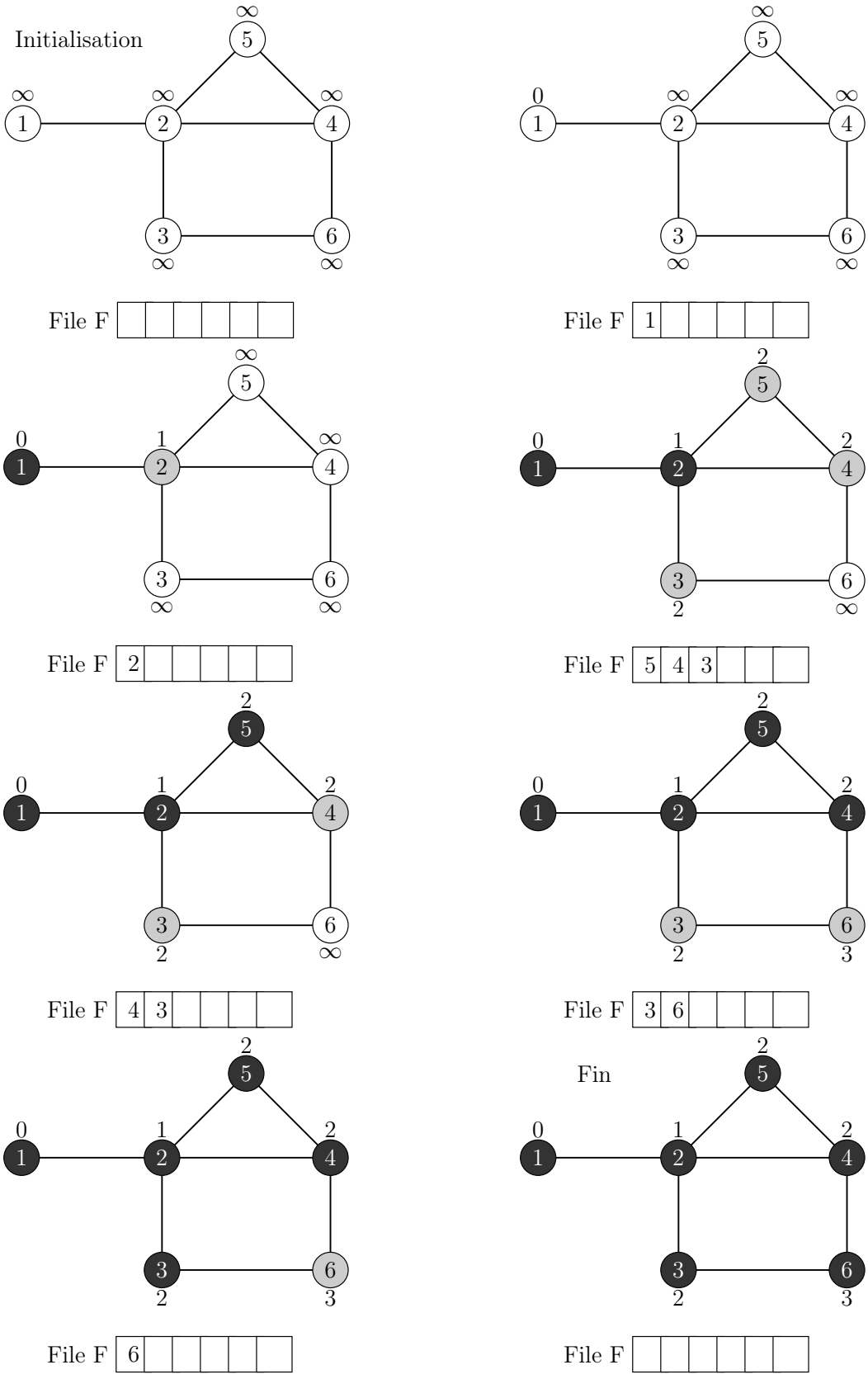


FIGURE 1.10 – Exemple de parcours en largeur d'un graphe à partir du sommet 1

1.3.6 Algorithme de parcours en profondeur

Principe de l'algorithme : à l'inverse du parcours en largeur, le parcours d'un graphe en profondeur est la descente au plus profond du graphe jusqu'à trouver un sommet dont les sommets voisins ont tous déjà été explorés ou un sommet qui n'a pas d'autre voisin que celui par lequel on l'a atteint. On revient en arrière jusqu'au dernier sommet ayant un voisin non encore exploré. C'est alors le point de départ d'une nouvelle descente. Ce processus se répète jusqu'à ce que tous les sommets aient été explorés. Ainsi, le sous-graphe correspondant à ce parcours est une forêt d'arborescences en profondeur. Nous adoptons également pour les sommets du parcours le code couleur $BLANC$ et $GRIS$, respectivement pour les sommets non encore visités et ceux qui le sont déjà.

L'algorithme générique 4 proposé est une solution de ce parcours basée sur une pile. Le même algorithme programmé sous forme récursive est donné en exercice. Son expression devient alors plus concise puisque la pile est implicitement gérée par la récursivité.

Algorithme 4 : Parcours en profondeur du graphe $G(V, E)$ à partir du sommet s

Données : p : pile des sommets en cours,**init à** $pileVide()$
$couleur[u]$: $u \in \{BLANC, GRIS\}$, **init à** $BLANC$, $\forall u \in V$
$trouve, prochainVoisin$: booléen

```
1  Fonction parcoursProfondeur(G, s) :
2  début
3      couleur[s] ← GRIS
4      p.push(s)
5      tant que ¬p.empty() faire
6          u ← p.peek()
7          trouve ← faux
8          prochainVoisin ← vrai
9          tant que (¬trouve) ∧ (prochainVoisin = vrai) faire
10             si u a encore un voisin alors
11                 v ← prochain voisin de u
12                 si couleur[v] = BLANC alors
13                     couleur[v] ← GRIS
14                     p.push(v)
15                     trouve ← vrai
16             sinon prochainVoisin ← faux
17         si ¬trouve alors p.pop()
```

De même que l'algorithme 2, l'algorithme 4 n'est qu'une structure minimale qui doit être complétée pour réaliser un calcul de distance, une recherche, etc.

À noter également que les algorithmes donnés sont des algorithmes génériques qui ne tiennent pas compte des structures de mémorisation du graphe proposées en 1.2. Il faudra donc les adapter à la structure de données considérée pour les utiliser.

1.3.7 Calcul d'un chemin correspondant à un parcours

De la même manière que l'algorithme 2 a été utilisé pour calculer les liens *fils* $\rightarrow$ *père* d'un parcours en largeur, l'algorithme 4 peut-être utilisé pour calculer ce tableau. Grâce à l'arborescence stockée dans le tableau $pere[]$, il est alors possible de retrouver le chemin entre le sommet s, initiateur d'un parcours, et n'importe quel sommet v. Pour cela, il faut remonter les ancêtres de v jusqu'au sommet s en mémorisant les sommets traversés puis inverser cette suite de sommets pour qu'elle donne bien le chemin de s à v.

L'algorithme 5 permettant de calculer ce chemin vérifie d'abord qu'il y a bien un chemin (ligne 3) : si la valeur de $pere$ est $NULL$, c'est qu'il n'y a pas de chemin entre s et v. Si le chemin existe, l'algorithme remonte d'abord du sommet destination v au sommet initiateur s en empilant les sommets : les sommets parcourus sont empilés dans la boucle "tant que" (lignes 5 à 7). A la sortie de la boucle, on dépile les sommets pour générer le chemin correct en les ajoutant au chemin retourné (lignes 9 à 11).

Algorithme 5 : Chemin du parcours entre les sommets s et v.

```
   Données : p : pile de sommets
             l : chemin
 1 Fonction chemin(G, s, v, pere) :
 2 début
 3     si pere[v] = NULL alors afficher ("pas de chemin entre s et v")
 4     sinon
 5         tant que (v ≠ s) faire
 6             p.push(v)
 7             v = pere[v]
 8         l.add(v)
 9         tant que ¬p.empty() faire
10             v = p.pop()
11             l.add(v)
12     retourner l
```

1.4 Recherche des chemins d'un graphe

La recherche de chemins dans un graphe est un problème classique de l'algorithmique sur les graphes. Un grand nombre d'algorithmes existent avec des objectifs différents : recherche de chemins depuis un sommet vers tous les autres, recherche d'un plus court chemins entre deux sommets, etc. Le problème de la recherche d'un plus court chemin d'un sommet vers un autre sommet ou vers tous les autres sommets est résolu par un parcours en largeur du graphe comme nous l'avons vu. Dans le cas de la recherche vers un seul sommet, l'algorithme s'arrête dès que le sommet est trouvé.

Nous abordons ici la recherche des chemins existants dans un graphe à l'aide d'une matrice d'adjacence. Pour cela nous voyons d'abord quelques définitions.

Définition 1.22 : Fermeture transitive d'un sommet x

La fermeture transitive du sommet x, notée $\Gamma^*(x)$, est l'ensemble des sommets placés sur toutes les chaînes (ou chemins) issues de x. Il s'agit également de l'ensemble de ses descendants.

Formellement *: $\Gamma^*(x) = \{x\} \cup \Gamma(x) \cup \Gamma(\Gamma(x)) \ldots$ où $\Gamma(x)$ est l'ensemble des successeurs de x*

Même remarque pour $\Gamma^{*-1}(x)$ avec $\Gamma^{-1}(x)$ pour l'ensemble des ancêtres de x.

Définition 1.23 : Fermeture transitive du graphe G

la fermeture transitive d'un graphe est la contraction de toutes les relations du type $a\mathcal{R}b$ et $b\mathcal{R}c$ en $a\mathcal{R}c$. C'est-à-dire qu'un arc est ajouté entre deux sommets a et c s'il existe en chemin entre eux. Comme l'opération est répétée tant qu'il est possible d'ajouter des arcs, il y aura un arc entre toute paire de sommets reliés par un chemin. La fermeture transitive d'un graphe s'obtient donc donc en ajoutant des arcs entre tous les sommets ayant une chaîne (ou un chemin) entre eux au graphe initial.

Formellement *: le graphe $G^*(V, E^*)$ est la fermeture transitive du graphe $G(V, E)$ si E^* est minimum vérifiant :*

- $E \subset E^*$
- $(x, y) \in E^* \wedge (y, z) \in E^* \Rightarrow (x, z) \in E^*$ *transitivité*

L'exemple de la figure 1.11 illustre la construction de fermeture transitive pour un graphe orienté.

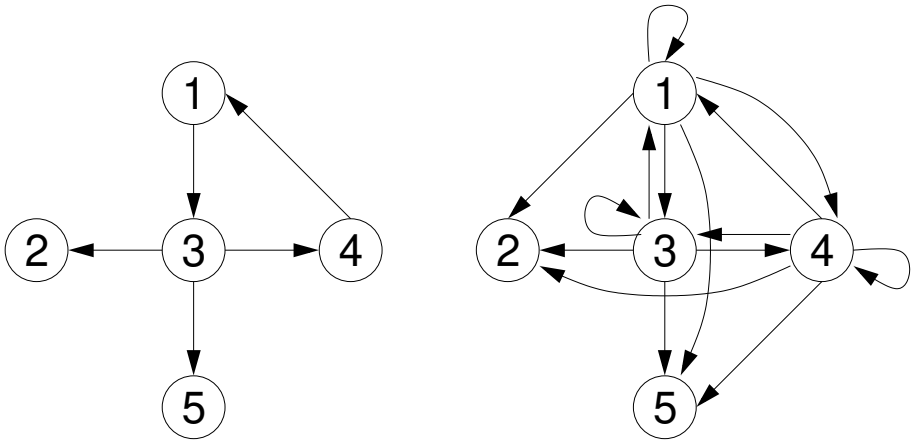


FIGURE 1.11 – Exemple de graphe G associé à sa fermeture transitive G^*

La fermeture transitive G^* d'un graphe G permet donc de savoir s'il existe un chemin d'un sommet x vers un sommet y dans G en examinant si le sommet x a comme voisin y dans G^*.

Si A est la matrice d'adjacence du graphe G, le calcul A^k, où A^k est le produit booléen de la matrice A k fois par elle-même, donne les chemins de longueur k. Si $m^k_{x,y} = 1$ signifie qu'il existe un chemin de longueur k qui relie x à y. Si $m^k_{x,y} = 0$ alors il n'existe pas de chemin de longueur exactement k.

A^*, la matrice d'adjacence du graphe G^* se calcule alors comme suit :

$$A^* = A + A^2 + A^3 + \ldots + A^{n-1}$$

où n est le nombre de sommets du graphe. Dans ce cas $m^*_{x,y} = 1$ signifie qu'il existe un chemin qui relie x à y et $m^*_{x,y} = 0$ sinon. La matrice A^* donne donc l'ensemble des chaînes (ou chemins) possibles dans le graphe.

La complexité associée à ce calcul est $O(n^4)$. Il est évidemment possible de faire mieux. Ce calcul restant cependant assez lourd, il n'est utile que lorsqu'on demande un grand nombre de fois si x est connecté à différents sommets de G.

1.5 Plus courts et plus longs chemins

Les algorithmes permettant la recherche de chemins dans un graphe que nous avons vu jusqu'à maintenant supposent que le calcul de la distance repose sur le nombre d'arcs parcourus.

La structure de graphe se prête cependant aussi à la représentation de relations non binaires, pour lesquelles ce qui nous intéresse n'est pas seulement de savoir s'il existe une relation mais aussi combien coûte le passage d'un sommet à un autre. Un exemple simple est la représentation des distances entre des villes. Ici les sommets sont les villes, les arêtes les routes qui les relient, pour lesquelles nous devons ajouter l'information de la distance. Dans ce cas les arcs peuvent éventuellement porter une *pondération*. Cette valeur peut servir, par exemple, à traduire le coût ou le temps de passage de l'état x à l'état y ou la distance entre les deux sommets x et y.

1.5.1 Pondérations

La *pondération* d'un arc (x, y) marque le coût de passage du sommet x au sommet y.

La figure 1.12 est un exemple de diagramme sagittal avec des pondérations notées $e1$ à $e4$.

Parmi les représentations de graphe sous forme de structure de données, les matrices d'adjacence offrent la possibilité de coder simplement des graphes pondérés en donnant la valeur associée à l'arête dans cet élément. Si la pondération d'une arête k reliant les sommets i et j est u_k alors $m_{i,j} = u_k$. De leur côté, les ensembles d'adjacence, qu'ils soient implémentés sous la forme de tableau ou de liste chaînées, doivent être complétés ou adaptés pour mémoriser les pondérations. Cela peut se faire soit en définissant une structure d'arc donnant le voisin et la pondération ou en dupliquant les structures de représentation, la seconde structure mémorisant alors la pondération plutôt que le voisin.

En plus de la recherche d'une plus courte distance, il est possible de calculer une plus longue distance à condition que le graphe ne présente pas de cycle. La recherche d'un

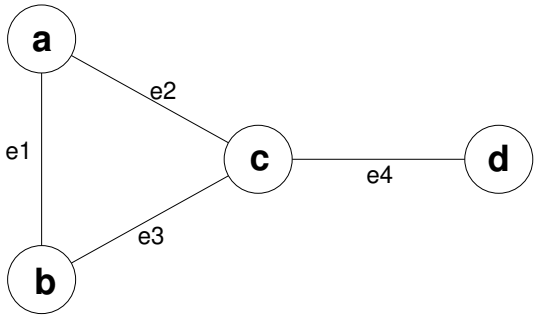


FIGURE 1.12 – Exemple de graphe avec des arêtes pondérées

plus court et long chemin est, par exemple, utile à la recherche des dates au plus tôt ou au plus tard dans un graphe d'ordonnancement (méthode PERT).

Les problèmes associés sont par conséquent les suivants :

- recherche d'un plus court ou plus long chemin d'un sommet s à un sommet v
- recherche de plus courts chemins de s vers les sommets atteignables depuis s
- calcul d'un distancier (plus court chemin entre tout couple du graphe).

1.5.2 Algorithme générique matriciel de Floyd-Warshall

Si le mode de stockage du graphe est une matrice d'adjacence A, les cases $A[s, v]$ de la matrice A donnent la distance entre deux sommets s et v connectés directement. Dans ce cas, on prend comme convention $A[s, v] = +\infty$ si les sommets s et v ne sont pas connectés. L'algorithme générique 6 matriciel de Floyd-Warshall permet alors de calculer toutes les distances entre chaque couple de sommets.

Algorithme 6 : Algorithme matriciel de Floyd-Warshall pour un graphe $G(V, E)$

Données : A : matrice d'adjacence du graphe G
Résultat : $PCC[x, y]$: le sommet qui précède y dans le plus court chemin de x à y

```
1 Fonction plusCourtChMatriciel(G)
2 début
3     pour chaque sommet z de G faire
4         pour chaque sommet x de G faire
5             pour chaque sommet y de G faire
6                 si ((A[x,z] ≠ +∞) ∧ (A[z,y] ≠ +∞)∧
7                 (A[x,z] + A[z,y]) < A[x,y])) alors
8                     A[x,y] ← A[x,z] + A[z,y]
9                     PCC[x,y] ← z
```

L'idée de l'algorithme est la suivante : pour tout couple (x, y) de G, on cherche, pour tout sommet intermédiaire z, si le chemin de x à y passant par z est un raccourci. Cet algorithme peut être utilisé également pour la recherche de plus long chemin en regardant si passer par le sommet z permet de faire un détour. L'algorithme 6 est une illustration de cette idée. $p[x, y]$ est le numéro du sommet prédécesseur de y sur le chemin de x à y,

initialisé à x.

À la fin de l'algorithme 6, il est possible de connaître la distance du plus court chemin de x à y, et grâce à $p[x, y]$ ce chemin peut être retrouvé.

La complexité de cet algorithme est $O(n^3)$, avec n le nombre de sommets du graphe, puisque nous avons trois boucles imbriquées qui parcourent n sommets.

1.5.3 Algorithme à fixation d'étiquettes de Dijkstra

L'algorithme de Dijkstra permet de calculer les plus courts chemins à origine unique, c'est-à-dire à partir d'un sommet x_0 donné et vers tous les autres sommets du graphe, dans un graphe simple orienté pondéré $G(V, E)$. Les poids des arêtes doivent être des valeurs positives ou nulles.

Principe de l'algorithme

- on recherche étape après étape un sommet y le plus proche de x_0 ;
- soit $pcd[x]$ la plus courte distance provisoire de x_0 à x pour tout sommet x ;
- soit $prec[x]$ le sommet précédent x sur le plus court chemin de x_0 à x ;
- soit $d(x, y)$ la longueur de l'arête (x, y) [10]
- soit X l'ensemble des sommets x ayant une valeur pour $pcd[x]$ définitive. Au départ, $X = \{x_0\}$;
- soit Y le complémentaire de X. On a en effet $V = X \cup Y$ et $X \cap Y = \emptyset$.
- on pose :
  - $\forall x \in V$et $x \neq x_0$, $l(x) = +\infty$ et $prec[x] = -1$
  - $l[x_0] = 0$
- on prend au départ : $X = \{x_0\}$ et $Y = V \setminus \{x_0\}$ avec $\forall y \in Y \cap \Gamma(x_0)$ on a $l(y) = d(x_0, y)$ et $prec(y) = x_0$

Compte tenu de ces éléments, l'idée de cet algorithme est de prendre parmi les sommets dont on connaît une distance provisoire à x_0, le sommet y qui a la distance provisoire la plus courte. On met alors à jour les distances provisoires des voisins de y en tenant compte de leur distance à y, qu'ils soient atteints ou non pour la première fois. Si un sommet voisin de y a sa distance provisoire à x_0 diminuée, cela signifie que le chemin qui passe par y est un raccourci par rapport au chemin qui permettait de l'atteindre précédemment. L'algorithme générique 7 illustre plus précisément cette idée.

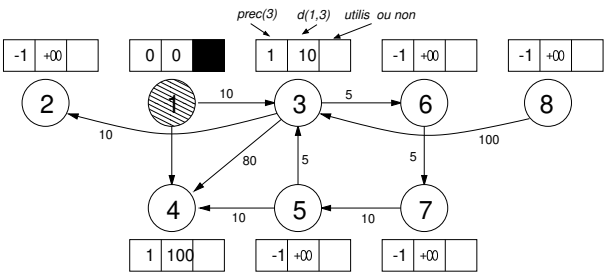
10. donnée ici sous la forme d'une fonction pour être générique mais pourrait être une matrice d'adjacence avec les pondérations

Algorithme 7 : Algorithme de Dijkstra, plus court chemin dans un graphe $G(V, E)$

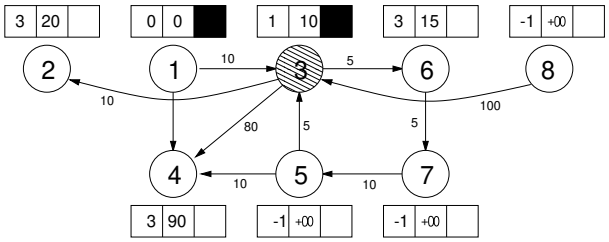
Données : $d(x, y)$: la longueur de l'arête (x, y)
Résultat : $pcd[x]$: la plus courte distance provisoire de x_0 à x $\forall x \in V$
$prec[x]$: le sommet précédent x sur le plus court chemin de x_0 à x
X : l'ensemble des sommets x ayant une valeur pour $l[x]$ définitive
Y : le complémentaire de X dans V $(V = X \cup Y)$

```
1  Fonction plusCourtChDijkstra(G, x_0)
2  début
3      pcd[x_0] ← 0
4      X ← {x_0}
5      Y = V \ {x_0}
6      pour chaque sommet x de G ≠ x_0 faire
7          pcd[x] ← +∞
8          prec[x] ← −1
9      pour chaque sommet y voisin de x_0 faire
10         pcd[y] ← d(x_0, y)
11         prec[y] ← x_0
12     tant que ∃y ∈ Y tel que pcd[y] < +∞ faire
13         prendre y ∈ Y tel que pcd[y] soit minimum
14         pour chaque z ∈ Y ∩ Γ(y) faire
15             si pcd(z) > pcd(y) + d(y, z) alors
16                 pcd[z] ← pcd[y] + d(y, z)
17                 prec[z] ← y
18         X ← X ∪ {y}
19         Y ← Y \ {y}
```

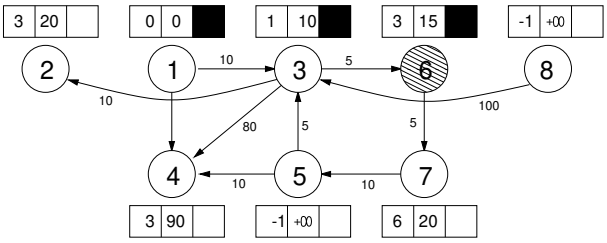
À la fin de l'algorithme, X contient tous les sommets x avec $pcd[x] < +\infty$ et Y contient tous les sommets inaccessibles depuis x_0. Il est possible de construire les plus courts chemins au départ de x_0 grâce à $prec[]$. La figure 1.13 montre un exemple d'étiquetage des sommets d'un graphe pour cet algorithme afin de connaître tous les plus courts chemins entre le sommet **1** et tous les sommets atteignables depuis cette origine.



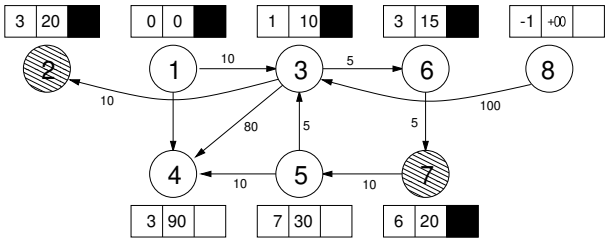
(a) Recherche du plus court chemin à partir de **1**



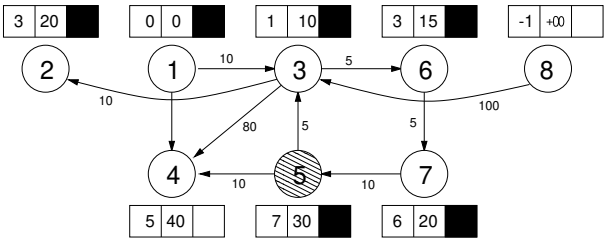
(b) **3** est le sommet le plus proche de **1**. On recalcule les plus courtes distances à **1**. On calcule les distances pour les voisins de **3**.



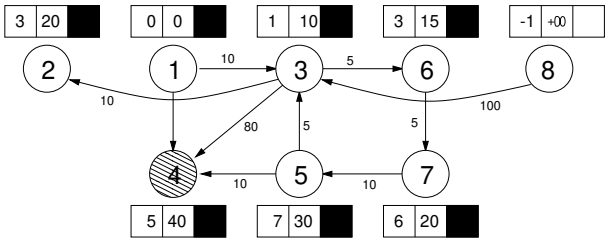
(c) **6** est le plus proche de **1** (non marqué). Modification des valeurs associées au sommet **7**.



(d) Les sommets **2** et **7** sont les plus proches (non marqués). Modification des valeurs du sommet **5**.



(e) **5** est le sommet le plus proche de **1** (non marqué). Le sommet **4** est maintenant à distance 40 au lieu de 90 du sommet **1**.



(f) **4** est le sommet le plus proche de **1** (non marqué). Il n'a pas de suivant et le sommet **8** n'est pas accessible. Fin de l'algorithme.

FIGURE 1.13 – Exemple de recherche des plus courts chemin dans un graphe $G(V, E)$ avec comme origine le sommet **1**

Nous voyons, dans le chapitre **??** sur la programmation dynamique, un autre algorithme permettant de calculer les plus courts chemins d'un sommet d'un graphe vers tous les autres sommets, l'algorithme de Bellmann-Ford. Cet algorithme permet de rechercher les plus courts, ou des plus longs chemins, à partir d'un sommet dans le cas d'arêtes de longueur et de signe quelconque, à condition qu'il ne contienne pas de circuit absorbant (circuit dont la somme des pondérations des arêtes est négative).

1.6 Connexité et forte connexité d'un graphe

La propriété de connexité sur les graphes traduit l'idée que le graphe peut-être composé de parties non reliées entre elles, donc de savoir si le graphe est en un seul "morceau" ou

non.

Définition 1.24 : Graphe connexe et composante connexe

Un graphe non orienté G est *connexe* s'il existe une *chaîne* entre toutes paires de sommets de G. Si G n'est pas connexe, alors il existe peut-être des sous-graphes de G connexes appelés *composantes connexes* de G. S'il existe p sous-graphes connexes alors p est le *nombre de connexité* de G. Un sommet isolé, sans arête incidente, est une composante connexe à lui seul.

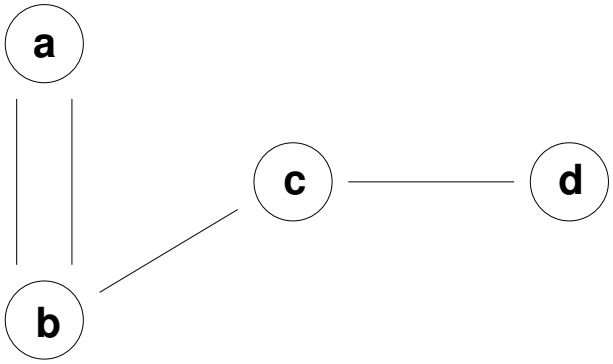


FIGURE 1.14 – Exemples de graphe connexe

Les propriétés de connexité sont utiles par exemple pour faire de la vérification de saisie de données dans les SIG (Systèmes d'Information Géographiques).

La propriété de connexité s'applique aussi aux graphes orientés et il est admis qu'un graphe orienté G est *connexe* si le graphe non-orienté associé est connexe. Pour exprimer le fait qu'il existe un chemin entre tous les sommets de G nous utilisons la propriété de connexité forte.

Définition 1.25 : Graphe fortement connexe et composantes fortement connexes

Un graphe orienté G est *fortement connexe* s'il existe un *chemin* entre toutes paires de sommets de G. Il est donc possible de relier deux sommets quelconques du graphe en suivant un chemin. Un sommet isolé est donc une composante fortement connexe à lui seul.

Si G n'est pas fortement connexe, alors il existe peut être des sous-graphes de G fortement connexes appelés *composantes fortement connexes* de G. Il existe donc un chemin entre deux sommets quelconques de la composante.

À noter que, si le graphe orienté est symétrique, alors il existe un arc (v, u) pour tout arc (u, v) et la notion de forte connexité se confond alors avec celle de connexité.

Définition 1.26 : Graphe k-connexe

Un graphe $G(V, E)$ est $k-$connexe s'il possède au moins k arêtes et qu'il existe au moins k chaînes disjointes entre toutes les paires de sommets de G.

La $k-$connexité d'un graphe implique que supprimer $k-1$ sommets ne déconnecte pas

le graphe, par contre, nous pouvons trouver un ensemble de k sommets qui déconnecte le graphe. Dans ce cas, la valeur k est minimale. Cet ensemble de k sommets n'est pas forcément unique mais pas non plus quelconque.

A noter que la définition, à travers l'utilisation de chaînes qui sont des suites de sommets, définit implicitement une sommet-connexité. On parle également de *k-arête-connexité* si le graphe est connexe et qu'il est possible de le déconnecter en supprimant k arêtes. k est minimal c'est-à-dire que la suppression de $k-1$ arêtes ne déconnecte pas le graphe. Dans le cas où un graphe est à la fois *k-arête-connexe* et *k-sommet-connexe*, on dit que le graphe est optimalement connecté. Les graphes complets sont par exemple des graphes optimalement connectés.

Les propriétés de k-connexité, sommet ou arête, permettent par exemple de vérifier la fragilité d'un réseau électrique, téléphonique ou de distribution d'eau. Pour réduire ces problèmes de fragilité, il est peut être imposé à un graphe une $k-$connexité. Par exemple un graphe $2-$connexe n'a pas de point d'articulation qui fragilise l'ensemble puisque sa disparition entraîne une coupure du graphe en deux composantes.

1.6.1 Algorithmes de calcul de composantes connexes

Pour un graphe G, un parcours en largeur ou en profondeur à partir d'un sommet s donne la composante connexe à laquelle appartient s. Il suffit de refaire la même chose à partir des sommets non visités pour isoler toutes les composantes connexes du graphe. L'algorithme générique 8 propose, pour un graphe orienté symétrique ou non orienté, la recherche de composantes connexes en parcourant l'ensemble des sommets pour garantir qu'aucune composante n'est oubliée.

Algorithme 8 : Numérotation en composantes connexes d'un graphe orienté symétrique ou non orienté $G(V, E)$

Données : $n \leftarrow 0$: numéro de la composante connexe courante
Résultat : $cc[u]$: table des numéros de composantes, **initialisée à** 0 $\forall u \in V$
1 **Fonction** *composanteConnexeNOriente*(G)
2 **début**
3 **pour chaque** $x \in G$ *tel que* $cc[x] == 0$ **faire**
4 $n \leftarrow n + 1$
5 parcours (largeur ou profondeur) des sommets y de G à partir de x en marquant les $cc[y]$ à n

La notion de connexité a été définie dans les graphes orientés en ne tenant pas compte de l'orientation des arcs. Il est alors possible de définir un algorithme identifiant les composantes connexes d'un graphe orienté. Dans ce cas, un parcours en largeur ou en profondeur donne une composante incomplète. Il faut alors relancer des parcours à partir des sommets non encore visités. Si l'un de ces parcours atteint un sommet y déjà visité, alors la composante connexe courante et la composante connexe de y représentent une seule et même composante connexe. Pour tenir compte de cette information, il faut maintenir une table des *correspondances* entre les numéros de composantes connexes identiques. Un parcours de ce tableau des correspondances est nécessaire en fin d'algorithme pour

unifier les numéros des correspondances. Une renumérotation de l'appartenance aux composantes connexes est ensuite possible. L'algorithme générique 9 propose un calcul de la numérotation en composantes connexes pour un graphe orienté.

Algorithme 9 : Numérotation en composantes connexes d'un graphe orienté $G(V, E)$

Données :
$n \leftarrow 0$: numéro de la composante connexe courante
$corr[u]$: correspondance entre composantes, **initialisée à** u $\forall u \in V$
Résultat :
$cc[u]$: numéros de composantes, **initialisée à** 0 $\forall u \in V$

```
1  Fonction composanteConnexeOriente(G)
2  début
3      pour chaque x ∈ V non encore visité faire // cc[x] ≠ 0
4          n ← n + 1
5          pour chaque sommet y du parcours en largeur issu de x faire
6              si cc[y] ≠ 0 alors
7                  corr[n] ← cc[y] // n > cc[y] par construction
                   // arrêt de l'exploration à partir de y
8              sinon cc[y] ← n
9      pour i de 1 à la taille de corresp faire corr[i] ← corr[corr[i]]
10     pour chaque sommet x de G faire cc[x] ← corr[cc[x]]
```

Le calcul de composantes fortement connexes suppose de parcourir de plusieurs fois les sommets et arêtes. Nous en donnons d'abord une version simple, avec une complexité élevée qui est améliorée par le second algorithme, l'algorithme de Tarjan.

1.6.2 Algorithme de calcul de composantes fortement connexes

Principe de l'algorithme : Pour trouver les composantes fortement connexes d'un graphe orienté $G(V, E)$, nous cherchons les sommets x appartenant à un parcours (largeur ou en profondeur) issu d'un sommet s de départ tel qu'il existe aussi un parcours issu de x passant par s. Cela revient à faire un premier parcours sur G pour trouver les descendants de s puis un parcours sur le graphe inverse G^{-1} pour trouver les ancêtres de s. Les sommets x qui se trouvent dans l'intersection des deux parcours, c'est-à-dire ceux qui sont à la fois ancêtres et descendants de s, sont bien tels qu'il existe un chemin de s vers x et un chemin de x vers s.

L'algorithme générique 10 est une illustration de l'algorithme décrit ici. Il donne le numéro 1 à la composante fortement connexe à laquelle appartient le sommet s de départ, puis 2 à la composante fortement connexe à laquelle appartient le prochain sommet s non encore visité, etc.

Algorithme 10 : Numérotation en composantes fortement connexes d'un graphe

Données : $ncfc$: nombre de composantes fortement connexes, **initialisé à** 0
Résultat : $cfc[u]$: tableau d'appartenance à une composantes fortement connexes, **initialisé à** 0 $\forall u \in V$

```
1 Fonction composantesFConnexes(G)
2 début
3     calcul de G^{-1}
4     pour chaque s ∈ V tel que cfc[s] = 0 faire
5         parcours ← sommets du parcours de G à partir s
6         parcours^{-1} ← sommets du parcours de G^{-1} à partir s
7         ncfc ← ncfc + 1
8         pour chaque x ∈ parcour ∩ parcours^{-1} faire  cfc[x] ← ncfc
```

Nous cherchons maintenant à évaluer la complexité de cet algorithme. La recherche des sommets accessibles à partir de s (descendants de s) coûte $0(n+m)$ (m le nombre d'arêtes et n le nombre de sommets du graphe) par un parcours en largeur ou un parcours en profondeur. Le nombre de sommets étant au plus proportionnel au nombre d'arêtes, on peut simplifier cette complexité en comptant $O(m)$ pour ce calcul. La recherche des ancêtres de s revient à un parcours en largeur ou en profondeur du graphe inverse G^{-1} avec une complexité également de $O(m)$. Le coût du calcul de G^{-1} est de $O(m)$. Le calcul des sommets ancêtres et descendants d'un sommet s de G a donc une complexité de $O(m)$. Puisque le graphe contient n sommets, la complexité de notre algorithme de numérotation en composantes fortement connexes est $O(nm)$.

1.6.3 Algorithme de Tarjan

Tarjan a proposé un algorithme permettant d'améliorer la complexité de l'algorithme simple présenté précédemment. Il est basé sur un parcours en profondeur.

L'algorithme repose sur les principes suivants :

- L'algorithme cherche le point d'entrée de la composante fortement connexe courante. Ce sommet est le plus vieil ancêtre commun à tous les membres de la même composante fortement connexe.
- Il utilise 2 piles : p où il empile les sommets de la composante fortement connexe potentielle courante et q qui sert à gérer le parcours en profondeur.
- Il utilise 2 tableaux : $pPetit[x]$ qui enregistre le plus petit numéro de visite des sommets déjà rencontrés dans la composante et $nv[x]$ qui enregistre le numéro du sommet dans le sens du parcours en profondeur.

Il procède de la manière suivante :

1. Il prend un premier sommet quelconque, auquel il attribue un numéro de visite et met dans les piles p et q. Il va ensuite parcourir le graphe en profondeur à l'aide de la pile q. Il regarde donc chaque voisin y du sommet courant x
2. Si y est un sommet du graphe atteint pour la première fois : l'algorithme lui attribue un numéro de visite dans les tableaux nv et $pPetit$ (ligne 24 et ligne 25).

Puis il le met dans les piles p et q ($lignes26et27$), x est potentiellement dans la composante fortement connexe courante.

3. Si y est un voisin de x tel que (x, y) est un arc arrière (ligne 28) avec y dans la composante fortement connexe courante ($y \in p$) alors l'algorithme diminue sa valeur de $pPetit$, ce qui sert par la suite pour identifier le sommet d'entrée dans la composante connexe (ligne 11).

4. Si tous les voisins de x ont été visité, l'algorithme remonte de son parcours en profondeur (ligne 17), mais avant il doit vérifier qu'il ne sort pas d'une composante fortement connexe (ligne 11). Si c'est le cas l'algorithme dépile les sommets de p jusqu'à x inclus et numérote les sommets dépilés avec le numéro de la composante fortement connexe courante (ligne 13 à 16). Pour finir l'algorithme propage dans le sens inverse du parcours la plus petit valeur de visite de la composante fortement connexe (lignes 18-19).

Exemple La figure 1.15 est un exemple de numérotation en composantes fortement connexes par l'algorithme de Tarjan.

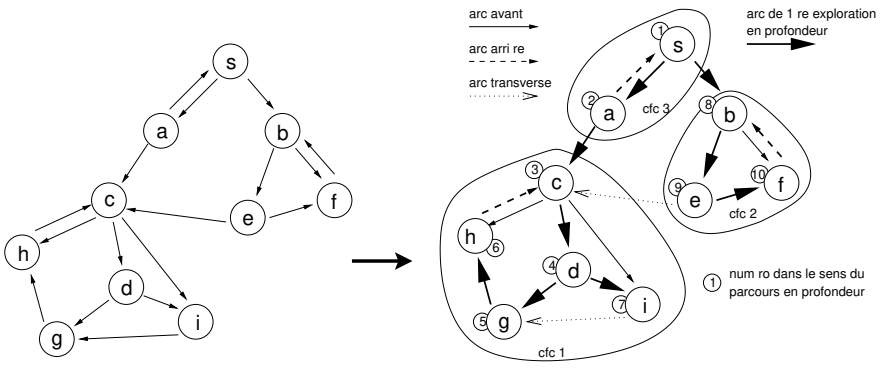


FIGURE 1.15 – Numérotation en composantes fortement connexes d'un graphe $G(V, E)$ par l'algorithme de Tarjan

Algorithme 11 : Algorithme de Tarjan

Données : p, q : piles, **initialisé à** $pileVide()$;
$ncfc$, npp : entiers, **initialisé à** 0 ;
$nv[s]$: tableau des numéros de visite, **initialisé à** 0 $\forall s \in G$;
$pPetit$:
Résultat : $cfc[1..n]$: le tableau des sommets donnant la composante d'appartenance

```
1  début
2      pour chaque s ∈ G tel que nv[s] = 0 faire
3          npp++ ;
4          nv[s] ← npp;
5          pPetit[s] ← npp;
6          p.push(s) /*          1er elt de la CC courante pour la num en CFC        */
7          q.push(s) /* 1er sommet dont il faut visiter les voisins                 */
8          tant que ¬q.emty() faire
9              x ← q.peek() ;
10             si x n'a plus de voisin à découvrir alors
11                 si pPetit[x] = nv[x] alors
                       /* x est l'entrée de la cfc courante                        */
12                     ncfc++ /*       nouvelle composante fortement connexe       */
13                     répéter
14                         y ← p.pop() ;
15                         cfc[y] ← ncfc
16                     jusqu'à y = x;
17                 x ← q.pop() ;
18                 si ¬q.empty() alors
                       /* propagation du num du plus vieil ancêtre commun          */
19                     pPetit[q.peek()] ← min(pPetit[p.peek()], pPetit[x])
20             sinon
21                 y ← voisinSuivant[x] /* 1er voisin non visité depuis x           */
22                 si nv[y] = 0 alors
                       /*                1ère rencontre avec ce sommet             */
23                     npp++ ;
24                     nv[y] ← npp;
25                     pPetit(y) ← npp /*  car il n'y a pas forcément de retour vers la
                           cfc de x                                                 */
26                     p.push(y) /*            y est dans la cfc courante          */
27                     q.push(y) /*       il faudra parcourir les voisins de y     */
28                 sinon
29                     si (nv[y] < nv[x]) ∧ (y ∈ p) alors
                           /* y visité avant x et ∈ à la cfc courante.(x, y) est un arc
                              transverse ou arrière dans la même cfc                */
30                         pPetit[x] ← min(pPetit[x], nv[y])
```

1.7 Arbres

La notion d'arbre est très largement utilisée dans le domaine informatique, par exemple parce qu'elle illustre bien la notion de choix et donc sert de support à l'exploration ou l'évaluation de solutions d'un problème.

Définition 1.27 : Arbre

Un arbre est un graphe non-orienté, connexe [11] et acyclique.

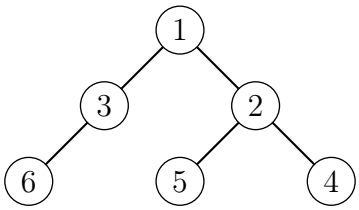


FIGURE 1.16 – Exemple d'arbre

La figure 1.16 donne un exemple d'arbre composé de 6 sommets. A noter que, d'après la définition, les sommets ne sont pas différenciés et qu'il n'y a donc pas de racine mais qu'un point isolé peut-être considéré comme un arbre. Les sommets n'ayant pas plus d'un voisin sont couramment appelés feuilles.

Les propriétés des arbres sont telles que, pour un graphe non orienté $G(V, E)$, les affirmations suivantes sont équivalentes :

- G est un arbre
- deux sommets quelconques de G sont reliés par une chaîne élémentaire unique
- G est connexe, mais si l'on enlève une arête quelconque de E, le graphe résultant n'est plus connexe
- G est connexe et son nombre d'arêtes est égal au nombre de sommets moins 1
- G est acyclique et son nombre d'arêtes est égal au nombre de sommets moins 1
- G est acyclique mais si une arête quelconque est ajoutée à E, le graphe résultant contient un cycle.

La preuve de l'équivalence de ces propriétés est traitée dans la partie exercice.

Définition 1.28 : Forêt

Une forêt est un graphe non orienté acyclique.

Une forêt est donc composée d'arbres. La figure 1.17 donne un exemple de composé de 6 sommets et 2 arbres.

Définition 1.29 : Arbre couvrant

Un arbre couvrant (ou recouvrant) d'un graphe non orienté $G(V, E)$ est un sous-graphe de G $G'(V, E')$ acyclique qui connecte tous les sommets de G.

Dans la figure 1.18 l'arbre avec les arêtes rouges est un arbre couvrant du graphe global.

11. Voir la définition en 1.6.

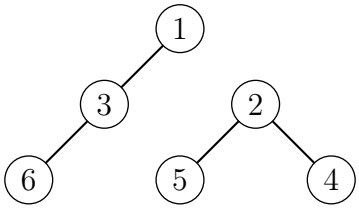


FIGURE 1.17 – Exemple de forêt

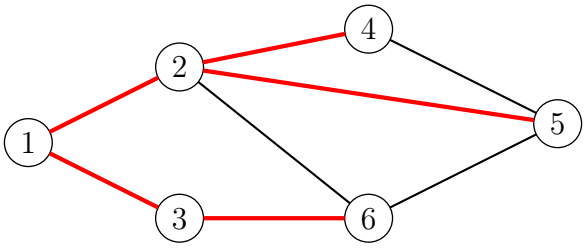


FIGURE 1.18 – Exemple d'arbre couvrant

Puisque qu'un arbre est un graphe particulier, ses arêtes peuvent porter une pondération. Trouver un arbre couvrant de poids minimal est un problème classique de l'algorithmique sur les graphes qui sera abordé dans la suite du cours.

Définition 1.30 : Arborescence

Une arborescence (ou d'arbre enraciné [12]) est un arbre dont on distingue un sommet particulier, la racine.

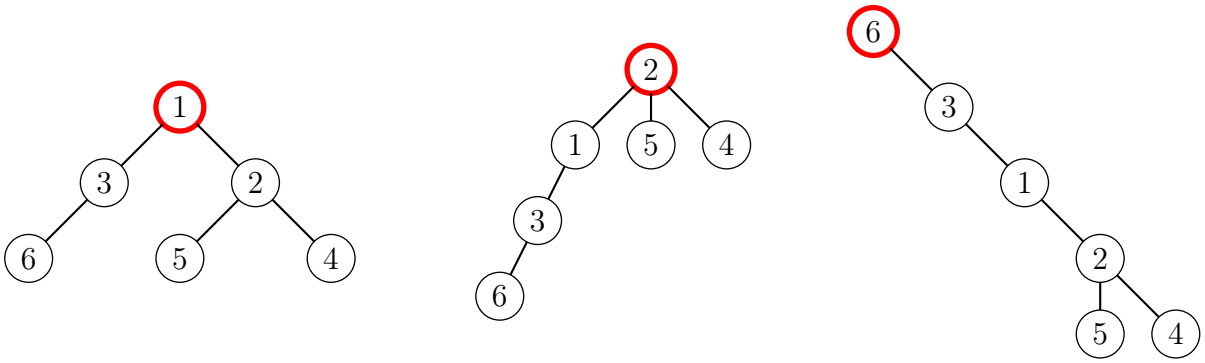


FIGURE 1.19 – Exemples d'arborescences

La figure 1.19 montre différente arborescences possibles à partir d'un même arbre où les racines sont données en rouge. Comme nous pouvons le voir n'importe quel sommet d'un arbre peut être choisi pour devenir la racine d'une arborescence. A noter que les arborescences sont des arbres donc qu'il existe une chaîne unique qui mène de la racine à un sommet donné. Cette chaîne est couramment appelée branche. Une sous-chaîne d'une branche est appelée branche ou sous-branche.

La définition d'une racine permet de :

12. D'après wikipedia. Attention toutefois à l'appellation exacte car le terme arbre est parfois utilisé pour un arbre enraciné pour des questions de simplicité

— introduire la notion de sommet père et de sommet fils pour les extrémités d'une arête : le sommet père est celui qui est le plus proche de la racine. Nous pouvons alors noter qu'un père peut avoir plusieurs fils mais chaque fils n'a qu'un père.
— définir la profondeur (au hauteur) d'un sommet comme la longueur de la chaîne entre la racine et ce sommet et la profondeur d'une arborescence comme la plus grande profondeur de ses sommets.

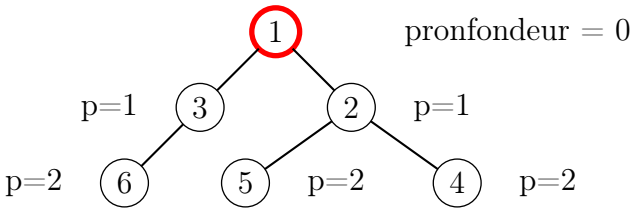


FIGURE 1.20 – Profondeurs dans une arborescence

La figure 1.20 donne les profondeurs dans l'arborescence de profondeur 2.

A titre d'exemple, la recherche de la sortie d'un labyrinthe peut-être représentée sous la forme d'une arborescence où la racine est le point de départ, les sommets les différentes positions atteintes et les branches les chemins parcourus. Les systèmes de fichiers sont également des arborescences, dotées d'une racine, et d'un chemin unique vers chaque fichier.

1.8 Recherche d'un arbre couvrant de poids minimal

Soit $G(V, E)$ un graphe. Ce problème de la recherche d'un arbre couvrant de poids minimal (ARPM) pour le graphe G consiste à trouver un sous-ensemble $T \subseteq E$ d'arêtes de G, acyclique, qui connecte tous les sommets de V et dont le poids total $w(T)$ soit minimal, avec : $w(T) = \sum_{(u,v)\in T} w((u, v))$

Résoudre ce problème est utile, par exemple, pour construire des lignes électriques pour relier plusieurs sites, ou construire un schéma électrique reliant les broches d'un circuit électronique.

Bien sûr, trouver une solution à cette recherche pourrait être réalisé en générant puis en évaluant tous les arbres couvrants possibles. Dans le problème de l'ARPM, les conditions sont cependant réunies pour qu'une stratégie gloutonne, c'est-à-dire une stratégie qui à chaque étape fait le choix le moins coûteux, garantisse un résultat optimal. Les algorithmes de Kruskal et de Prim permettent de résoudre ce problème de manière optimale. Ces algorithmes sont étudiés dans le chapitre sur la programmation gloutonne.

1.9 Caractérisation des graphes

Dans cette partie nous définissons les propriétés et caractéristiques qui sont attribuées aux graphes.

Définition 1.31 : Graphe complémentaire

Soit le graphe $G(V, E)$, le graphe complémentaire de G, noté $\bar{G} = (V, \bar{E})$, est le graphe sur le même ensemble de sommets dans lequel deux sommets sont reliés si et seulement s'ils ne sont pas adjacents dans G. Dans le cas des graphes non-orientés, $\bar{E} = \mathcal{P}_2(V) \setminus E$ où $\mathcal{P}_2(V)$ est l'ensemble des paires de V. Dans le cas des graphes orientés, $\bar{E} = V \times V \setminus E$.

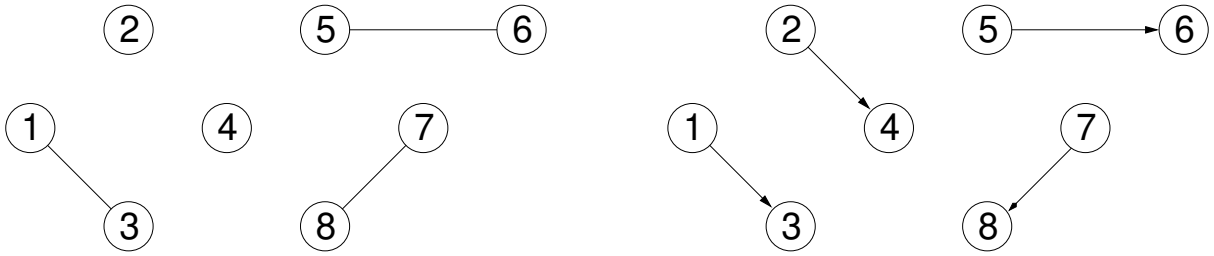
Définition 1.32 : Graphe inverse

Soit $G(V, E)$ un graphe orienté. Le graphe inverse de G, noté $\tilde{G} = (V, \tilde{E})$ ou G^{-1}, est le graphe sur le même ensemble de sommets dans lequel deux sommets x et y sont reliés par l'arc (x, y) si et seulement si x est le successeur de y dans G.

Le graphe inverse a donc ses arcs orientés dans le sens opposé des arcs du graphe initial.

Définition 1.33 : Couplage d'un graphe

Le couplage d'un graphe $G(V, E)$ est un ensemble d'arêtes E' qui n'ont pas d'extrémité commune.



(a) Couplage du graphe de la figure 1.8a (b) Couplage du graphe de la figure 1.8a

FIGURE 1.21 – Exemples de couplages

Les figures 1.21b et 1.21a donnent des exemples de couplages pour le graphe donné à la figure 1.8a. Il est possible de le constater que plusieurs couplages peuvent être définis à partir d'un graphe et qu'une arête peut appartenir à plusieurs couplages.

Définition 1.34 : Union (resp. intersection) de graphes

L'union de deux graphes $G(V, E)$ et $H(W, F)$ (resp. l'intersection), notée $G \cup H$ (resp. $G \cap H$), est le graphe dont l'ensemble des sommets est $V \cup W$ (resp. $V \cap W$) et l'ensemble des arêtes est $E \cup F$ (resp. $E \cap F$).

Définition 1.35 : Graphe complet

Un graphe non orienté $G(V, E)$ est dit *complet* s'il existe en tous les sommets une arête vers chacun des autres sommets : $\forall x \in V, \forall y \in V | y \neq x, \{x, y\} \in E$. Il est noté K_n. On l'appelle aussi une *clique*, mais plus généralement lorsqu'il est un sous-graphe complet d'un graphe quelconque.

Un graphe non-orienté complet à n sommets est un graphe qui contient $\frac{n(n-1)}{2}$ arêtes. Il a un diamètre de 1 puisque chacun des sommets est à distance 1 des autres sommets.

Définition 1.36 : Coupe dans un graphe

Une coupe, dans un graphe connexe, est une partition [13] de l'ensemble des sommets V en deux sous-ensembles (classes) V_1 et V_2 telle qu'il n'y ait plus d'arête entre un sommet de V_1 et un sommet de V_2. Le graphe est ainsi "coupé"en deux. Une coupe peut donc être définie par un ensemble d'arêtes, ayant une de leur extrémité en V_1 et l'autre en V_2, et qui, si elles sont supprimées de l'ensemble E, définissent une partition des sommets telle qu'il n'y ait pas d'arêtes entre les deux sous-ensembles. Dans ce cas, les arêtes restantes ont leurs deux extrémités dans un même sous-ensemble.

Définition 1.37 : Graphe biparti

Un graphe $G(V, E)$ est *biparti* s'il existe une coupe telle que tout élément de E a une extrémité dans V_1 et l'autre dans V_2.

À noter que, dans ce cas, toutes les arêtes du graphe font partie de la coupe.

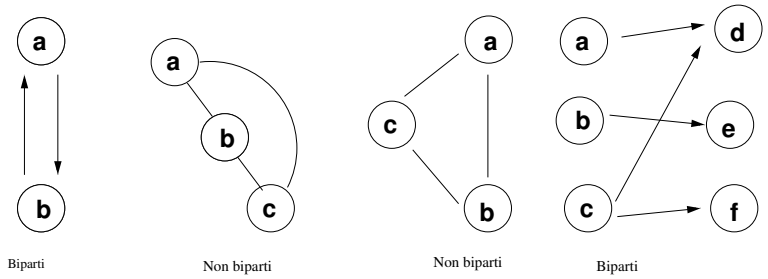


FIGURE 1.22 – Exemple de graphes bipartis et non-bipartis

La figure 1.22 donne des exemples de graphes bipartis et de graphes non bipartis.

Pour prouver qu'un graphe $G(V, E)$ est biparti, il est possible de montrer explicitement l'appartenance des sommets à l'un des ensembles V_1 ou V_2, et que chacun des arcs a une extrémité dans V_1 et l'autre dans V_2. Cela peut aussi être fait plus simplement, pour les graphes non orientés, en s'appuyant sur la proposition suivante :

Proposition 1.2 *Un graphe G est biparti si et seulement s'il n'admet pas de circuit de longueur impaire.*

La preuve de cette proposition est donnée en exercice dans la feuille d'exercice associée à ce chapitre.

Définition 1.38 : Graphe biparti complet

Un graphe biparti G est *complet* si tout sommet de V_1 est *adjacent* à tout sommet de V_2 et réciproquement. Si $|V_1| = p$ et $|V_2| = q$, on note $K_{p,q}$ ce graphe biparti complet.

Un graphe $K_{p,q}$ admet $p+q$ sommets, $p \times q$ arêtes et le degré des sommets de V_1 (resp. V_2) est de q (resp. p). Enfin le diamètre de $K_{p,q}$ est $D(K_{p,q}) = 2$ si $(p+q \geq 3)$ et $D(K_{p,q}) = 1$ si $(p = q = 1)$.

13. Nous rappelons qu'une partition d'un ensemble X est un ensemble de parties non vides de X deux à deux disjointes et qui recouvrent X.

Pour illustrer cette partie sur la définition des graphes et leur propriétés, nous donnons, à titre d'exemple, quelques graphes remarquables en annexe.

1.10 Synthèse

À l'issue de ce chapitre, et après avoir traité les exercices qui vont avec, vous devez être capables de :

- Maîtriser les concepts de base et le vocabulaire sur les graphes ;
- Modéliser un problème sous la forme d'un graphe ;
- Choisir une représentation adaptée pour mémoriser un graphe, en fonction d'un algorithme ;
- Utiliser les principaux algorithmes sur les graphes : parcours, recherche de chemin, etc. ;
- Écrire un algorithme spécifique utilisant une structure de graphe.

De nombreux autres algorithmes pourraient être encore traités, comme les algorithmes d'ordonnancement, le couplage de sommets, la recherche de flux maximal, etc. Tous ces algorithmes classiques sont abordés dans une littérature abondante sur le sujet. Bien que l'intérêt est réel de traiter de manière plus complète ces algorithmes, cela dépasse le cadre de cette partie du cours.

Introduction à l'optimisation

Nous avons rapidement évoqué le problème de l'optimisation dans l'introduction. Wikipédia définit l'optimisation comme : "L'optimisation est une branche des mathématiques cherchant à modéliser, à analyser et à résoudre analytiquement ou numériquement les problèmes qui consistent à minimiser ou maximiser une fonction sur un ensemble [14]". Transposée à l'informatique cette définition pourrait être exprimée comme la modélisation et la résolution de problèmes de minimisation ou de maximisation d'une fonction sur des données. Par exemple, la recherche d'un plus court chemin dans un graphe est un problème d'optimisation où la fonction objectif est la longueur du chemin et les données la structure de graphe.

Les chapitres suivants abordent la résolution de certains problèmes combinatoires. Les problèmes combinatoires sont des problèmes dont les solutions sont discrètes, indépendantes les unes des autres, contrairement aux problèmes continus. Par exemple, la recherche d'un plus court chemin entre deux sommets dans un graphe est un problème combinatoire car il constite, à la base [15], à comparer toutes les combinaisons possibles de sommets, donc tous les chemins possibles, qui relient les deux sommets pour trouver le chemin de taille minimale.

L'ensemble des solutions d'un problème combinatoire est souvent un ensemble fini. Par exemple, il y a un nombre fini de chemins élémentaires qui relient deux sommets. Si cet ensemble est fini le nombre des solutions possibles peut cependant être si grand qu'il ne permet pas de calculer toutes ces solutions quand la taille des données augmente. Par exemple, dans un arbre binaire il existe de l'ordre de 2^n chemins entre les feuilles et la racine, où n est la hauteur de l'arbre. Ce nombre augmentant exponentiellement il ne devient plus envisageable de calculer tous les chemins au delà d'une certaines valeur de n. Mais l'ensemble des solutions peut aussi être infini, par exemple si nous incluons, à la recherche de chemin, les chemins avec boucle dans un graphe possédant un ou des cycles.

La résolution de problèmes combinatoire est souvent liée au besoin de trouver une solution optimale. En effet, trouver une solution optimale à un problème revient souvent à énumérer les solutions possibles, donc les combinaisons possibles. Comme nous l'avons vu, il n'est pas toujours possible d'énumérer toutes les solutions. L'objectif de l'optimisation peut alors être de trouver la meilleure solution possible pour un problème ou de s'approcher de celle-ci s'il n'est pas possible de la trouver en trouvant une approche permettant de réduire le nombre de solutions à explorer, mais ce n'est pas toujours possible.

14. https ://fr.wikipedia.org/wiki/Optimisation_(math%C3%A9matiques)
15. Il est bien sûr aussi possible de trouver un algorithme qui calcule le meilleur chemin et de démontrer que le résultat obtenu est bien optimal mais cela ne change pas la nature du problème.

À noter que la programmation linéaire, que nous voyons au chapitre suivant, ne fait en général pas partie de l'optimisation discrète (donc de la combinatoire) si les valeurs des solutions sont des valeurs réelles car l'ensemble des solutions est continu. Par contre la programmation linéaire en nombres entiers, qui ne considère que des solutions avec des valeurs entières, fait, elle, partie de l'optimisation discète car, en bornant les valeurs pouvant être attribuées aux variables, l'ensemble des solutions possibles devient un ensemble discret. Ce problème est NP-Complet et ne permet donc pas, à priori, de trouver la solution dans un temps raisonnable. Nous nous contentons d'aborder ici le problème à variables réeles dont la résolution est polynomiale.

Chapitre 2

Programmation Linéaire

La programmation linéaire a pour but de résoudre les problèmes d'optimisation avec *maximisation* ou *minimisation* d'une fonction objectif en respectant un certain nombre de contraintes. Les paramètres du problème sont pour cela définis sous la forme de variables, les contraintes du problème sous la forme d'inéquations et l'objectif à optimiser doit être exprimé sous la forme une fonction linéaire (une somme pondérées des variables). Le résultat attendu est la valeur des paramètres permettant de maximiser ou minimiser l'objectif. C'est cet ensemble qui porte le nom de programme linéaire.

2.1 Exemples de problème linéaire

Enfin de comprendre ce qu'est un problème d'optimisation linéaire nous commençons par un exemple.

Nutriment	A	B
Courgette	2 l/m^2	1 l/m^2
Navet	1 l/m^2	2 l/m^2
Quantité disponible	8 l	7 l

TABLE 2.1 – Quantité nécessaire et stock disponible de nutriment

Le problème traité vise à maximiser la production de courgettes et de navets[1]. Pour que ces légumes se développent correctement, il est nécessaire de leur apporter deux types de nutriments A et B. La quantité de nutriments nécessaire diffère en fonction du légume et nous n'en possédons qu'une quantité limitée. Le tableau 2.1 donne la quantité nécessaire en l/m^2 à apporter pour chaque légume et le stock disponible. Avec ces apports le rendement est de 4 kg/m^2 pour les courgettes et de 5 kg/m^2 pour les navets. Nous cherchons à obtenir le plus grand poids de légumes possible et pour cela nous devons déterminer combien de m^2 il faut consacrer à chacun.

Les navets étant ce qui produit le plus grand poids au m^2, nous pourrions être tentés de produire le maximum de navets. Avec les 7 litres de nutriment de type B, nous pouvons

1. source :https://moodle.caseine.org, cours de recherche opérationnelle - Nadia Brauner

ainsi cultiver une surface de 3,5 m^2 pour les navets pour produire 17.5kg de navets. Comme cette production consomme tout le nutriment de type B, il n'est pas possible de cultiver en plus des courgettes. Si nous cultivons le maximum de courgettes, puisque nous avons plus de nutriment de type A, il n'est possible de produire que 16 kg de légumes, ce qui n'est pas mieux. La question qui se pose alors est de savoir si un mixte courgettes-navets ne permettrait pas de produire encore plus de légumes. Ainsi, si réduisons la surface de navets à 3 m^2, il possible, avec le nutriment de type B non utilisé, de cultiver 1 m^2 de courgettes qui donne 4 kg de légumes. La production totale atteint alors 18 kg, ce qui est mieux que de ne cultiver que des navets.

La solution optimale combine donc une surface de courgette avec une surface de navets mais comment trouver ces surfaces ? Une première solution est de tester, de manière combinatoire, les valeurs possibles pour la surface à consacrer à chacun des légumes et de voir si nous trouvons mieux. Comme ces valeurs ne sont pas entières cela peut représenter un nombre important de calculs.

C'est ici que la programmation linéaire apporte une solution. Pour cela, il faut d'abord modéliser le problème : identifier les variables puis définir la fonction à maximiser et les contraintes. Une fois cette modélisation faite, nous appliquons une méthode de résolution, soit graphique, soit basée sur un algorithme. Nous donnons dans la suite la modélisation du problème de la production de légumes et sa résolution avec ces deux méthodes.

2.1.1 Modélisation

La première chose à faire, pour la résolution d'un problème d'optimisation par programmation linéaire, est de trouver les paramètres du problèmes, appelés variables. Ici, nous cherchons la surface de terrain consacrée respectivement aux courgettes et aux navets puisque le poids final, que nous voulons optimiser, en dépend. Nous posons :

- x_1, la surface consacrée aux courgettes
- x_2, la surface consacrée aux navets

Ensuite il faut définir la fonction à optimisé. Ici c'est le poids total de légumes, notons le z, qui doit être maximisé. Ce poids dépend des deux surfaces x_1 et x_2. Puisque chaque m^2 de surface consacrée aux courgettes produit 4 kg et chaque m^2 de surface consacrée aux navets produit 5 kg, nous avons $z = 4x_1 + 5x_2$.

Pour finir nous définissons les contraintes. Puisque nous avons un stock limité de nutriment et que, pour pousser, les légumes ont besoin de ces nutriments, nous sommes limités dans le nombre de m^2 que nous pouvons consacrer à chaque légume. Nous avons 8 litres de nutriment A. Comme il faut 2 litres de ce nutriment pour chaque m^2 de courgettes et 1 litre pour chaque m^2 de navets, nous obtenons la contraintes suivante : $2x_1 + x_2 \leq 8$. Pour le nutriment B, en appliquant le même raisonnement nous obtenons la contrainte : $x_1 + 2x_2 \leq 7$. Nous savons également que les surfaces sont positives. Nous ajoutons donc les contraintes de positivité : $x_1 \geq 0$ et $x_2 \geq 0$.

Le tout se résume dans le programme linéaire suivant :

$$\begin{cases} \text{maximiser} & z & = & 4x_1 & + & 5x_2 & & \\ \text{sous les contraintes :} & & & & & & & \\ \textit{nutriment } A & 2x_1 & + & x_2 & \leq & 8 & & \\ \textit{nutriment } B & x_1 & + & 2x_2 & \leq & 7 & & \\ \textit{positivité} & x_1 & \geq & 0 & \text{et} & x_2 & \geq & 0 \end{cases}$$

Un programme linéaire est donc caractérisé par des variables réelles, un objectif et des contraintes linéaires, c'est à dire une somme de variables avec des coefficients.

2.1.2 Résolution

Si nous considérons le problème de manière générale, avec un nombre quelconque de variables et de contraintes, il n'est pas toujours possible de trouver une solution optimale. Mais certains cas sont plus simples que les autres. L'idée sous-jacente aux méthodes de résolution que nous allons voir est que chacune des contraintes délimite une zone de l'espace des solutions dans laquelle la contrainte est vérifiée. L'intersection des zones délimitées par les contraintes donne l'espace des solutions possibles. Il faut ensuite chercher dans cet espace la meilleure solution. En deux dimensions, cette méthode peut être réalisée à partir de tracés de droites dans le plan (l'espace des solutions). Nous parlons alors de méthode graphique.

Méthode graphique

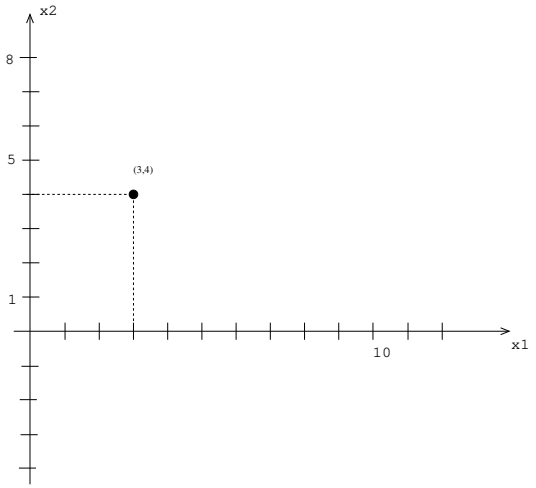


FIGURE 2.1 – Espace des solutions avec le point (3,4)

Pour notre problème exemple nous pouvons trouver une solution graphique puisque nous n'avons que deux variables. Les variables x_1 et x_2 définissent ainsi les deux dimensions du plan des solutions : chaque point du plan correspond à une valeur de x_1 et une valeur x_2. Par exemple, sur la figure 2.1, le point $(x_1 = 3, x_2 = 4)$ est une solution possible mais nous n'avons pas vérifié qu'il respecte bien les contraintes. Nous cherchons donc dans ce

plan le couple de valeurs qui respecte les contraintes et qui nous permet de produire le plus grand poids de légumes.

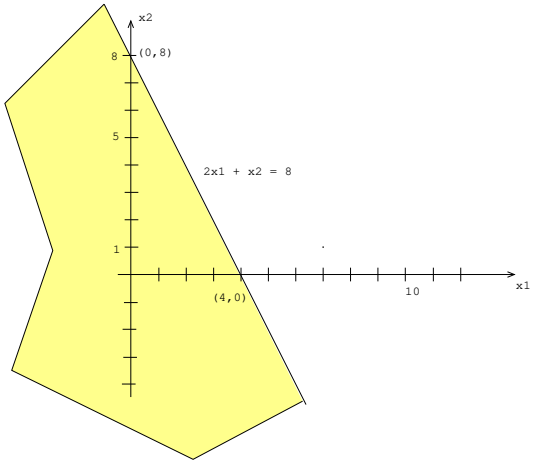
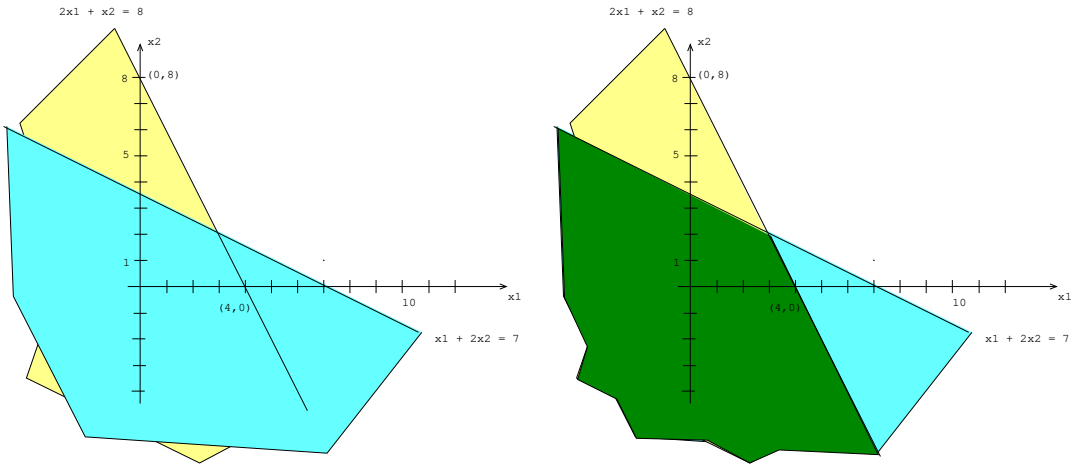


FIGURE 2.2 – Partage du plan par une contrainte

Nous commençons pas identifier la zone dans laquelle nous devons chercher les solutions. Nous pouvons voir qu'une contrainte divise ce plan en deux zones : la zone où la contrainte est vraie et celle où elle est fausse. La séparation entre les deux zones est matérialisée par la droite sur laquelle la contrainte est à l'égalité. Par exemple, si nous prenons la contrainte sur le nutriment A, la droite $2x_1 + x_2 = 8$ (qui passe par les point ($x_1 = 4$,$x_2 = 0$) et ($x_1 = 0$,$x_2 = 8$)) définit cette limite. Comme nous pouvons le voir sur la figure 2.2, tous les points en dessous de la droite (la partie jaune) vérifient la contrainte (par exemple, pour le point ($x_1 = 0$, $x_2 = 0$), nous avons $2x_1 + x_2 = 0$ ce qui est bien ≤ 8) et tous les points au dessus de la droite ne la vérifie pas (par exemple, pour le point ($x_1 = 4$, $x_2 = 2$) nous avons $2x_1 + x_2 = 10$ ce qui n'est pas ≤ 8).



(a) Ajout de la seconde contrainte (b) Intersection des contraintes en vert

FIGURE 2.3 – Partage du plan par deux contraintes

Si nous ajoutons maintenant la seconde contrainte, $x_1 + 2x_2 \leq 7$, nous pouvons ajouter une nouvelle droite sur la figure, qui définit une nouvelle zone en bleu sur la figure 2.3a, et l'intersection en vert foncé, sur la figure 2.3b.

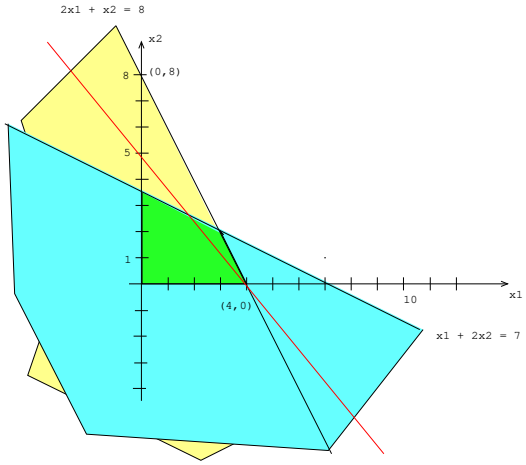


FIGURE 2.4 – Zone des solutions possibles en vert

Pour finir, nous ajoutons les contraintes d'égalité ($x_1 \geq 0$ et $x_2 \geq 0$) qui limitent la zone des solutions possibles à la partie du plan où x_1 et x_2 sont positifs, donc la partie aux dessus et à droite des axes x_1 et x_2. Nous obtenons ainsi la zone en vert clair sur la figure 2.4.

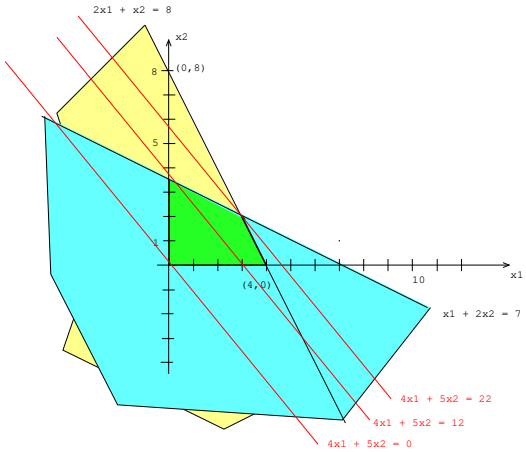


FIGURE 2.5 – Déplacement de la fonction objectif

Dans cette zone, toute combinaison de x_1 et x_2 vérifie donc les contraintes. Parmi toutes ces solutions possibles nous cherchons celle qui donne le meilleur résultat z, avec $z = 4x_1 + 5x_2$. Nous pouvons remarquer que z définit également une droite dont la valeur de z va déterminer la position sur le plan. La figure 2.5 représente, en rouge, plusieurs droites avec différentes valeurs de z. Nous voyons que la droite qui passe à l'origine (le point ($x_1 = 0$, $x_2 = 0$)) donne une valeur de $z = 0$. Plus nous déplaçons cette droite vers la droite, plus la valeur de z augmente. Le maximum est atteint pour la droite le plus à droite, avec une valeur de $z = 22$, qui est donc la meilleure solution possible : $x_1 = 3$ et $x_2 = 2$.

À noter que, si l'intersection de la droite était un segment, alors chaque point du segment serait une solution du programme linéaire avec une même valeur optimale pour l'objectif.

Nous pouvons noter finalement que, malgré le fait que les navets aient un meilleur ren-

dement au m^2, la meilleure solution suppose de cultiver une plus grande surface de courgettes, donc que les solution intuitives qui auraient consisté à favoriser la culture de navets ne donnent pas les meilleures solutions.

Méthode du simplexe

Le problème de la résolution graphique est qu'elle n'est utilisable qu'avec deux variables. Avec trois variables, la résolution se passe dans l'espace, les contraintes deviennent des plans et la représentation devient beaucoup plus complexe, et en tous les cas irréalisable sur une feuille de papier. Avec 4 variables et au delà, nous ne connaissons pas de support pour réaliser la résolution. Il faut donc utiliser une autre méthode.

La méthode la plus connue pour faire cela est la méthode du simplexe. Le simplexe est le nom donné à la zone de satisfaction des contraintes ou de réalisabilité (la zone en vert de la figure 2.4). L'idée est de parcourir le simplexe pour trouver une meilleure solution en considérant tour à tour chacune des variables et en utilisant la propriété que l'optimal se trouve sur le pourtour de simplexe. Il est possible d'illustrer ceci sur la figure 2.4 : en considérant l'augmentation la variable x_2 nous parcourons l'axe vertical, jusqu'au maximum où nous pouvons aller c'est-à-dire au point (0,3.5). En ce point le maximum n'est pas atteint, ce qui se traduit par un coefficient positif pour la variable x_1 dans la nouvelle fonction objectif. La méthode augmente alors x_1 pour parcourir la droite $x_1 + 2x_2 = 7$, ce qui a également pour effet de diminuer x_2, nous atteignons ainsi la valeur optimale au point (3,2).

Au début de la méthode, des variables d'écart sont ajoutées pour transformer les inéquations, qui sont difficiles à manipuler, en équations. Comme leur nom l'indique, les variables d'écart contiennent l'écart existant entre la partie droite et la partie gauche de l'inéquation, permettant ainsi la transformation en équation. La nouvelle forme est appelée forme standard et c'est avec elle que nous déroulons l'algorithme du simplexe. Ainsi le programme linéaire précédent, dans sa forme standard, devient :

$$\begin{cases} \text{maximiser} & z & = & 4x_1 & + & 5x_2 & & \\ \text{sous les contraintes :} & & & & & & & \\ C_1 : & x_3 & = & 8 & - & 2x_1 & - & x_2 \\ C_2 : & x_4 & = & 7 & - & x_1 & - & 2x_2 \\ C_p : & x_1, & x_2, & x_3, & x_4, & \geq & 0 & \end{cases}$$

Au début les variables recherchées sont mises à zéro, ce qui revient à partir du point (0,0) sur le graphique, de ce fait le vecteur des variables est :

$$(x_1, x_2, x_3, x_4) = (0, 0, 8, 7)$$

Les valeurs de x_3 et x_4 sont trouvées en remplaçant les valeurs de x_1 et x_2 (donc par 0) dans les contraintes.

Nous augmentons ensuite les valeurs des variables pour nous diriger vers la solution optimale. La variable x_2 ayant le plus grand coefficient, c'est elle que nous augmentons en premier, pour augmenter le plus possible la fonction objectif z. Il faut donc voir,

parmi les deux contraintes, jusqu'où nous pouvons augmenter x_2 (x_1 reste à 0) tout en respectant les contraintes de positivité.

$$\begin{array}{lrcl} C_1: & x_2 > 8 & \Rightarrow & x_3 < 0 \\ C_2: & x_2 > \frac{7}{2} & \Rightarrow & x_4 < 0 \text{ contrainte la plus stricte} \end{array}$$

Il n'est donc pas possible de donner une valeur plus grande que $\frac{7}{2}$ à x_2 si nous voulons rester dans le simplexe et garder des valeurs positives. Ceci est visible sur la figure 2.4 où nous voyons que, sur l'axe x_2, le simplexe s'arrête en 3.5.

Nous utilisons la contrainte C_2 pour remplacer x_2 (on dit la faire entrer en base puisqu'elle va passer du côté gauche des équations) puisque c'est celle qui contraint le plus la variable :

$$x_2 = \frac{7}{2} - \frac{1}{2}x_1 - \frac{1}{2}x_4$$

Ceci qui nous donne le programme suivant, en remplaçant x_1 :

$$\begin{cases} z = 4x_1 + 5(\frac{7}{2} - \frac{1}{2}x_1 - \frac{1}{2}x_4) \\ x_3 = 8 - 2x_1 - (\frac{7}{2} - \frac{1}{2}x_1 - \frac{1}{2}x_4) \\ x_2 = \frac{7}{2} - \frac{1}{2}x_1 - \frac{1}{2}x_4 \end{cases}$$

Qui se réduit en :

$$\begin{cases} z = \frac{35}{2} + \frac{3}{2}x_1 - \frac{5}{2}x_4 \\ x_3 = \frac{9}{2} - \frac{3}{2}x_1 + \frac{1}{2}x_4 \\ x_2 = \frac{7}{2} - \frac{1}{2}x_1 - \frac{1}{2}x_4 \end{cases}$$

Notons que la droite associée à x_2 ($x_2 = \frac{7}{2} - \frac{1}{2}x_1 - \frac{1}{2}x_4$) est, sur les figures de la résolution graphique, la droite qui limite le simplexe en haut ($x_1 + 2x_2 = 7$ lorsque x_4 est à 0). En modifiant x_1 tout en maintenant la contrainte C_2 , nous nous déplaçons le long de cette droite.

Les valeurs associées aux variables sont maintenant :

$$(x_1, x_2, x_3, x_4) = (0, \frac{7}{2}, \frac{9}{2}, 0)$$

Et la fonction objectif vaut $z = \frac{35}{2}$. Nous pouvons constater que, dans cette fonction, la variable x_1 a un coefficient positif, ce qui signifie que, si nous augmentons x_1, il est possible de gagner plus. Nous allons donc augmenter cette variable (la faire entrer en base) :

$$\begin{array}{lrcl} C_1: & x_1 > 3 & \Rightarrow & x_3 < 0 \text{ contrainte la plus stricte} \\ C_2: & x_1 > 7 & \Rightarrow & x_2 < 0 \end{array}$$

Nous utilisons la contrainte C_1 pour remplacer x_1, la faire entrer en base :

$$x_1 = 3 - \frac{2}{3}x_3 + \frac{1}{3}x_4$$

Ceci qui nous donne le programme suivant, en remplaçant x_1 :

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

Les valeurs associées aux variables sont maintenant :

$$\big(x_1, x_2, x_3, x_4\big) = \big(3, 2, 0, 0\big)$$

Avec des variables d'écart nulles, ce qui signifie que tous les nutriments sont utilisés puisqu'il n'y a plus d'écart entre la valeur disponible et la valeur consommée par les variables. À noter que ce n'est toujours le cas dans les programmes linéaires. Il n'y a plus de variable à coefficient positif dans notre fonction objectif, nous avons donc trouvé la valeur optimale de $z = 22$ qui apparaît dans la fonction objectif. Elle est bien cohérente avec ce que nous avions obtenu en résolution graphique : 22 kilos de légumes, pour respectivement 3 m^2 de courgettes et 2 m^2 de navets.

2.1.3 Second exemple

Dans ce second exemple, nous partons directement du problème formalisé. Il est définit par 2 variables et cinq contraintes, une de plus que précédemment :

$$\begin{cases} \text{maximiser} & x_1 & + & x_2 & & \\ \text{avec les contraintes} & 4x_1 & - & x_2 & \leq & 8 \\ & 2x_1 & + & x_2 & \leq & 10 \\ & -5x_1 & + & 2x_2 & \leq & 2 \\ \text{et} & x_1, x_2 & \geq & 0 & & \end{cases}$$

Méthode graphique

Il est également possible de résoudre ce programme linéaire graphiquement car il n'a que 2 dimensions. Les cinq contraintes permettent de définir des partitions du plan en 5 demi-plans. En faisant l'intersection de ces demi-plans compatibles avec les contraintes, la zone *réalisable* se définit (en vert sur la figure 2.6). Nous pouvons voir que l'introduction d'une troisième contrainte donne une forme différente au simplexe.

Le programme linéaire est vérifié en tout point de cette zone et il existe une valeur maximale pour l'objectif car $x_1 + x_2 = 0$ est dans la zone réalisable. Dans cet exemple, un seul point maximise la fonction objectif. Il suffit alors de déplacer la fonction linéaire objectif, ici la droite $x_1 + x_2 = b$, sur toute la zone de réalisabilité. On trouve dans ce cas un point de cette zone permettant de garantir une plus grande valeur pour l'objectif : $(2, 6)$. La figure 2.6 illustre cette résolution. Les solutions de ce programme linéaire sont $x_1 = 2$ et $x_2 = 6$.

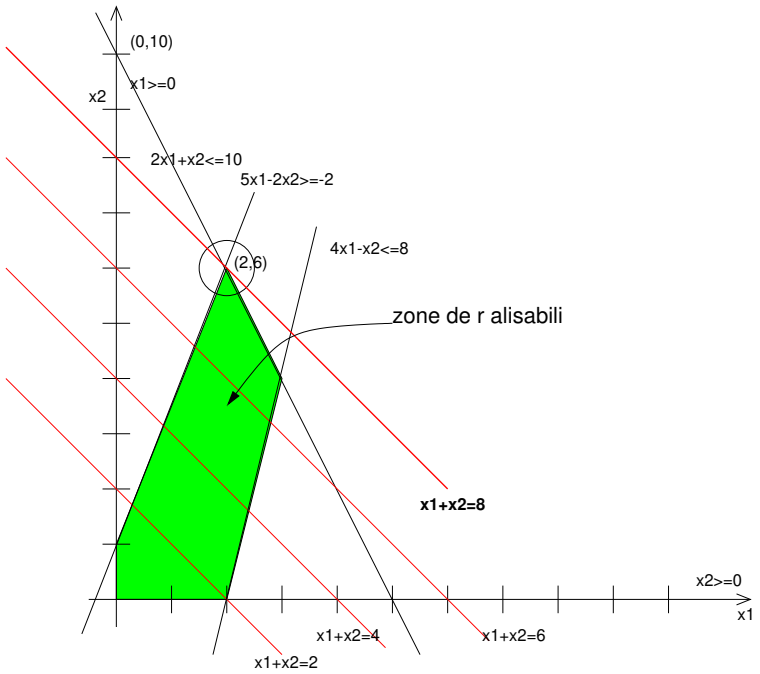


FIGURE 2.6 – Résolution graphique d'un programme linéaire à deux dimensions

Méthode du simplexe

Le programme linéaire devient, sous forme standard :

$$\begin{cases} \text{maximiser} & x_1 + x_2 \\ \text{avec les contraintes} & \\ C_1 & x_3 = 8 - 4x_1 + x_2 \\ C_2 & x_4 = 10 - 2x_1 - x_2 \\ C_3 & x_5 = 2 + 5x_1 - 2x_2 \\ C_p & x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \end{cases}$$

Notre point de départ est :

$$\big(x_1, x_2, x_3, x_4, x_5\big) = \big(0, 0, 8, 10, 2\big)$$

Avec une valeur de fonction objectif $z = 0$.

Comme les coefficients des deux variables sont les mêmes dans la fonction à maximiser nous pouvons prendre l'une ou l'autre des variables x_1 ou x_2 pour la faire entrer en base. Prenons x_2 et regardons les limites posées par les contraintes de positivité.

Dans la contrainte C_1, la variable x_2 a un coefficient positif ce qui signifie qu'en l'augmentant ne risquons pas de briser la contrainte de positivité, car plus nous augmentons la valeur de x_2 plus la variable en base correspondante (x_3) augmente. Les seules contraintes limitant la valeur de x_2 sont les contraintes C_2 et C_3. Pour calculer, quelle est la contrainte la plus stricte nous utilisons le fait que la valeur de x_2 a été mise à 0.

$$\begin{array}{lrcl} C_2 : & x_2 > 10 & \Rightarrow & x_4 < 0 \\ C_3 : & x_2 > 1 & \Rightarrow & x_5 < 0 \text{ contrainte la plus stricte} \end{array}$$

La troisième contrainte étant la plus stricte, nous exprimons x_2 à partir de cette contrainte :

$$x_2 = 1 + \frac{5}{2}x_1 - \frac{1}{2}x_5$$

Et nous la remplaçons dans le programme linéaire pour obtenir :

$$\begin{cases} z = 1 + \frac{7}{2}x_1 - \frac{1}{2}x_5 \\ x_3 = 9 - \frac{3}{2}x_1 - \frac{1}{2}x_5 \\ x_4 = 9 - \frac{9}{2}x_1 + \frac{1}{2}x_5 \\ x_2 = 1 + \frac{5}{2}x_1 - \frac{1}{2}x_5 \end{cases}$$

Ce qui nous donne la solution suivante :

$$\big(x_1, x_2, x_3, x_4, x_5\big) = \big(0, 1, 9, 9, 0\big)$$

Avec une valeur de fonction objectif $z = 1$. Nous avons suivi l'axe x_2 pour arriver au point $(x_1 = 0, x_2 = 1)$.

Nous faisons maintenant entrer x_1 en base. Pour calculer, quelle est la contrainte la plus stricte nous utilisons le fait que la valeur de x_5 a été mise à 0. dans la troisième équation x_1 a un coefficient positif. Nous ne l'utilisons donc pas.

$$\begin{array}{lrcl} C_1 : & x_1 > 6 & \Rightarrow & x_3 < 0 \\ C_2 : & x_1 > 2 & \Rightarrow & x_4 < 0 \text{ contrainte la plus stricte} \end{array}$$

Nous utilisons donc la contrainte C_2 pour exprimer x_1 :

$$x_1 = 2 - \frac{2}{9}x_4 + \frac{1}{9}x_5$$

Ce qui donne le programme linéaire suivant :

$$\begin{cases} z = 8 - \frac{7}{9}x_4 - \frac{5}{18}x_5 \\ x_3 = 6 - \frac{1}{3}x_4 - \frac{2}{3}x_5 \\ x_1 = 2 - \frac{2}{9}x_4 + \frac{1}{9}x_5 \\ x_2 = 6 - \frac{5}{9}x_4 - \frac{2}{9}x_5 \end{cases}$$

Nous obtenons la solution suivante :

$$\big(x_1, x_2, x_3, x_4, x_5\big) = \big(2, 6, 6, 0, 0\big)$$

Avec une valeur de fonction objectif $z = 8$.

Dans laquelle les valeurs de x_1 et x_2 correspondent bien à ce que donne la solution graphique. Nous pouvons également observer que les valeurs maximales pour les contraintes C_2 et C_3 sont atteintes puisque les variables d'écart x_4 et x_5 sont à zéro. Par contre, dans la contrainte C_1 la variable x_3 reste avec une valeur de 6, ce qui signifie que cette contrainte n'est pas à l'égalité, elle est donc moins contraignante pour ces valeurs de x_1 et x_2. Si nous reprenons les valeurs de la contrainte initiale :

$$4x_1 - x_2 = 2 \geq 8$$

Nous retrouvons bien cet écart de 6 affecté à x_3. Il est également possible de constater sur la figure 2.6 que la solution optimale n'est pas sur la droite de cette contrainte.

2.2 Problème général

Les exemples précédents nous ont permis de présenter les principes de base de la programmation linéaire. Nous nous intéressons maintenant à la généralisation du problème.

En fait, la démarche est la même pour un programme linéaire avec un plus grand nombre de variables. La région de réalisabilité est alors formée, comme il a été suggéré précédemment, par l'intersection des demi-espaces définis dans les contraintes du programme linéaire. Cette région est convexe, ce qui veut dire que si nous traçons entre droite entre deux points quelconques de cette région, toute la droite fait partie de la région. Lorsque le nombre de variables est n, le simplexe est de dimension n et les demi-espaces de dimension n sont caractérisés par des hyperplans de dimension $n-1$. Ce faisant, la fonction objectif est un hyperplan de dimension $n-1$, mobile dans le simplexe. Or comme le simplexe est convexe, la solution du programme se trouve sur l'un des hyperplans définis par l'intersection entre l'hyperplan de l'objectif et le simplexe. Cet intersection a une dimension inférieure ou égale à $n-1$. Suivant la dimension de cette intersection, la solution du programme linéaire est unique ou non. C'est cette propriété qui est utilisée par *l'algorithme du simplexe* pour rechercher la solution. Cette recherche s'arrête dès la rencontre d'un *optimum local* qui est en fait un *optimum global* car la région réalisable est convexe.

2.2.1 Définition

La programmation linéaire permet de résoudre une catégorie de problèmes d'optimisation caractérisés par :

- Un ensemble de variables, généralement notées x_j
- un ensemble de contraintes qui peuvent s'exprimer sous la forme d'inégalités linéaires. Une inégalité linéaire est de la forme $f(x_1, x_2, \ldots, x_n) \leq b$ ou $\geq b$ (inégalité au sens large en programmation linéaire), où f est une fonction linéaire donc de la forme : $f(x_1, x_2, \ldots, x_n) = \sum_{j=1}^{n} a_j x_j$
- Un objectif d'optimisation exprimé sous la forme d'une fonction linéaire $f(x_1, x_2, \ldots, x_n)$.

Pour la résolution du problème, nous passons donc d'abord par une phase de formalisation. Les programmes linéaires sont principalement exprimés sous la *forme canonique* et la *forme standard*. Dans la forme canonique, les contraintes s'expriment sous la forme d'inégalités ($\leq$) alors que celles-ci sont des égalités dans la forme standard.

Le problème de programmation linéaire est la *maximisation/minimisation* d'une fonction linéaire. La résolution de ce problème peut se faire sous forme graphique si nous n'avons deux variables mais la résolution générale se fait par la méthode du *simplexe* qui utilise la forme standard.

2.2.2 Forme canonique

La forme canonique est une manière particulière d'écrire le programme linéaire. Il s'agit de transformer le programme initial, si besoin, afin que la fonction objectif soit une fonction

à *maximiser* et que les contraintes soient toutes des inégalités bornées par un majorant constant. Le programme linéaire en forme standard s'écrit donc de la manière suivante, avec $n + m$ inégalités (n variables et m contraintes) :

$$\begin{cases} \text{maximiser} & \sum_{j=1}^{n} c_j x_j \text{ (fonction objectif)} \\ \text{avec les contraintes} & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \text{ avec } i = 1, \ldots, m \\ \text{et les contraintes de positivité} & x_j \geq 0 \text{ avec } j = 1, \ldots, n \end{cases}$$

2.2.3 Conversion sous forme canonique

La conversion d'un problème linéaire sous forme canonique peut nécessiter un certain nombre de transformations pour arriver au format demandé.

Soit le programme linéaire suivant à transformer sous forme canonique :

$$\begin{cases} \text{minimiser} & -2x_1 & + & 3x_2 & & \\ \text{avec les contraintes} & x_1 & + & x_2 & = & 7 \\ & x_1 & - & 2x_2 & \leq & 4 \\ & & & x_1 & \geq & 0 \end{cases}$$

Minimiser la fonction objectif : Minimiser une fonction linéaire est équivalent à maximiser la fonction linéaire opposée. Il suffit donc de tout multiplier par -1. La forme prise par le nouveau programme linéaire peut alors varier par rapport à ce que nous voyons dans ce cours ()coefficients négatifs dans la fonction objectif, contrainte avec un nombre négatif, etc) et nécessiter l'introduction de nouvelles variables pour revenir à la structure de base.

Contrainte de positivité La conversion d'un programme linéaire avec des variables sans contrainte de positivité se fait en remplaçant la variable sans contrainte de positivité par la différence de deux variables ayant elles des contraintes de positivité :

$$x_j \rightarrow x'_j - x''_j \text{ avec } x'_j \geq 0 \text{ et } x''_j \geq 0$$

Il faut également faire le remplacement dans la fonction objectif et dans toutes les inéquations et équations utilisant la variable x_j. Ainsi, $c_j x_j$ devient $c_j x'_j - c_j x''_j$ dans la fonction objectif et $a_{ij} x_j$ devient $a_{ij} x'_j - a_{ij} x''_j$. Dans l'exemple, le programme linéaire devient le programme suivant :

$$\begin{cases} \text{maximiser} & 2x_1 & - & 3x'_2 & + & 3x''_2 & & \\ \text{avec les contraintes} & x_1 & + & x'_2 & - & x''_2 & = & 7 \\ & x_1 & - & 2x'_2 & + & 2x''_2 & \leq & 4 \\ \text{et} & & & & & x_1, x'_2, x''_2 & \geq & 0 \end{cases}$$

Transformation des égalités S'il existe une contrainte dont l'expression se fait avec une égalité, il faut la transformer en deux contraintes, l'une majorée par la valeur de

l'égalité, au sens large, et la deuxième minorée par l'opposé de la valeur de l'égalité, les coefficients de cette deuxième équation ayant été inversés également. Ainsi la première contrainte du programme linéaire exemple se transforme ainsi :

$$x_1 + x'_2 - x''_2 = 7 \Rightarrow \begin{cases} x_1 + x'_2 - x''_2 \leq 7 \\ -x_1 - x'_2 + x''_2 \leq -7 \end{cases}$$

Si on pose $x'_2 = x_2$ et $x''_2 = x_3$, on obtient le programme linéaire suivant :

$$\begin{cases} \text{maximiser} & 2x_1 & - & 3x_2 & + & 3x_3 & & \\ \text{avec les contraintes} & x_1 & + & x_2 & - & x_3 & \leq & 7 \\ & -x_1 & - & x_2 & + & x_3 & \leq & -7 \\ & x_1 & - & 2x_2 & + & 2x_3 & \leq & 4 \\ \text{et} & & & & & x_1, x_2, x_3 & \geq & 0 \end{cases}$$

Une fois le problème linéaire mis sous forme canonique, nous pouvons parler de programme linéaire. Nous utilisons la forme canonique pour générer la forme standard qui est utilisée par l'algorithme du simplexe.

2.2.4 Conversion sous forme standard

La forme standard est la forme utilisée par l'algorithme du simplexe. Si les contraintes ont la forme canonique, il faut ajouter des variables d'écart marquant la différence entre la valeur de la contrainte et la fonction linéaire exprimant la contrainte. Ainsi :

$$\text{Si } \sum_{j=1}^{n} a_{ij} x_j \leq b_i$$

On introduit la variable *d'écart* s pour transformer l'inégalité précédente en l'égalité suivante :

$$s = b_i - \sum_{j=1}^{n} a_{ij} x_j \text{ avec } s \geq 0$$

Pour simplifier, on note x_{n+i} la variable d'écart associée à la i-ème contrainte, sachant que le programme linéaire compte n contraintes. D'où l'écriture de la i-ème contrainte sous la forme de l'égalité suivante :

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j \text{ avec } x_{n+i} \geq 0$$

Le programme exemple traité au paragraphe 2.2.3 s'écrit donc comme suit :

$$\left\{\begin{array}{lrcrcrcrcr} \text{maximiser} & 2x_1 & - & 3x_2 & + & 3x_3 & & & & \\ \text{avec les contraintes} & x_4 & = & 7 & - & x_1 & - & x_2 & + & x_3 \\ & x_5 & = & -7 & + & x_1 & + & x_2 & - & x_3 \\ & x_6 & = & 4 & - & x_1 & + & 2x_2 & - & 2x_3 \\ \text{et} & x_1, x_2, x_3, x_4, x_5, x_6 & \geq & 0 & & & & & & \end{array}\right.$$

On appelle les variables d'écart, les *variables de base*, et les variables utilisées jusqu'alors dans les fonctions linéaires et dans la fonction objectif sont les *variables hors base*.

Dans la suite, on oublie les contraintes de positivité des variables et on pose z la valeur de l'objectif. Le programme linéaire précédent s'écrit donc sous la forme suivante :

$$\left\{\begin{array}{rcrcrcrcr} z & = & 2x_1 & - & 3x_2 & + & 3x_3 & & \\ x_4 & = & 7 & - & x_1 & - & x_2 & + & x_3 \\ x_5 & = & -7 & + & x_1 & + & x_2 & - & x_3 \\ x_6 & = & 4 & - & x_1 & + & 2x_2 & - & 2x_3 \end{array}\right.$$

Forme générale d'un programme linéaire standard

Un programme linéaire peut également être exprimé de manière générale à partir des données qui le caractérisent.

$$\left\{\begin{array}{rcrcl} z & = & v & + & \sum_{j\in N} c_j x_j \\ x_i & = & b_i & - & \sum_{j\in N} a_{ij} x_j \text{ pour } i \in B \end{array}\right.$$

Avec N l'ensemble des indices des variables *hors base* ($|N| = n$), qui caractérise donc les variables dont nous recherchons la valeur, et B l'ensemble des indices des variables de *base* ($|B| = m$), qui caractérise donc les variables d'écart, une par contrainte. La matrice A contient les coefficients a_{ij} des variables hors base, le vecteur b est formé des constantes b_i des égalités linéaires et le vecteur c des coefficients c_j de la fonction objectif.

La suite de données (N, B, A, b, c, v) représente la forme standard d'un programme linéaire.

Par exemple, le programme linéaire suivant :

$$\left\{\begin{array}{rcrcrcrcr} z & = & 28 & - & \frac{x_3}{6} & - & \frac{x_5}{6} & - & \frac{2x_6}{3} \\ x_1 & = & 8 & + & \frac{x_3}{6} & + & \frac{x_5}{6} & - & \frac{x_6}{3} \\ x_2 & = & 4 & - & \frac{8x_3}{3} & - & \frac{2x_5}{3} & + & \frac{x_6}{3} \\ x_4 & = & 18 & - & \frac{x_3}{2} & + & \frac{x_5}{2} & & \end{array}\right.$$

peut être représenté par (N, B, A, b, c, v) avec les valeurs suivantes :

$$B = \left\{1, 2, 4\right\} \quad N = \left\{3, 5, 6\right\} \quad A = \begin{pmatrix} -\frac{1}{6} & -\frac{1}{6} & \frac{1}{3} \\ \frac{8}{3} & \frac{2}{3} & -\frac{1}{3} \\ \frac{1}{2} & -\frac{1}{2} & 0 \end{pmatrix}$$

$$b = \begin{pmatrix} 8 \\ 4 \\ 18 \end{pmatrix} \quad c = \begin{pmatrix} -\frac{1}{6} \\ -\frac{1}{6} \\ -\frac{2}{3} \end{pmatrix} \quad v = 28$$

2.3 Algorithme du simplexe

Il existe différentes techniques permettant de trouver des solutions à un programme linéaire. L'algorithme du simplexe est un algorithme donnant de très bons résultats en terme de vitesse de résolution bien qu'il soit exponentiel dans les pires cas. À l'inverse l'algorithme de *l'ellipsoïde*[2] est lui linéaire mais il est très lent en pratique. Dans le cas où les solutions possible doivent être des nombres entiers, la recherche des solutions devient un problème NP-Complet.

L'algorithme du simplexe n'est pas un algorithme polynomial dans le cas le plus défavorable. En revanche, en pratique il donne d'excellentes performances. Son mode de calcul ressemble un peu à la méthode du pivot de Gauss. Pour pouvoir utiliser cet algorithme, il faut que le programme linéaire soit écrit sous la forme standard. Très souvent, les logiciels de résolution de programmes linéaires admettent différentes présentations pour l'écriture des programmes linéaires et se chargent de faire la conversion avant l'étape de la résolution.

Le principe de cet algorithme est de maximiser la fonction objectif en maximisant chacune des variables hors base dans la fonction objectif. Pour cela, on extrait ces variables les unes après les autres et on remplace leur expression dans le reste du programme. Lorsqu'il ne reste plus aucune variable avec un coefficient positif dans l'expression de la fonction objectif et que la solution est valide, cela signifie qu'il n'est plus possible d'augmenter sa valeur. Il ne reste plus qu'à mettre les variables de la fonction objectif à cet instant, à zéro. Les autres variables sont des variables de base pour lesquelles il est facile de calculer la valeur. On a donc tous les éléments pour donner la solution du programme linéaire, valeur de la fonction objectif et valeur des variables hors base du programme initial.

2.3.1 Exemple d'exécution avec 3 variables

Nous traitons ici un exemple à trois variables, donc qui ne peut pas être facilement résolu de manière graphique.

Soit le programme suivant écrit sous sa forme canonique :

$$\begin{cases} \text{maximiser} & & & & & & 3x_1 & + & x_2 & + & 2x_3 \\ \text{avec les contraintes} & x_1 & + & x_2 & + & 3x_3 & \leq & & 30 & & \\ & 2x_1 & + & 2x_2 & + & 5x_3 & \leq & & 24 & & \\ & 4x_1 & + & x_2 & + & 2x_3 & \leq & & 36 & & \\ \text{et} & & & & & x_1, x_2, x_3 & \geq & & 0 & & \end{cases}$$

La forme standard de ce programme linéaire est la suivante :

$$\begin{cases} z & = & & & 3x_1 & + & x_2 & + & 2x_3 \\ x_4 & = & 30 & - & x_1 & - & x_2 & - & 3x_3 \\ x_5 & = & 24 & - & 2x_1 & - & 2x_2 & - & 5x_3 \\ x_6 & = & 36 & - & 4x_1 & - & x_2 & - & 2x_3 \\ \text{et} & & x_1, x_2, x_3, x_4, x_5, x_6 & & & \geq & 0 & & \end{cases}$$

2. https://fr.wikipedia.org/wiki/M%C3%A9thode_de_l%27ellipso%C3%AFde

Ce système de contraintes a 3 équations et 6 inconnues. Il y a par conséquent un nombre infini de solutions *réalisables*. Il s'agit pour nous de trouver celle qui maximise la fonction objectif.

Solution de base

Pour trouver les solutions de base, nous mettons les variables à droite du signe égal à zéro (variable hors base). Les variables de base prennent donc la valeur de la constante des équations linéaires. Dans l'exemple, les solutions de base sont les suivantes :

$$\big(x_1, x_2, x_3, x_4, x_5, x_6\big) = \big(0, 0, 0, 30, 24, 36\big)$$

Dans ce cas, la valeur de l'objectif est $z = 0$ et $x_i = b_i \ \forall i \in B$. Comme nous l'avons vu précédemment, la solution de base n'est pas toujours une solution réalisable. Ceci ne remet pas en question la résolution du programme linéaire par l'algorithme du simplexe.

Réécriture de l'ensemble des équations et la fonction objectif

Pour ramener en base les variables que nous recherchons, en conservant les contraintes du programme linéaire initial, il faut le réécrire dans les équations sous une autre forme. Nous itérons sur ces transformations jusqu'à ce qu'il ne soit plus possible d'améliorer la fonction objectif, c'est-à-dire que les contraintes ne permettent plus d'augmenter les valeurs des variables.

Pour réécrire le programme linéaire, il faut choisir une variable *hors base* x_e ayant un coefficient positif dans la fonction objectif, sans quoi il serait impossible d'augmenter la fonction objectif, les variables étant toujours positives. On choisit en général la variable x_e ayant le plus fort coefficient dans la fonction objectif. Il faut ensuite donner à cette variable x_e la plus grande valeur possible sans qu'aucune contrainte ne soit violée.

Soit l l'équation du programme linéaire la plus restrictive pour la valeur de x_e, c'est à dire l'équation qui limite le plus l'augmentation de la valeur de x_e.

On exprime alors x_e en fonction de la variable x_l, des autres variables hors base et de la constante. Ainsi, x_e devient une variable de base et x_l devient une variable hors base.

Dans les autres équations du programme linéaire, ainsi que dans la fonction objectif, on remplace x_e par l'expression trouvée au niveau de l'équation l du programme linéaire.

Exemple (itération 1) : Dans l'exemple, on choisit d'augmenter la variable x_1. On garde les autres variables hors base à zéro et on augmente la valeur de x_1 le plus possible, tout en conservant des valeurs positives pour les variables de base. On ne garde que l'équation imposant la contrainte la plus stricte. On observe donc :

$$\begin{aligned} x_1 > 30 &\Rightarrow x_4 < 0 \\ x_1 > 12 &\Rightarrow x_5 < 0 \\ x_1 > 9 &\Rightarrow x_6 < 0 \text{ contrainte la plus stricte} \end{aligned}$$

Comme expliqué précédemment, on permute les rôles entre x_1 et x_6. La variable x_1 devient une variable de base et x_6 devient une variable hors base. Pour cela il faut exprimer x_1 en fonction des autres variables dans la dernière équation :

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}$$

Réécrivons la première équation de ce programme linéaire :

$$\begin{array}{l} x_4 = 30 - x_1 - x_2 - 3x_3 \\ x_4 = 30 - (9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}) - x_2 - 3x_3 \\ x_4 = 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \end{array}$$

On réécrit les autres équations et la fonction objectif et on obtient le même programme linéaire, mais écrit autrement :

$$\left\{ \begin{array}{rcrcrcrcr} z & = & 27 & + & \frac{x_2}{4} & + & \frac{x_3}{2} & - & \frac{3x_6}{4} \\ x_1 & = & 9 & - & \frac{x_2}{4} & - & \frac{x_3}{2} & - & \frac{x_6}{4} \\ x_4 & = & 21 & - & \frac{3x_2}{4} & - & \frac{5x_3}{2} & + & \frac{x_6}{4} \\ x_5 & = & 6 & - & \frac{3x_2}{2} & - & 4x_3 & + & \frac{x_6}{2} \end{array} \right.$$

Il s'agit d'une opération de *pivot* avec x_1 une variable *entrante* et x_6 une variable *sortante*.

Avec cette nouvelle présentation du programme linéaire, on cherche la valeur de la solution de base en annulant toutes les variables hors base. On obtient une valeur de $z = 27$ pour les solutions suivantes :

$$\big(\overline{x_1}, \overline{x_2}, \overline{x_3}, \overline{x_4}, \overline{x_5}, \overline{x_6}\big) = \big(9, 0, 0, 21, 6, 0\big)$$

Exemple (itération 2) : Pour la deuxième itération on choisit d'augmenter la variable x_3. On a les contraintes suivantes sur la valeur de x_3 afin que les valeurs des variables de base restent positives :

$$\begin{array}{rcl} x_3 > 18 & \Rightarrow & x_1 < 0 \\ x_3 > \frac{42}{5} & \Rightarrow & x_4 < 0 \\ x_3 > \frac{3}{2} & \Rightarrow & x_5 < 0 \text{ contrainte la plus stricte} \end{array}$$

On passe donc x_3 du coté des variables de base et on passe x_5 du coté des variables hors base. On remplace ensuite la valeur de x_3 dans les autres équations ainsi que dans la fonction objectif. On obtient une autre formulation pour le programme linéaire :

$$\left\{ \begin{array}{rcrcrcrcr} z & = & \frac{111}{4} & + & \frac{x_2}{16} & - & \frac{x_5}{8} & - & \frac{11x_6}{16} \\ x_1 & = & \frac{33}{4} & - & \frac{x_2}{16} & + & \frac{x_5}{8} & - & \frac{5x_6}{16} \\ x_3 & = & \frac{3}{2} & - & \frac{3x_2}{8} & - & \frac{x_5}{4} & + & \frac{x_6}{8} \\ x_4 & = & \frac{69}{4} & + & \frac{3x_2}{16} & + & \frac{5x_5}{8} & - & \frac{x_6}{16} \end{array} \right.$$

La fonction objectif devient $z = \frac{111}{4}$ avec la solution de base suivante :

$$\big(\overline{x_1}, \overline{x_2}, \overline{x_3}, \overline{x_4}, \overline{x_5}, \overline{x_6}\big) = \big(\frac{33}{4}, 0, \frac{3}{2}, \frac{69}{4}, 0, 0\big)$$

Exemple (itération 3) : Augmentation de la valeur de x_2. Il s'agit de la dernière variable ayant un coefficient positif dans la fonction objectif. En agissant sur les valeurs possible pour x_2, en annulant les autres variables hors base, on trouve l'équation la plus restrictive :

$$\begin{aligned} x_2 > 132 &\Rightarrow x_1 < 0 \\ x_2 > 4 &\Rightarrow x_3 < 0 \end{aligned}$$

x_2 peut être aussi grand que possible car x_4 et x_2 grandissent ensemble

La deuxième équation est la plus restrictive. On exprime donc x_2 en fonction de x_3, x_5 et x_6. Après remplacement de la valeur de x_2 dans le reste du programme linéaire on obtient le programme linéaire suivant :

$$\begin{cases} z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \\ x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \\ x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \\ x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2} \end{cases}$$

Exemple (solution du programme linéaire) : Les coefficients des variables hors base de la fonction objectif sont tous négatifs. Il n'est donc plus possible de l'augmenter encore. Par conséquent, on a atteint la solution optimale. Pour connaître les valeurs des variables du programme linéaire initial permettant d'atteindre cet optimal, on procède de la même manière que précédemment, on annule les valeurs des variables hors base :

$$\left(\overline{x_1}, \overline{x_2}, \overline{x_3}, \overline{x_4}, \overline{x_5}, \overline{x_6}\right) = \left(8, 4, 0, 18, 0, 0\right) \text{ avec } z_{max} = 28$$

La solution du problème initial est donc :

$$\begin{cases} x_1 = 8 \\ x_2 = 4 \\ x_3 = 0 \end{cases}$$

Les variables d'écart, ajoutée au moment de l'expression du programme linéaire sous la forme standard ne sont pas toujours nulles à l'issu de la résolution du programme linéaire. Comme leur nom l'indique, elles marquent la latitude existant entre la valeur de la contrainte et la solution.

Ici $x_4 = 18$. Cela signifie qu'il existe un écart de 18 entre la fonction linéaire exprimant la contrainte et la borne sur cette contrainte. Cette borne aurait donc pu être plus restrictive. On le vérifie en remplaçant les valeurs de x_1, x_2 et x_3 dans la première inéquation du programme linéaire initial :

$$\begin{aligned} \underbrace{x_1 + x_2 + x_3}_{12} &\leq 30 \\ &\leq 30 \end{aligned}$$

NB : Attention les solutions ne sont pas toujours entières.

2.3.2 Calcul du pivot

L'exemple présenté dans le paragraphe précédent illustre une méthode clairement déterministe pour la manipulation des équations dans le but de réécrire le programme linéaire. Dans ce paragraphe est exposé l'algorithme permettant de calculer des nouveaux coefficients du programme linéaire sous sa forme standard $(\widehat{N}, \widehat{B}, \widehat{A}, \widehat{b}, \widehat{c}, \widehat{v})$, compte tenu de l'indice de la variable sortante l et de la variable entrante e. Les autres paramètres de cette fonction sont les valeurs des coefficients permettant de décrire le programme linéaire sous forme standard, c'est à dire (N, B, A, b, c, v). L'algorithme 12 illustre le calcul des nouveaux indices.

Algorithme 12 : Algorithme du pivot pour la réécriture de la nouvelle forme du programme linéaire compte tenu des deux variables entrante et sortante.

(N, B, A, b, c, v) : le programme linéaire courant sous sa forme standard
l : l'indice de la variable sortante
e : l'indice de la variable entrante
retourner $(\widehat{N}, \widehat{B}, \widehat{A}, \widehat{b}, \widehat{c}, \widehat{v})$
Fonction `pivot`(N, B, A, b, c, v, l, e)
début
    `/* coefficients de l'équation pour la nouvelle variable de base */`
    $\widehat{b_e} \leftarrow b_l / a_{le}$
    pour $j \in N \setminus \{e\}$ **faire**
        $\widehat{a_{ej}} \leftarrow a_{lj} / a_{le}$
    $\widehat{a_{el}} \leftarrow 1 / a_{le}$
    `/* coefficients des contraintes restantes */`
    pour $i \in B \setminus \{l\}$ **faire**
        $\widehat{b_i} \leftarrow b_i - a_{ie} \times \widehat{b_e}$
        pour $j \in N \setminus \{e\}$ **faire**
            $\widehat{a_{ij}} \leftarrow a_{ij} - a_{ie} \times \widehat{a_{ej}}$
        $\widehat{a_{il}} \leftarrow a_{ie} \times \widehat{a_{el}}$
    `/* calcul de la fonction objectif */`
    $\widehat{v} \leftarrow v + c_e \times \widehat{b_e}$
    pour $j \in N \setminus \{e\}$ **faire**
        $\widehat{c_j} \leftarrow c_j - c_e \times \widehat{a_{ej}}$
    $\widehat{c_l} \leftarrow -c_e \times \widehat{a_{el}}$
    `/* ensemble des variables de base et hors base */`
    $\widehat{N} \leftarrow N \setminus \{e\} \cup \{l\}$
    $\widehat{B} \leftarrow B \setminus \{l\} \cup \{e\}$

2.3.3 Calcul du simplexe

L'algorithme 13 montre comment on itère sur la réécriture du programme linéaire jusqu'à ce que la fonction objectif ne puisse plus être augmentée.

Algorithme 13 : Algorithme du calcul du simplexe.

```
1  (A, b, c) : le programme linéaire courant sous sa forme canonique
2  retourner (x̄_1, x̄_2, ..., x̄_n)
3  Fonction simplexe(A, b, c)
4  début
5      (N, B, A, b, c, v) ← initialiseSimplexe(A, b, c)
6      tant que ∃j ∈ N vérifiant c_j > 0 faire
7          choisir e ∈ N avec c_e > 0
8          pour chaque i ∈ B faire
9              si a_ie > 0 alors
10                 Δ_i ← b_i / a_ie
11             sinon
12                 Δ_i ← +∞
13         choisir l ∈ B qui minimise Δ_i
14         si Δ_i = +∞ alors
15             retour "non borné"
16         sinon
17             (N, B, A, b, c, v) ←pivot(N, B, A, b, c, v, l, e)
18     pour i de 1 à n faire
19         si i ∈ B alors
20             x̄_i ← b_i
21         sinon
22             x̄_i ← 0
```

2.3.4 Applications

La programmation linéaire est utilisée pour résoudre des problèmes d'optimisation comme :

- la planification de l'affectation des personnels sur les vols des compagnies aériennes ;
- la maximisation du pétrole extrait en fonction du nombre de puits de forages ;
- la résolution du problème de graphes et de combinatoires des flots ;
- le calcul d'un ordonnancement de tâches sur des machines en régime permanent ;
- ...

2.4 Synthèse

À l'issue de ce chapitre, et après avoir traité les exercices qui vont avec, vous devez être capables de :

- Comprendre le champ d'application de la programmation linéaire ;
- Modéliser un problème simple sous la forme d'un programme linéaire ;

— Résoudre un programme linéaire simple soit avec l'algorithme du simplexe, soit avec une méthode graphique ;

La programmation linéaire s'appuie sur le domaine de la recherche opérationnelle. Son utilisation est, bien sûr, plus large que les problèmes simples abordés dans le cours. Par exemple, elle peut viser la transformation de problèmes non-linéaires en problèmes linéaire, en définissant des variables de substitution, la résolution de problèmes où les variables sont discrètes, etc..

Chapitre 3

Programmation dynamique

Avec la programmation linéaire nous avons vu comment trouver une solution à un problème d'optimisation dont les variables sont réelles et la fonction objectif est linéaire. Nous abordons ici un cas où les variables sont entières et où la recherche de l'optimal suppose d'explorer tout ou partie des solutions possibles.

Nous avons déjà évoqué le fait que trouver un chemin de taille minimale entre deux sommets d'un graphe est un problème combinatoire car il peut se résoudre en calculant tous les chemins élémentaires (ou combinaisons de sommets) existants entre les deux sommets puis en cherchant le plus court parmi ceux-ci. De la même manière choisir des objets, caractérisés par un poids et une valeur, à mettre dans un sac à dos ne pouvant supporter plus qu'un poids fixé [1] pour maximiser la valeur du sac est un problème combinatoire. Il peut-être résolu en prenant toutes les combinaisons d'objets dont la somme des poids ne dépasse pas la limite du sac à dos et en choisissant la combinaison qui a le plus de valeur parmi celles-ci. Ces approches "brute force" nécessitent souvent un grand nombre de calculs pour la résolution d'un problème et il est nécessaire de trouver des approches plus intelligentes peut améliorer l'efficacité de la recherche.

La programmation dynamique est une des techniques de programmation qui visent à réduire le temps de calcul d'une solution optimale. Elle peut aussi bien s'appliquer si le problème a une complexité polynomiale (cas du plus court chemin) que si le problème a une complexité exponentielle (cas du sac à dos). Elle est souvent appliquée à des problèmes d'optimisation où :

- il existe un très grand nombre de solutions (combinatoire). Il est donc coûteux, voire impossible, de calculer toutes les solutions pour choisir la meilleure.
- on recherche une solution optimale (il peut en exister plusieurs), par exemple celle dont le coût est minimal ou maximal

1. Ce problème est un problème classique de la programmation dynamique. Il est plus connu sous le nom de problème du sac à dos ou knapsack problem

3.1 Introduction

La programmation dynamique aborde la recherche de solutions optimales lorsqu'une qu'une solution peut-être calculée à partir de sous-solutions. Si nous sommes sûr que la solution optimale finale dépend des solutions optimales des sous-solutions, alors le problème peut être représenté de manière récursive et la recherche de la solution optimale ne nécessite que le calcul d'un nombre limité de sous-solutions.

Le principe la programmation dynamique a été introduit par Richard Bellmann et utilisé dans l'algorithme de Bellmann-Ford pour la recherche d'un plus court chemin. Il est également appelé principe d'optimalité de Bellmann. Il est illustré dans ce chapitre par le problème de la recherche de la plus courte distance entre deux sommets u et s, déjà évoqué au chapitre 1. Puisque le plus court chemin, celui qui a la plus courte distance, arrive au sommet v alors il passe forcément par un des voisins du sommet v. De ce fait la plus courte distance entre u et v peut-être calculée à partir de la plus courte distance entre u et chacun des voisins de v (sous-problème optimal). Elle repose sur le chemin qui obtient le plus petit score quand on additionne la longueur du chemin de u à un voisin de v et la distance de ce voisin à v.

Cette formalisation conduit, de manière implicite à une solution récursive. Le problème est que cette solution récursive conduit à calculer un nombre exponentiel de sous-solutions puisque, à partir de chaque voisins il faut refaire le travail pour chaque voisin, ce qui conduit à un arbre. La solution récursive n'est donc plus utilisable dès que la taille de l'arbre grandit. Pour résoudre ce problème, la programmation dynamique utilise la technique de la mémoïsation. La mémoïsation consiste à mémoriser des solutions intermédiaires pour en éviter le recalcul. En programmation dynamique elle permet de stocker les sous-solutions utilisées pour la recherche de la solution optimale.

3.2 Mémoïsation

Nous illustrons l'intérêt de la technique de mémoïsation, dans les problèmes combinatoires avec le calcul de la suite de Fibonacci. Leonardo Fibonacci est un mathématicien italien ayant vécu aux 12 et 13ème siècles. Il est célèbre pour avoir pour avoir posé le problème suivant [2] : "Un homme met un couple de lapins dans un lieu isolé de tous les côtés par un mur. Combien de couples obtient-on en n mois si chaque couple engendre tous les mois un nouveau couple à compter du troisième mois de son existence ? En supposant que les lapins ne meurent jamais.".

Il est possible de voir, d'après la formulation du problème, qu'il n'y aura pas d'autre couple de lapin que le couple initial avant que celui-ci puisse se reproduire, c'est-à-dire, le troisième mois. Ensuite le nombre de lapins au mois n est la somme du nombre de lapins du mois précédent (L_{n-1}) et des lapins engendrés par les couples en âge de se reproduire, donc le nombre de lapins qui a au moins deux mois (L_{n-2}), puisque chacun de ces couples va donner naissance à un nouveau lapin . Ce qui nous donne la définition récursive de la

2. La formulation est inspirée de https://fr.wikipedia.org/wiki/Suite_de_Fibonacci

suite de Fibonacci modifiée suivante :

$$L_n = \begin{cases} 1 \text{ si } n = 1 \text{ ou } 2 \\ L_n = L_{n-1} + L_{n-2} \text{ si } 2 < n \end{cases}$$

En utilisant cet définition nous pouvons écrire l'algorithme 14 qui donne un programme récursif de calcul de la suite de Fibonacci.

Algorithme 14 : Fonction récursive de calcul de la suite de Fibonacci

```
1 Fonction FiboRec(n)
2 début
3     si n ≤ 2 alors retourner 1
4     sinon retourner FiboRec(n-1) + FiboRec(n-2)
```

Si nous regardons maintenant ce qui se passe lors de l'appel de la fonction **FiboRec(5)**, nous obtenons l'arbre donné par la figure 3.1.

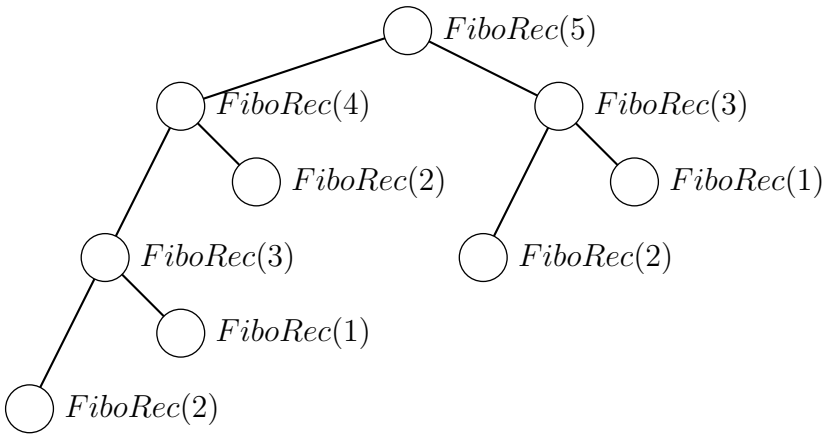


FIGURE 3.1 – Arbre du calcul de l'appel **FiboRec(5)**

Il est facile de voir, à partir de cet arbre, que l'appel de la fonction récursive, engendre :

1. un nombre exponentiel de calculs : 1 pour $FiboRec(1)$ et $FiboRec(2)$, 2 pour $FiboRec(3)$, 4 pour $FiboRec(4)$, 8 pour $FiboRec(5)$, et donc 2^{n-2} pour $FiboRec(n), (n > 2)$,
2. engendre le recalcul d'un grand nombre de valeurs de la fonction : par exemple la valeur **FiboRec(3)** est calculée deux fois et on peut voir qu'elle le sera 4 fois pour **FiboRec(6)**, etc.

Pour éviter le recalcul des valeurs déjà calculées, la technique de mémoïsation consiste à mémoriser ces valeurs dans un tableau. La construction de la solution ne se fait plus alors de manière descendante, en partant du résultat à obtenir mais de manière ascendante, en calculant les valeurs qui sont réutilisées jusqu'à la valeur à obtenir.

L'algorithme 15 donne un programme dynamique de calcul de la suite de Fibonacci modifiée. Les valeurs successives de la suite de Fibonacci sont stockées dans le tableau *Fiboal*.

Algorithme 15 : Fonction de calcul de la suite de Fibonacci avec mémoïsation

```
1 Fonction FiboDyn(n)
2 FiboVal[] : tableau d'entiers, de dimension n
3 début
4     FiboVal[1] ← 1
5     FiboVal[2] ← 1
6     pour i ← 3 à n faire  FiboVal[i] ← FiboVal[i − 1] + FiboVal[i − 2]
7     retourner FiboVal[n]
```

La complexité de cet algorithme dépend de la boucle for et elle est linéaire, en $O(n)$. Elle est donc très faible par rapport à l'algorithme récursif.

Il est important de remarquer que, d'une manière générale, la mémoïsation compense la complexité algorithmique par une utilisation mémoire, en mémorisant les valeurs intermédiaires dans un tableau. Si la taille du problème considéré augmente, la taille du tableau de la solution avec mémoïsation augmente en même temps, parfois de manière non linéaire, ce qui peut conduire à une saturation de la mémoire.

À noter tout de même que la mémoïsation n'implique pas forcément d'abandonner la programmation récursive. Il est en effet possible de tester les calculs qui ont déjà été réalisés pour éviter de les recalculer. Par exemple, dans le calcul de la suite de Fibonacci, il est possible d'initialiser toutes les valeurs du tableau à 0 avec de commencer le calcul puis, de limiter les appels récursif aux cas où la valeur correspondante dans le tableau est à 0.

Le calcul des termes de la suite de Fibonacci ne constitue pas un problème d'optimisation. Ce n'est donc pas à proprement parlé un problème de programmation dynamique, même s'il utilise la technique de mémoïsation qui a été popularisée par la programmation dynamique. Comme nous l'avons dit précédemment la programmation dynamique aborde la recherche de solutions optimales lorsqu'une qu'une solution peut-être calculée à partir de sous-solutions. Nous illustrons ce cas dans la section suivante avec l'exemple du plus court chemin dans un graphe.

3.3 Illustration de la programmation dynamique

Comme nous l'avons dit la programmation dynamique repose sur le principe d'optimalité de Bellmann. Ce principe correspond aux problèmes d'optimisation dont la solution finale repose sur le calcul des solutions optimales des sous-problèmes. Pour pouvoir l'appliquer, il faut donc que le problème soit compatible avec ce principe.

Pour illustrer l'approche de la programmation dynamique nous traitons le problème de la recherche de la plus courte distance entre deux sommets d'un graphe. Le problème du calcul de la plus courte distance est bien un problème d'optimisation puisque nous cherchons, parmi plusieurs solutions possibles, la solution avec la plus petite valeur : le calcul vise à minimiser la distance parcourue entre le sommet de départ u et le sommet d'arrivée v.

Une solution naïve, ou “brute force”, au problème serait de considérer tous les chemins possibles entre u et v et de calculer, pour chaque chemin la distance qu'il parcourt. Le nombre de chemins entre deux sommets dépend du nombre n de sommets du graphe et du nombre maximum m de voisins pour chaque sommet. Au pire cas il y a donc m^n solutions et il n'est donc pas envisageable de les calculer toutes pour des graphes de taille importante. Il faut donc chercher un algorithme qui réalise le travail de manière plus efficace.

3.3.1 Structure de la solution optimale (1)

Comme dit précédemment, nous pouvons remarquer que le plus court chemin entre le sommet u et le sommet v passe forcément par l'un des sommets voisins de v. Si nous connaissons les distances optimales de u à chacun des voisins de v, il est alors facile de calculer la plus courte distance de u à v en ajoutant la distance entre le voisin courant et v à la distance entre u et le voisin courant. Cela se traduit par la formule suivante :

$$d(u, v) = \min_{w \in \Gamma(v)} (d(u, w) + d(w, v))$$

Où $\Gamma(v)$ est l'ensemble des voisins de v.

À noter que si les solutions optimales des sous-problèmes ne permettent pas de déduire la solution optimale du problème alors il n'est pas possible d'appliquer la programmation dynamique. Pour pouvoir appliquer la programmation dynamique, il est donc important de savoir montrer que le principe de Bellmann s'applique bien au problème d'optimisation considéré.

3.3.2 Solution récursive (2)

Comme nous le voyons dans l'exemple, l'expression de la solution optimale finale en fonction des solutions optimales des sous-solutions conduit implicitement à une formulation, et donc une programmation, récursive. Ainsi, à partir de la formule précédente, nous pouvons implanter le calcul de la plus courte distance entre deux sommets u et v, ce qui donne l'algorithme récursif 16.

La réalisation de la solution récursive permet de voir si, dans les sous-solutions calculées, certaines sont employées plusieurs fois à des endroits différents du calcul. Cela se traduit par plusieurs appels à la fonction avec les mêmes paramètres, donc plusieurs fois le même calcul. Ceci peut être illustré par notre cas d'exemple où le calcul de la plus courte distance entre u et chacun des voisins de v peut utiliser plusieurs fois la même valeur comme illustré par la figure 3.2. Sur cette figure nous voyons que le plus court chemin entre le sommet A et le sommet L passe forcément par l'un des voisins de L : K, G ou H, donc le calcul de la plus courte distance entre le A et L utilise forcément le calcul de la plus courte distance et l'un des trois voisins. Or le calcul des plus courtes distances de A à K et de A à G utilisent tous les deux la plus courte distance entre A et J. De même le calcul de la plus courte distance entre de A à G et de A à H utilisent tous les deux la plus courte distance entre A et F.

Algorithme 16 : Calcul de la plus courte distance par récurrence

```
1 Fonction PCDRec(u, v)
2 d[][] : la matrice des distances
3 début
4     si u = v alors retourner 0
5     pcd ← ∞
6     pour chaque sommet w voisin de v faire
7         si pcd > PCDRec(u, w) + d(w, v) alors
8             pcd ← PCDRec(u, w) + d(w, v)
9     retourner pcd
```

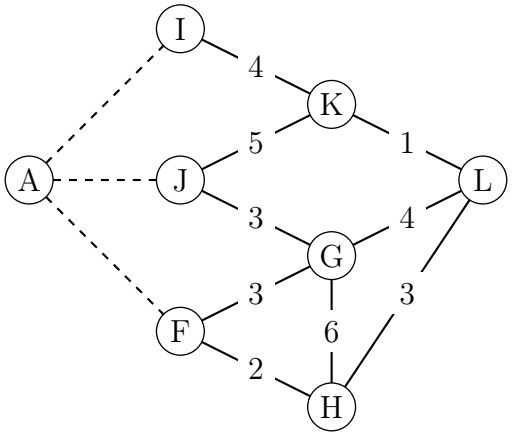


FIGURE 3.2 – Exemple de graphe avec des distances

Faire plusieurs fois le même calcul est évidemment inefficace. Dans certains cas, comme nous l'avons vu pour le calcul de la suite de Fibonacci, cela peut même conduire à des performances catastrophiques. Dans notre cas, la complexité de l'algorithme récursif est exponentielle du fait de ces calculs redondants.

3.3.3 Solution en programmation dynamique (3)

Pour éviter le recalcul de sous-solutions, la programmation dynamique a recours à la mémoïsation pour éviter le recalcul des sous-solutions de la programmation récursive : les valeurs des sous-solutions sont mémorisées dans un tableau, sur le modèle de ce qui a été fait dans l'algorithme 15.

Pour le calcul de la plus petite distance, l'approche par programmation dynamique utilise donc une approche ascendante de calcul, où l'algorithme commence par explorer les sommets le plus proches du sommet de départ et utilise les distances parcourues pour calculer les sommets plus éloignés, contrairement à la programmation récursive qui utilise une méthode descendante. Les distances calculées sont mémorisées au fur et à mesure de la progression dans un tableau pour resservir dans les itérations suivantes.

L'algorithme 17 utilise cette approche pour calculer la plus petite distance entre deux

sommets. Il réalise n itérations ce qui lui permet d'être sûr de couvrir tous les les sommets du graphe mais peut aussi s'arrêter avant, si la file qu'il utilise est vide. A l'aide d'une file l'algorithme parcourt l'ensemble des chemins qui relient le sommet u au sommet v, sur le même principe que le parcours en largeur.

Algorithme 17 : Calcul de la plus courte distance entre deux sommets d'un graphe

Données : n : le nombre de sommets du graphe
$d[][]$: la matrice des distances
f : file des sommets en cours **initialisée à** file vide
Résultat : $pcd[v]$: la plus courte distance de u à v, $\forall v \in V$

```
1  Fonction PCDDyn(u, v)
2  début
3      pcd[u] ← 0
4      pour chaque sommet w ≠ u de V faire pcd[w] ← +∞
5      nIter ← 1 /*                nombre d'itérations                */
6      f.add(u)
7      tant que (nIter < n) ∧ (¬f.empty()) faire
8          somCourant ← f.poll()
9          pour chaque sommet w voisin de somCourant faire
10             si pcd[w] > pcd[somCourant] + d(somCourant, w) alors
11                 pcd[w] ← pcd[somCourant] + d(somCourant, w)
12             if ¬f.contains(w) then f.add(w)
13         nIter ← nIter + 1
14     retourner pcd[v]
```

La complexité de cet algorithme est en $O(nm)$ puisque nous avons deux boucles imbriquées, la première dépendant du nombre de sommets et la seconde du nombre d'arêtes.

Comme nous pouvons le constater dans l'algorithme 17, il y a de fortes chances pour que le calcul de la plus courte distance de u à v permette de trouver des plus courtes distances entre u et un plus grand nombre de sommets que simplement v. De plus cet algorithme permet de calculer la distance mais ne donne pas le plus court chemin. L'algorithme de Bellman-Ford, aussi appelé algorithme à correction d'étiquettes, que nous voyons dans la suite, permet le calcul à la fois des plus courtes distances entre un sommet u et tous les sommets v d'un graphe et des chemins eux-mêmes. Il permet donc de construire la solution optimale.

3.3.4 Algorithme de Bellmann - Ford

L'algorithme présenté ici permet de rechercher des plus courts (des plus longs chemins) à partir d'un sommet dans le cas d'arêtes de longueur et de signe quelconque, à condition qu'il ne contienne pas de circuit absorbant (circuit dont la somme des pondérations des arêtes est négative). Cette approche pour résoudre de manière efficace le problème en utilisant la programmation dynamique a été proposée en même temps par les deux

chercheurs Bellmann et Ford et porte donc leur deux noms. L'approche a ensuite été utilisée avec succès sur de nombreux problèmes.

Principe de l'algorithme On recherche des plus courts chemin d'origine unique u :
— soit $d(x, y)$ la longueur de l'arc (x, y) ;
— $pcd(v)$ est la plus courte distance provisoire de u au sommet v.
L'initialisation de l'algorithme est la même que celle de l'algorithme de Dijkstra. Le principe de comparaison d'une distance temporaire à celle d'autres origines est également identique. La recherche se fait sur la base des arêtes qui arrivent à un sommet avec une correction de l'étiquette $pcd(v)$ dès qu'une meilleure solution est trouvée. Lorsque plus aucune longueur $pcd(v)$ n'est modifiée, alors l'algorithme peut s'arrêter. Au pire il faut n itérations pour calculer toutes ces distances, comme dans l'algorithme précédent.

Algorithme 18 : Algorithme de Bellman-Ford, plus court chemin dans un graphe $G(V, E)$

Données : n : le nombre de sommets du graphe
$d(x, y)$: la longueur de l'arête $\{x, y\}$, $\forall x, y \in V$
Résultat : $pcd[x]$: la plus courte distance de x_0 à x, $\forall x \in V$
$prec[x]$: le sommet précédent x sur le plus court chemin de u à x

```
1  Fonction plusCourtChBellmanFord(G, u)
2  début
3      pcd[u] ← 0
4      prec[u] ← −1
5      pour chaque sommet x ≠ u de V faire
6          pcd[x] ← +∞
7          prec[x] ← −1
8      nIter ← 1 /*                  nombre d'itérations                  */
9      modif ← VRAI /*   VRAI si une valeur de pcd[y] a été modifiée   */
10     tant que (nIter < n) ∧ modif faire
11         modif ← FAUX
12         pour chaque arête {x, y} de G faire
13             si pcd[y] > pcd[x] + d(x, y) alors
14                 pcd[y] ← pcd[x] + d(x, y)
15                 prec[y] ← x
16                 modif ← VRAI
17         nIter ← nIter + 1
```

Le chemin le plus court entre le sommet u et le sommet v est alors trouvé en remontant le tableau des prédécesseurs à partir du sommet v.

L'algorithme générique 18 illustre l'idée donnée ici pour les plus courts chemins. L'algorithme peut aussi bien être utilisé pour calculer les plus longs chemins si le graphe ne possède pas de cycle en remplaçant l'initialisation $pcd[x] \leftarrow +\infty$ par $pcd[x] \leftarrow -\infty$ et en changeant la comparaison $pcd[y] > pcd[x] + d(x, y)$ par $pcd[y] < pcd[x] + d(x, y)$.

A l'issue du premier calcul, il est possible de compléter l'algorithme pour vérifier si le

graphe ne possède pas de circuit absorbant. Cela se fait en ajoutant une boucle supplémentaire parcourant l'ensemble des arcs. Si ce parcours génère une modification d'une valeur trouvée à l'issue de l'algorithme alors le graphe possède un circuit absorbant.

Illustration Pour illustrer cet algorithme nous proposons de calculer les plus courtes distances entre le sommet J du graphe donné à la figure 3.3 et les autres sommets du graphe, en déroulant l'algorithme de Bellmann-Ford.

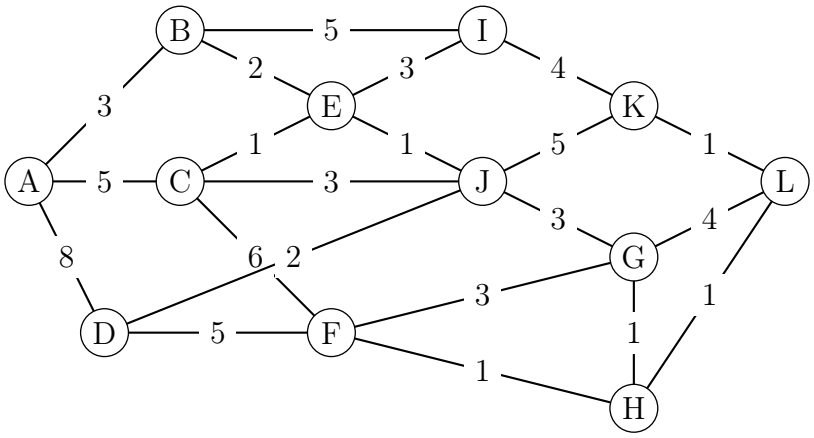


FIGURE 3.3 – Exemple de graphe avec des distances

Le tableau $d(x, y)$ est donc défini de la manière suivante :

Après l'initialisation, les tableaux *pcd* et *prec* sont donc dans l'état suivant :

Sommet	A	B	C	D	E	F	G	H	I	J	K	L
pcd	∞	∞	∞	∞	∞	∞	∞	∞	∞	0	∞	∞
prec	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Parmi les arêtes considérées à la première itération, seule les arêtes issues de du sommet J changent des valeurs puisque les sommets non accessibles n'ont pas de valeur $d(x, y)$.

Sommet	A	B	C	D	E	F	G	H	I	J	K	L
pcd	∞	∞	3	2	1	∞	3	∞	∞	0	5	∞
prec	-1	-1	J	J	J	-1	J	-1	-1	-1	J	-1

La deuxième itération permet l'accès à tous les sommets. Les valeurs de distances ne sont pas encore toutes finalisées.

Sommet	A	B	C	D	E	F	G	H	I	J	K	L
pcd	8	2	2	2	1	6	3	4	4	0	5	6
prec	C	E	E	J	J	G	J	G	E	-1	J	K

Après la troisième itération, le sommet I est couvert et les distances aux sommets A et L sont modifiées.

Sommet	A	B	C	D	E	F	G	H	I	J	K	L
pcd	5	2	2	2	1	5	3	4	7	0	5	5
prec	B	E	E	J	J	H	J	G	E	-1	J	H

Après cette troisième itération il n'y a plus de changement dans les tableaux et l'algorithme s'arrête.

A partir du tableau *prec* nous pouvons retrouver par exemple le chemin le plus court entre A et A : le prédécesseur de A est B, le prédécesseur de B est E et le prédécesseur de E est J. Le chemin est donc J, E, B, A.

3.4 Généralisation

Sur cet exemple nous pouvons constater que la programmation dynamique apporte une solution élégante et beaucoup plus efficace qu'une solution naïve.

D'une manière plus générale, la difficulté de la programmation dynamique réside dans la mise en évidence et la preuve que la solution optimale peut être calculée à partir des sous-solutions optimales (principe d'optimalité de Bellmann), car tous les problèmes combinatoires n'admettent pas de sous-structure optimale, indispensable pour mettre en œuvre la programmation dynamique. Une fois cette preuve faite, la conception d'un algorithme de programmation dynamique se divise en quatre grandes étapes :

1. caractériser la structure d'une solution optimale. Il s'agit de montrer que le problème possède une sous-structure optimale. C'est-à-dire que le problème initial peut-être exprimé en fonction des sous-solutions et que la solution optimale du problème ne dépend que des solutions optimale des sous-problèmes.
2. définir récursivement la valeur d'une solution optimale. C'est l'expression formelle (sous une forme mathématique) de la solution optimale en fonction des sous-solutions.
3. calculer la valeur d'une solution optimale en remontant progressivement jusqu'à l'énoncé du programme initial (calcul des sous-solutions), en mémoïsant les résultats.
4. construire une solution optimale pour les informations calculées.

Les 3 premiers points sont incontournables pour la résolution du problème posé, c'est-à-dire trouver la valeur optimale. En revanche, le point 4 n'a d'intérêt que lorsque construire la solution optimale est intéressant en plus de sa valeur, c'est-à-dire, dans le cas de la plus courte distance, trouver le chemin le plus court.

La mise en évidence d'une solution par programmation dynamique dépend du problème considéré. Il n'y a donc pas de procédure générique pour trouver une programmation dynamique à un problème. Néanmoins certains indices permettent de suspecter la possibilité d'une telle solution : la complexité élevée du problème et le recalcul d'un nombre important de sous-solutions. Pour finir il faut noter que la diminution de la complexité en temps de calcul de la programmation dynamique est obtenue par une augmentation de la taille des données puisqu'il faut stocker les solutions intermédiaires. C'est donc la taille mémoire nécessaire qui peut devenir bloquante dans la réalisation d'un calcul.

3.5 Synthèse

À l'issue de ce chapitre, et après avoir traité les exercices qui vont avec, vous devez être capables de :

- Comprendre le champ d'application de la programmation dynamique ;
- Proposer une approche de programmation dynamique à un problème ayant une expression récursive conduisant à une complexité exponentielle ;
- Concevoir un algorithme de programmation dynamique ;

Chapitre 4

Programmation gloutonne

4.1 Introduction

Nous avons vu qu'un problème d'optimisation combinatoire est un problème dans lequel l'exploration des solutions, pour trouver la meilleure, est combinatoire. La génération de l'ensemble des solutions peut cepedant être très coûteuse. Il est donc nécessaire d'établir des stratégies pour réduire le nombre de solutions à explorer. La programmation dynamique vue au chapitre précédent, est une stratégie possible mais son utilisation peut-être surdimensionnée pour certains problèmes, voire non applicable si la structure du problème n'est pas adaptée. La programmation gloutonne peut alors parfois apporter une solution.

Pour un problème d'optimisation combinatoire, nous parlons de programmation gloutonne si le programme de résolution, celui qui recherche les solutions, utilise une stratégie qui repose sur un choix glouton : un choix qui optimise l'objectif en ne tenant compte que de la situation courante. Par exemple, dans la recherche d'un plus court chemin entre deux sommets d'un graphe dont les arêtes sont pondérées, un choix glouton consiste à prendre la meilleure solution courante, c'est-à-dire à prendre l'arête qui minimise localement le coût du chemin, donc l'arête ayant la plus faible pondération. La stratégie gloutonne procède de manière incrémentale, de choix en choix, en prennant la solution qui semble la meilleure au moment du choix, jusqu'à arriver à la solution finale. Attention une stratégie gloutonne peut permettre de trouver une solution optimale, dans certains cas, mais elle ne donne pas **forcément** une solution optimale. Seule une preuve permet alors de montrer que la stratégie trouve une solution optimale au problème.

L'intérêt de la programmation gloutonne est qu'elle repose sur des algorithmes ayant une complexité polynomiale. Lorsque l'algorithme est optimal, c'est-à-dire lorsqu'il trouve la solution optimale quelles que soient les valeurs qui caractérisent le problème, la solution gloutonne est particulièrement intéressante car généralement peu coûteuse. Ce qui ne veut pas non plus dire qu'il n'existe pas d'algorithme moins coûteux. Nous présentons le cas de l'arbre couvrant de poids minimal au début de ce chapitre pour illustrer ce propos.

Dans le cas de certains problèmes complexes, la programmation gloutonne peut apporter une solution heuristique, c'est-à-dire de calcul d'une solution sous-optimale mais de meilleure qualité qu'une solution quelconque. Ceci est illustré par le second exemple traité, le cas de la coloration de graphe.

Pour finir le cas du choix des activités nous permet d'illustrer que la manière de faire le choix glouton peut conduire à une solution optimale ou non.

4.2 Arbre couvrant de poids minimal

Pour illustrer la programmation gloutonne et étudier la démarche qui conduit à la mise en place d'une stratégie, nous prennons dans un premier temps un cas où la stratégie gloutonne permet à coup sûr de calculer une solution optimale. Le cas traité est la recherche d'un arbre couvrant de poids minimal. Nous rappelons que le problème de la recherche d'un arbre couvrant de poids minimal (ACPM), pour un graphe non-orienté connexe $G = (V, E)$ dont les arêtes sont pondérées tel que celui de la figure 4.2, consiste à trouver un sous-ensemble $T \subseteq E$ d'arêtes de G, acyclique, qui connecte tous les sommets de V et dont le poids total, la somme des pondérations des arêtes appartenant à l'arbre T est minimal. Formellement, si w est une fonction de pondération des arêtes du graphe, le poids $w(T)$ de l'arbre T est calculé avec : $w(T) = \sum_{(u,v)\in T} w((u, v))$. Par exemple, sur la figure 4.2 les arêtes en rouge forment un arbre couvrant de poids $w(T) = 36$. De manière assez évidente, cet arbre n'est pas de poids minimal.

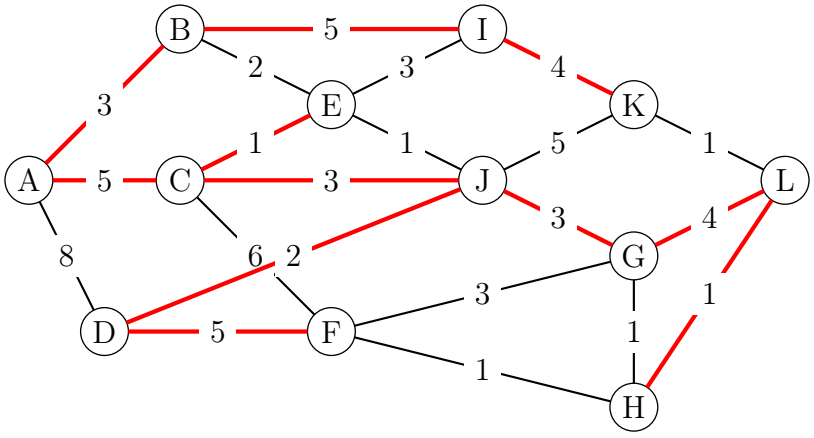


FIGURE 4.1 – Exemple de graphe non-orienté connexe avec des arêtes pondérées

Sans réflexion préliminaire, trouver la solution optimale à ce problème pourrait être réalisé en générant puis en évaluant tous les arbres couvrants possibles. La complexité de la génération de ces arbres est en $O(n!)$ au pire cas puisque, dans le cas d'un graphe complet, nous avons n étapes (une par sommet) avec n choix possibles. Il n'est donc pas envisageable de faire le calcul dès que le nombre de sommets est grand. Heureusement, dans le problème de l'ACPM, les conditions sont réunies pour qu'une stratégie gloutonne garantisse un résultat optimal. Cette stratégie consiste, à chaque étape, d'augmenter au minimum le poids total, qui est l'objectif global, en ajoutant l'arête de plus faible poids.

D'une manière générale une stratégie gloutonne s'appuie sur les propriétés du problème pour éviter de parcourir toutes les solutions possibles et les évaluer. Ainsi, pour le problème de l'ACPM, nous nous appuyons sur la propriété suivante : *Pour toute coupe d'un graphe non-orienté pondéré, si une arête est de poids strictement inférieur aux autres, alors elle appartient à l'arbre couvrant de poids minimal.* Cette propriété nous dit que

pour tout sommet, ou toute composante connexe du graphe, l'arête qui le relie à l'ACPM est celle qui a le plus petit poids. Cette propriété nous permet d'appliquer une stratégie gloutonne qui va choisir, à partir de composantes connexes, l'arête de poids minimal pour les connecter. Il suffit alors, à partir de l'ensemble des sommets V sans leur arêtes, de construire une solution incrémentale qui ajoute les arêtes identifiées comme faisant partie de l'ACPM. Sur cette base plusieurs algorithmes gloutons peuvent être conçus : les plus connus sont les algorithmes de Borůvka, Kruskal et de Prim.

L'algorithme de Borůvka est donné par l'algorithme 19. L'idée est d'identifier, par composante connexe d'une forêt, la plus petite arête qui la relie à une autre composante connexe (d'après la propriété précédente nous savons que cette arête fera partie de l'ACPM) puis de regrouper les deux composantes. L'algorithme commence avec une forêt constituée des sommets sans leur arêtes. Ensuite pour chacune des arêtes si ses extrémités ne font pas partie de la même composante connexe et que la pondération de l'arête est plus petite que celle actuellement enregistrée alors cette arête devient la plus petite arête.

Algorithme 19 : Algorithme de Borůvka : recherche de ACPM dans un graphe

```
1  Données :
2    Graphe G = (V, E)
3    w(x) : E → ℝ une fonction de pondération des arêtes
4    cc : tableau des composantes connexes
5    nbcc : entier, nombre de composantes connexes
6    ppa : tableau des plus petites arêtes par composante connexe
7    F : graphe /* F est une forêt                                   */
8
9  début
10   F ← (V, ∅)
11   cc ← composanteConnexeNOriente(F)
12   nbcc ← cc.longueur
13   tant que nbcc > 1 faire
14     pour i de 1 à nbcc faire ppa[i] ← ∞
15     pour e = {u, v} ∈ E faire
16       si cc[u] ≠ cc[v] alors
17         si w(e) < w(ppa(cc[u])) alors ppa(cc[u]) ← e
18         si w(e) < w(ppa(cc[v])) alors ppa(cc[v]) ← e
19     pour i de 1 à nbcc faire
20       si ppa[i] ≠ ∞ alors F ← F ∪ e
21     cc ← composanteConnexeNOriente(F)
22     nbcc ← cc.longueur
```

L'algorithme initial de Borůvka est conçu pour un graphe dont les arêtes ont des pondérations différentes. Si, à partir d'un sommet, deux arêtes ont le même poids, il faut définir un régle de choix entre ces arêtes pour faire fonctionner l'algorithme. La formulation donnée ici utilise la première arête sélectionnée. Il est également possible de faire une renumérotation de ces arêtes.

Une optimisation évidente de l'algorithme consiste à limiter les arêtes évaluées dans la boucle **pour** aux arêtes qui ne sont pas dans les composantes connexes. De la même manière il n'est pas nécessaire de recalculer les composantes connexes à chaque fois puisque nous savons quelles sont les composantes qui sont fusionnées une fois l'arête ajoutée à F. En tenant compte de ces optimisations la complexité de cet algorithme est de $O(M \times Log(N))$, où M est le nombre d'arêtes du graphe et N le nombre de sommets. Sa complexité est plus élevée si nous devons calculer les composantes connexes.

Il existe d'autres algorithmes optimaux de recherche d'un ACPM : les algorithmes de Kruskal et de Prim :

- L'algorithme de Kruskal, comme celui de Borůvka, travaille de façon incrémentale en regroupant des ensembles. Il considère donc une forêt dont il réduit le nombre d'arbres à chaque étape par réunion de deux arbres en un seul. La réduction se fait en parcourant la liste des arêtes triées par ordre de poids croissant ce qui évite la boucle **while** de l'algorithme de Borůvka. Cet algorithme est adapté aux graphes denses. Par l'utilisation de structures de données appropriées (ensembles disjoints), cet algorithme peut s'exécuter en $O(M \times Log(N))$.
- l'algorithme de Prim considère un seul arbre, qui grandit étape par étape jusqu'à recouvrir le graphe. Il part d'un sommet quelconque. À chaque étape, il ajoute à l'arbre en construction l'arête de poids minimal parmi celles qui permettent de rajouter un nouveau sommet à l'arbre. Quand il ne peut plus ajouter de nouveau sommet, c'est qu'ils sont tous connectés, et nous avons donc un arbre recouvrant. Cet algorithme est adapté aux graphes peu denses. En utilisant des files de priorité implantées efficacement, cet algorithme peut s'exécuter en $O(M + N \times Log(N))$.

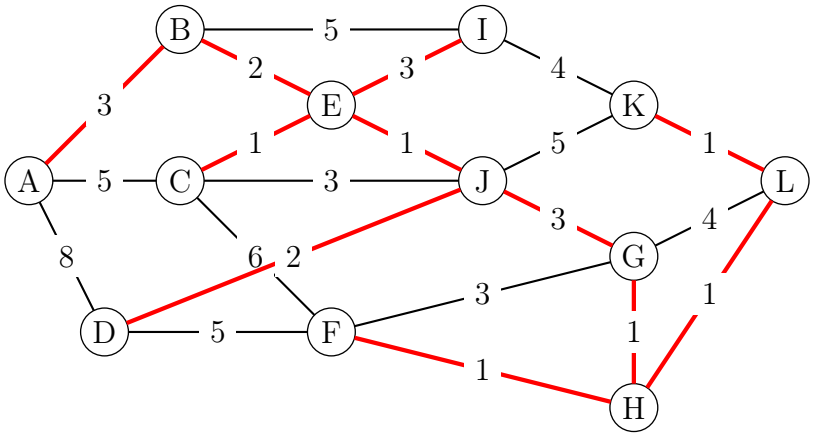


FIGURE 4.2 – ACPM trouvé par l'algorithme de Prim. $w(T) = 19$

À partir des complexités des algorithmes de Borůvka, Kruskal et de Prim nous pouvons remarquer que, si la propriété initiale sur la coupe d'un graphe pondéré nous permet de conclure qu'un algorithme glouton optimal (qui trouve une solution optimale) existe, il est encore nécessaire de travailler à la qualité de l'algorithme en question. Cette conclusion s'applique d'une manière générale aux algorithmes gloutons : une propriété peut permettre de réduire l'exploration mais c'est sa mise en œuvre qui déterminera la complexité de l'algorithme résultant.

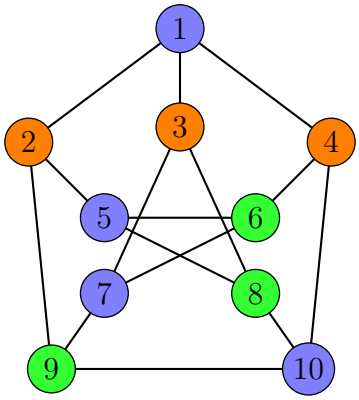


FIGURE 4.3 – Coloration gloutonne du graphe de Petersen à partir du sommet 1

4.3 Coloration de graphes

Dans notre second exemple nous considérons le problème de la coloration d'un graphe non orienté avec un nombre minimal de couleurs. Colorier un sommet consiste simplement à lui associer une couleur et colorier un graphe consiste à colorier chacun des sommets du graphe de telle sorte que deux voisins quelconques n'aient pas la même couleur. Le problème de trouver de trouver le nombre minimal de couleurs pour réaliser la coloration d'un graphe est un problème classique d'optimisation combinatoire. Sur la figure 4.3 un graphe de Petersen a été colorié avec 3 couleurs. Le problème de la minimisation consiste donc à savoir si cette coloration utilise un nombre minimal de couleurs, donc savoir s'il est possible de la faire avec 2 couleurs. Ce qui n'est évidement pas possible dans ce cas puisque certains sommets ont 3 voisins et la coloration est optimale.

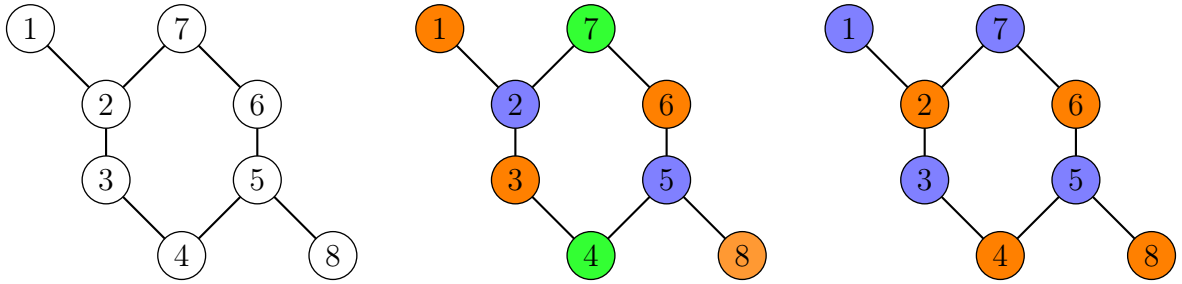
Dans le cas de la coloration minimale, une stratégie gloutonne peut consister à classer les sommets par ordre de degré décroissant et à attribuer à chaque sommet une couleur différente de la couleur de ses voisins ayant déjà une couleur. Pour garantir d'utiliser un nombre minimal de couleur, l'attribution se fait en définissant un ordre dans les couleurs possibles et en prenant la première couleur disponible dans cet ordre. Cette statégie est mise en œuvre par l'algorithme 20. Cet algorithme commence par trier les sommets par ordre croissant de degré (nombre de voisins) dans la lsite *lst*. Il utilise ensuite cette liste triée pour affecter les couleurs aux sommets : pour chaque sommet il prend la première couleur libre dans le tableau *couleur*.

Dans le cas du graphe de Petersen de la figure 4.3, les sommets ayant tous le même nombre de voisins, ils sont traités dans l'ordre des numéros de sommets croissants et avec un ordre des couleurs qui est : bleu, orange, vert. Dans ce cas le résultat est optimal, notre algorithme glouton obtient donc un bon résultat.

Cependant les algorithmes gloutons ne donnent pas forcément une solution optimale. C'est le cas pour la coloration du graphes de la figure 4.4a. L'algorithme glouton commence par le sommet 2, un des deux sommets ayant 3 voisins, et utilise le même ordre des couleurs que pour la coloration du graphe de Petersen. Le résultat de l'algorithme glouton est donné par la figure 4.4b. L'algorithme donne 3 couleurs alors que le graphe admet une coloration optimale à 2 couleurs comme le montre la figure 4.4c.

Algorithme 20 : Algorithme glouton de coloration d'un graphe $G = (V, E)$

```
1  Données :
2    Graphe G = (V, E)
3    couleurSom[] : tableau des couleurs des sommets, init à -1
4    couleur[] : tableau de couleurs
5    lst : file des sommets triés, init à ∅
6  début
       /* Génération de la liste triée des sommets                    */
7      pour (u, v) ∈ E faire Γ(u) + +; Γ(v) + +
8      pour v ∈ V faire l.add(v)
9      tant que ¬l.empty() faire
10       maxV ← −1
11       pour i ← 0 à l.size() faire
12         si |Γ(l.get(i))| > max alors
13           max ← |Γ(l.get(i))| ; som ← i
14       lst.add(som)
15       l.remove(som)
       /* Attribution des couleurs                                    */
16     tant que ¬lst.empty() faire
17       som ← lst.poll()
18       colored ← faux
19       color ← 0
20       tant que (¬colored) faire
21         used ← faux
22         pour chaque voisin v de som faire
23           si couleurSom[v] = color alors used ← vrai
24         si ¬used alors
25           couleurSom[som] ← couleur[color]
26           colored ← vrai
27         color ← color + 1
```



(a) Graphe non colorié (b) Coloration gloutonne (c) Coloration optimale

FIGURE 4.4 – Sous-optimalité de l'algorithme glouton de coloration

En fait le problème général de la coloration de graphes est connu pour être NP-Complet. Il n'est donc pas possible de trouver une solution optimale par un algorithme glouton. La solution gloutonne est cependant trouvée en un temps relativement court. Au delà, pour montrer les qualités de l'algorithme, il pourrait être nécesaire de montrer que les solutions trouvées sont toujours à une distance maximale de l'optimale mais ce n'est pas notre objectif ici.

Plus généralement, la programmation gloutone n'offre pas toujours des solutions optimales mais elle a un intérêt car elle repose sur des algorithmes simples, polynomiaux qui gérèrent des solutions sous-optimales mais généralement avec des résultats corrects. Par exemple dans le cas de la coloration nous ne pouvons pas garantir que la solution trouvée donne le plus petit nombre de couleurs mais le nombre trouvé reste généralement peu élevé. Nous parlons de solution heuristique.

4.4 Problème du choix d'activités

Le problème du choix d'activité permet d'illustrer le fait plusieurs choix gloutons, avec des critèes différents, peuvent exister pour un même problème mais qu'ils ne donnent pas tous le même résultat. Il souligne donc qu'il faut être attentif aux choix du critère utilisé.

4.4.1 Présentation du problème

Le problème du choix d'activité est un problème d'ordonnancement qui consiste à choisir un ensemble d'activités, chacune ayant une date de début et de fin fixe, avec la contrainte qu'il n'est pas possible de faire deux activités en même temps. C'est, par exemple, le cas d'une personne qui doit choisir les activités qu'elle peut faire dans une journée parmis un ensemble d'activités proposées à des horaires fixés. L'objetcif est de pouvoir faire le plus grand nombre d'activités dans la période donnée.

Les données de ce problème sont les suivantes :

- Soit $S = \{a_1, \ldots, a_n\}$ un ensemble de n activités ;
- Chaque activité a_i doit être traitée pendant une plage de temps, définie par sa date de début d_i et sa date de fin f_i avec $(0 \leq d_i < f_i)$;
- Les activités sont traitées par une ressource unique qui est partagée entre les

différentes activités et les traite une par une ;
— Les activités a_i et a_j sont dites compatibles si $f_i \leq d_j$ ou $f_j \leq d_i$;
Il s'agit alors de choisir le sous ensemble S d'activités compatibles le plus grand possible.

4.4.2 Exemple

Soit l'ensemble des activités S donné par la figure 4.5 et dont le tableau 4.1 synthétise les valeurs d_i, f_i et l_i (longueur de l'activité) pour des activités a_1 à a_{14}.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
d_i	1	3	0	5	3	5	6	8	8	2	12	13	15	16
f_i	4	5	6	7	8	9	10	11	12	13	13	16	17	19
l_i	3	2	6	2	5	4	4	3	4	11	1	3	2	3

TABLE 4.1 – Date de début et de fin des activités

Sur la figure il est facile de pouvons constater que :
— Le sous ensemble $\{a_3, a_9, a_{11}, a_{13}\}$ est correct mais n'est pas le plus grand possible ;
— Par contre $\{a_1, a_4, a_8, a_{11}, a_{12}, a_{14}\}$ est plus grand (taille maximale) ;
— Et $\{a_2, a_4, a_9, a_{11}, a_{12}, a_{14}\}$ est un autre sous ensemble de taille maximale, représenté en rouge sur la figure.

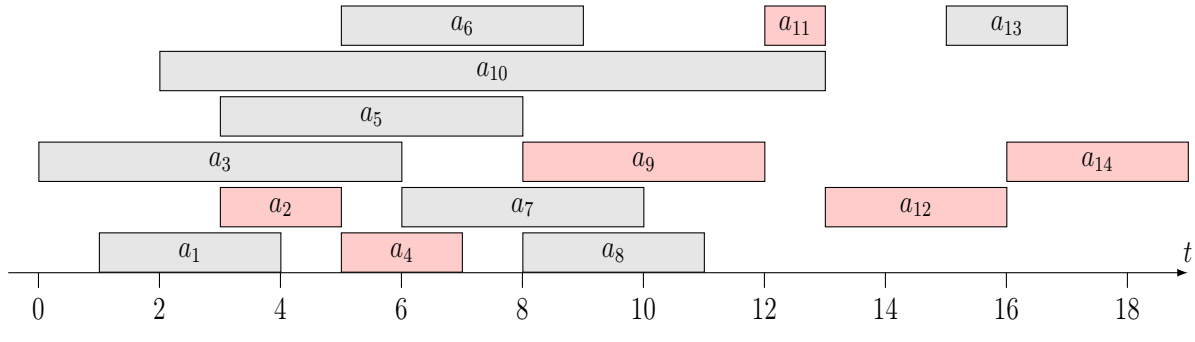


FIGURE 4.5 – Représentation de l'ensemble S des activités de la table 4.1 dans le temps

4.4.3 Choix du critère

Nous montrons ici que le choix du critère glouton, celui pour lequel nous allons prendre la plus grande, ou plus petite, valeur a un impact sur les résultats de l'algorithme glouton.

Sélection par la durée de l'activité

Un premier critère qui peut être utilisé est la durée des activités. En effet la durée des activités n'est pas considérée dans le résultat final et, en choisissant les activités les plus courtes en premier, nous pouvons espérer laisser plus de temps pour ajouter d'autres

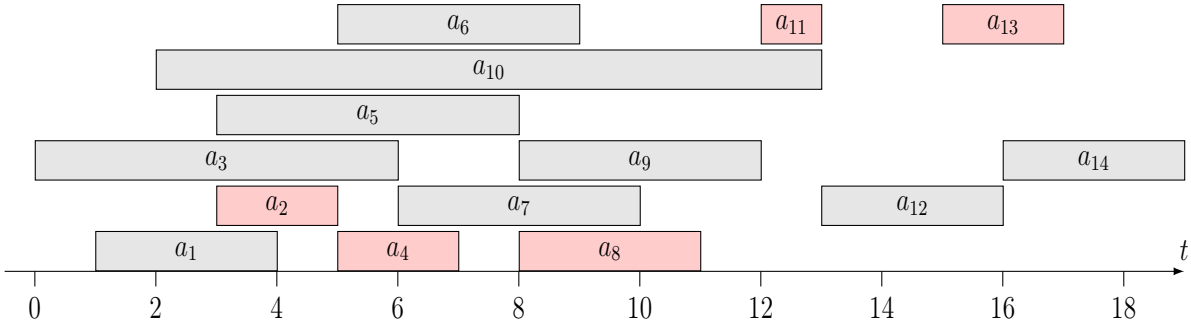


FIGURE 4.6 – Activités sélectionnées par le critère de durée des tâches

activités. L'algorithme correspondant trie la liste des activités par durée croissante puis parcourt cette liste en choisissant une activité dès qu'elle est compatible avec le sous-ensemble déjà constitué.

La figure 4.6 montre en rouge les activités sélectionnées par l'algorithme utilisant le critère de durée de l'activité. Nous obtenons $S = \{a_{11}, a_2, a_4, a_{13}, a_8\}$ donc 5 activités alors que nous avons vu précédemment qu'il existe des ensembles possibles à 6 activités. Le fait que, au moins sur cet exemple l'algorithme ne donne pas un résultat optimal prouve qu'il n'est pas optimal en général. Il serait en effet facile de mettre en place une preuve par l'absurde où nous supposons que l'algorithme donne toujours un résultat optimal puisque nous pouvons produire un cas où il ne l'est pas.

Sélection par la date de début d_i

Sélectionner les activités par leur date de début revient à toujours prendre la première tâche prête. Nous réduisons le temps perdu entre deux activités et espérons ainsi pouvoir en réaliser le plus possible. L'algorithme correspondant trie la liste des activités par date de début puis parcourt cette liste en choisissant une activité dès qu'elle est compatible avec le sous-ensemble déjà constitué.

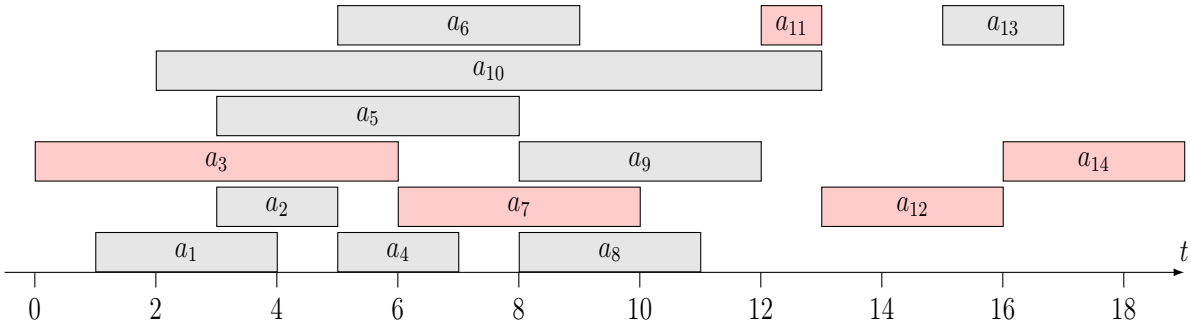


FIGURE 4.7 – Représentation de l'ensemble S des activités dans le temps

La figure 4.7 montre en rouge les activités sélectionnées par l'algorithme utilisant le critère de date de début de l'activité. Nous obtenons $S = \{a_3, a_7, a_{11}, a_{12}, a_{14}\}$ donc encore seulement 5 activités. L'algorithme n'est pas donc pas non plus optimal.

D'autres critères, comme le nombre d'incompatbilités, pourraient aussi être évalués qui

conduirait aussi à des résultats non optimaux. Nous nous intéressons maintenant au critère qui nous permet de trouver les valeurs optimales.

Sélection par date de fin

En sélectionnant les activités par date de fin, pour prendre l'activité qui termine au plus tôt, nous espérons laisser un maximum de temps disponible pour prendre d'autres activités.

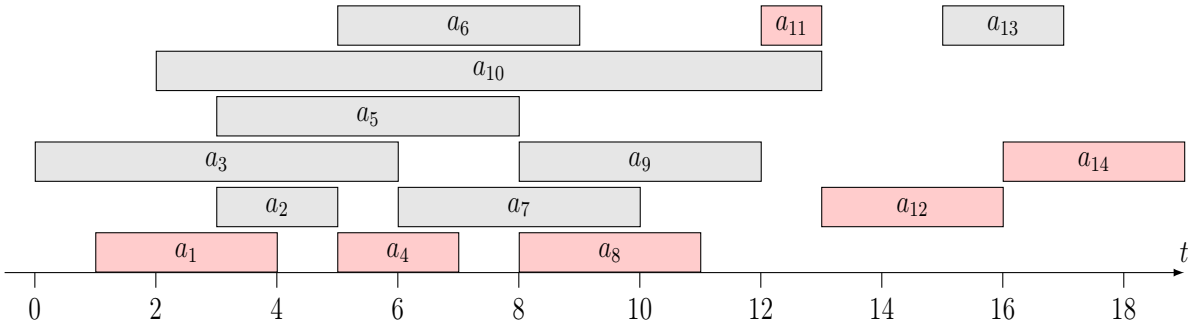


FIGURE 4.8 – Représentation de l'ensemble S des activités dans le temps

La figure 4.8 montre en rouge les activités sélectionnées par l'algorithme utilisant le critère de date de fin de l'activité. Nous obtenons $S = \{a_1, a_4, a_8, a_{11}, a_{12}, a_{14}\}$ donc 6 activités, qui est le nombre optimal. Attention cependant, ce n'est parce que l'algorithme arrive dans ce cas au nombre optimal d'activité qu'il est optimal, c'est-à-dire qu'il trouve toujours le nombre optimal d'activité. Il est donc nécessaire d'établir la preuve de l'optimalité de l'algorithme, ce que nous faisons dans la suite.

Établir une preuve de l'optimalité de l'algorithme nécessite de montrer que, quelque soit l'ensemble S, l'algorithme trouve toujours la meilleure solution. Nous supposons que l'ensemble des activités $S = \{(d_1, f_1), (d_2, f_2), \ldots, (d_n, f_n)\}$ est trié par ordre croissant des dates de fin. La preuve se fait en deux étapes : d'abord nous montrons que la solution optimale contient la première activité de la liste triée et, ensuite, nous montrons qu'une solution optimale du sous-problème de choix des activités qui commence à la fin de la première activité est une solution optimale du problème si on lui ajoute cette premmière activité.

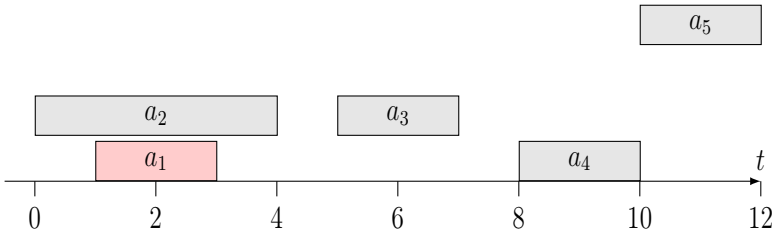


FIGURE 4.9

La première étape de la preuve consiste donc à montrer qu'il existe une solution optimale contenant l'activité qui termine au plus tôt (d_1, f_1), le premier élément de la solution

retournée par l'algorithme glouton. Pour cela, nous considérons une solution optimale X. Si X contient (d_1, f_1) nous avons ce que nous voulons. Sinon, nous sommes dans la situation illustrée par la figure 4.9 où nous avons représenté en rouge l'activité (d_1, f_1) qui, puisque nous avons trié la liste par date de fin au plut tôt, se termine nécessairement avant la première activité de la solution optimale X. Nous pouvons alors échanger le premier élément de X avec (d_1, f_1) sans nuire à l'optimalité car cet échange ne modifie pas le nombre d'éléments de X, et dans la mesure où (d_1, f_1) se termine avant la première activité, cette substitution ne perturbe pas le reste de la solution.

Nous montrons ensuite que, si X est une solution optimale contenant (d_1, f_1), alors $X_1 = X - (d_1, f_1)$ est une solution optimale pour le sous-problème du choix d'activités commençant après la date f_1. Pour cela nous procédons par l'absurde : nous supposons que X_1 n'est pas une solution optimale pour ce sous-problème et nous considérons X_2 une solution qui l'est.

A partir de X_2 nous construisons $X_3 = X_2 + (d_1, f_1)$. Cet ensemble X_3 est une solution du problème initial car, par définition X_2, ne contient que des activités commençant après la date f_1, donc qui ne sont pas incompatibles avec les activités de X_2. Nous allons maintenant compter les éléments de ces ensembles. D'une part, nous avons, $|X_1| = |X| - 1$, le nombre d'éléments de l'ensemble X_1 est inférieur de 1 au nombre d'éléments de l'ensemble X car X_1 est obtenu à partir de X en lui retirant un élément. D'autre part $|X_3| = |X_2|+1$ car X_3 est obtenu en ajoutant un élément à X_2. Puisque X_2 est une solution optimale au sous-problème et que X_1 ne l'est pas, hypothèse de notre raisonnement par l'absurde, alors il y a plus d'activités dans X_2 que dans X_1 : $|X_2| > |X_1|$. Puisque $|X_1| = |X| - 1$, nous avons $|X_2| > |X| - 1$. En ajoutant 1 à chacun des termes de cette inégalité et en utilisant la seconde égalité nous trouvons $|X_3| > |X|$. X_3 est donc une solution au problème initial comportant plus d'éléments que X, ce qui est absurde car X est une solution optimale de ce problème.

Nous avons donc montré les deux points suivants : (1). Il existe une solution optimale contenant (d_1, f_1), (2). Si X est une solution optimale contenant (d_1, f_1), alors $X_1 = X - (d_1, f_1)$ est une solution optimale pour le sous-problème du choix d'activités commençant à la date f_1. Nous pouvons alors réitérer cet argument et montrer de proche en proche, en réduisant au fur et à mesure la taille de l'ensemble, que la solution gloutonne est une solution optimale. Puisque la liste des activités est triées, nous nous contentons donc de passer en revue toutes les activités une seule fois et la complexité est linéaire.

Les algorithmes 21 et 22 donnent respectivement une solution récursive et itérative pour la résolution de ce problème du choix des activités avec le recourt de la programmation gloutonne. Le problème à résoudre est $S_{0,n+1}$. Étant donné la manière dont sont triées les activités, les algorithmes cherchent les activités mutuellement compatibles en commençant par la première, puis en examinant les activités les unes après les autres jusqu'à la dernière. Dans les deux cas, si on ne parle pas de la phase de tri des activités pour lequel il existe un algorithme en $O(n \log n)$, la complexité du choix du plus grand ensemble d'activités mutuellement compatibles a une complexité en $O(n)$ dans les deux mises en œuvre, avec n le nombre d'activités.

Algorithme 21 : Algorithme récursif glouton choisissant un nombre optimal d'activités mutuellement compatibles parmi un ensemble d'activités ordonnées

```
   /* Les listes sont triées par ordre croissant des dates de fin     */
 1 s : liste des dates de début des activités
 2 f : liste des dates de fin des activités
 3 Fonction activiteChoixGloutonRec(s, f, i, P)
 4   i : numéro de la dernière activité choisie
 5 début
 6     m ← i + 1
       /* Recherche de la première activité compatible avec a_i        */
 7     tant que (m ≤ s.size()) ∧ (s[m] < f[i]) faire m ← m + 1
 8     si m ≤ s.size() alors
 9         activiteChoixGloutonR(s, f, m, P)
10         retourner P.push(a_m)
11     sinon retourner ∅
12 Fonction activiteChoixGlouton(s, f)
13   A : liste donnant la solution gloutonne
14   P : pile d'activités
15 début
16     A.add(0) /* La première activité (0) est toujours choisie      */
17     activiteChoixGloutonRec(s, f, 0, P)
18     tant que ¬P.empty() faire A.add(P.pop())
19     retourner X
```

Algorithme 22 : Algorithme itératif glouton choisissant un nombre optimal d'activités mutuellement compatibles parmi un ensemble d'activités ordonnées

```
   /* Les listes sont triées par ordre croissant des dates de fin     */
 1 s : liste des dates de début des activités
 2 f : liste des dates de fin des activités
 3 Fonction activiteChoixGloutonI(s, f)
 4 début
 5     n ← s.size()
 6     A.add(0) /* La première activité (0) est toujours choisie      */
 7     i ← 0
       /* Recherche la première activité compatible avec a_i           */
 8     pour m de 1 à n − 1 faire
 9         si f[i] ≤ s[m] alors
10             A.add(a_m)
11             i ← m
12     retourner A
```

4.5 Èléments de stratégie gloutonne

Nous l'avons vu, le choix d'une solution optimale à partir de choix qui semblent les meilleurs à un instant donné peut être une approche, elle ne conduit pas toujours à une solution globalement optimale. Pour établir le cadre de l'utilisation d'un algorithme glouton pour résoudre un problème de manière optimal, il est alors nécessaire de montrer qu'il conduit inévitablement à une solution optimale.

4.5.1 Étapes de conception

Pour montrer qu'une approche gloutonne conduit conduit inévitablement à une solution optimale, il est indispensable de trouver une sous structure avec un choix glouton qui ne laisse plus qu'un sous problème à résoudre de manière optimal. On en déduit les étapes de conception d'un algorithme glouton :

1. ramener le problème d'optimisation à un problème avec un choix qui ne donne qu'un seul sous problème à résoudre ;
2. prouver qu'il y a toujours une solution optimale du problème initial qui fasse le choix glouton (la solution donnée par le choix glouton est peut être une parmi toutes les solutions optimales du problème initial) ;
3. prouver qu'après un choix glouton, la solution optimale du sous problème associée au choix glouton que l'on a fait, conduit bien à la solution optimale du problème initial.
4. construire un algorithme.

Il est difficile de savoir s'il existe une résolution gloutonne d'un problème avant de se poser ces questions.

4.5.2 Propriétés du choix glouton

Propriété 4.1

On peut arriver à une solution globalement optimale en effectuant un choix localement optimal (sans tenir compte des résultats des sous problèmes, approche descendante).

Ceci montre la différence avec la programmation dynamique qui part des sous problèmes pour faire le choix optimal du problème de taille supérieure (approche ascendante).

Pour montrer qu'on se trouve bien dans cadre de cette propriété, on peut utiliser l'astuce du théorème montré dans l'exemple du choix des activités. L'idée est de montrer une solution globalement optimale d'un sous problème, de la modifier pour le choix glouton et enfin de monter qu'elle conduit à un problème similaire de taille plus petite. Souvent un pré-traitement est nécessaire sur les données en entrée, comme ici pour l'exemple du choix des activités. Cela est encore le cas pour le calcul d'un arbre de recouvrement de poids minimal avec l'algorithme de Prim.

4.5.3 Sous structure optimale

Propriété 4.2

Une sous structure est optimale si une solution optimale du problème contient les solutions optimales des sous problèmes.

Cette propriété est valable à la fois pour la programmation dynamique et gloutonne. Pour une résolution gloutonne, on suppose que nous sommes arrivés à un sous problème en ayant fait le choix glouton dans le problème original. Ce qu'il faut prouver, c'est que la solution optimale du sous problème associé au choix glouton fait avant, donne une solution optimale au problème original. Il s'agit d'une récurrence sur les sous problèmes.

4.5.4 Programmation gloutonne et programmation dynamique

Programmation gloutonne et dynamique sont elles interchangeables ? Ces deux stratégies de résolution utilisent la même propriété sur la sous structure optimale. Il paraît donc possible d'adopter une mise en œuvre ascendante (programmation dynamique) pour résoudre un problème pour lequel la stratégie gloutonne est valide. Par contre, l'inverse n'est pas vrai, dès lors qu'il est nécessaire de faire un choix entre plusieurs alternatives.

4.6 Synthèse

À l'issue de ce chapitre, et après avoir traité les exercices qui vont avec, vous devez être capables de :

- Comprendre le champ d'application de la programmation gloutonne ;
- Proposer une approche de programmation gloutonne à un problème ;
- Concevoir un algorithme de programmation gloutonne ;

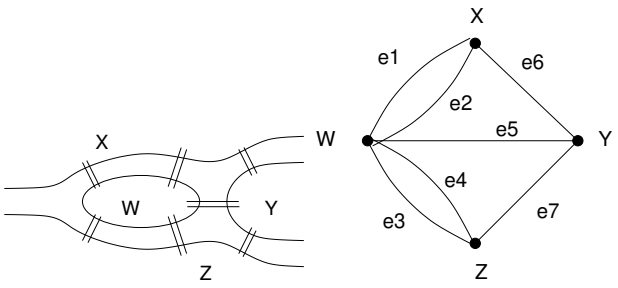
Ce chapitre est une première approche de la programmation gloutonne. Il s'agit d'une sensibilisation à ce type de programmation. Il arrive fréquemment que les algorithmes existant dans la littérature y ait recourt sans que cela soit explicitement dit. Pour pouvoir résoudre des problèmes d'optimisation dont la complexité est polynomiale, il est important de se poser les bonnes questions quant à la manière de construire cette solution. Les propriétés mise en évidence permettent de mieux connaître le problème traité d'une part et de savoir s'il est possible de recourir à la programmation dynamique ou gloutonne. La lecture des deux chapitres sur le sujet et la compréhension du devoir montrent comment il est possible de travailler face à un problème d'optimisation. Il n'est cependant pas toujours possible d'utiliser ces techniques. Le problème de coloration d'un graphe et le problème du voyageur de commerce en sont des exemples. Les solutions proposées dans ces cas utilisent des heuristiques dont les résultats sont sous optimaux, avec ou sans garanties par rapport à la valeur optimale, même si celle-ci n'est pas accessible dans un temps polynomiale. Le chapitre suivant traite du calcul des cycles eulérien et hamiltonien avec une brève introduction de la notion d'algorithme d'approximation.

Chapitre 5

Recherche de cycle dans un graphe

5.1 Introduction

Le problème de la recherche de cycle dans un graphe n'est pas neuf, il est même le point de départ de la théorie des graphes. Il a été abordé la première fois par Leonhard Euler suite à son passage dans la ville de Königsberg, en 1736. Les habitants de cette ville cherchaient à savoir s'il était possible pour eux de partir de chez eux, de traverser tous les ponts de la ville (plan 5.1a) une et une seule fois et ensuite de rentrer chez eux. Euler établit les conditions d'existence d'un tel parcours en modélisant la ville à l'aide de ce qui allait devenir un graphe (figure 5.1b). Il publia ce résultat en 1741 sans fournir de preuve. Celle-ci ne sera publiée par Hierholzer qu'en 1873. Les conditions nécessaires à son existence n'étant pas réunies pour la ville de Königsberg, la réponse au problème initial est donc négative. Ainsi les graphes dotés des propriétés énoncées par Euler sont dits graphes eulériens.



(a) Plan sommaire de la ville de Königsberg

(b) Modélisation de la ville par un graphe

FIGURE 5.1 – Problème du parcours unique de tous les ponts de la ville de Königsberg

Ce chapitre traite de deux problèmes, d'abord le problème de la recherche d'un cycle eulérien et chinois, puis le problème de la recherche d'un cycle hamiltonien, ou plutôt sa variante appelée "*problème du voyageur de commerce*", dans un graphe non orienté. Ces deux problèmes de recherche de cycle ont une formulation assez semblable, mais leur objectif et leur résolution diffèrent.

Le premier, la recherche d'un cycle eulérien, est un problème classique d'algorithmique sur les graphes pour lequel nous cherchons l'algorithme le plus performant, donc le moins coûteux. Ce n'est pas un problème optimisation puisqu'il n'y a pas de quantification du résultat trouvé. Un cycle en vaut un autre et il n'est donc pas nécessaire de comparer toutes les solutions possibles, ce qui limite l'exploration combinatoire. Nous présentons un algorithme de résolution en temps polynomial .

Le second, le problème du voyageur de commerce, est un problème d'optimisation puisque nous cherchons la tournée le cycle le plus court dans le graphe, ce qui nous conduit à comparer les cycles et augmente l'exploration combinatoire. Le problème est NP-Complet et, dans la mesure où il n'existe pas d'algorithme polynomial donnant la solution optimale, nous l'utilisons pour donner quelques éléments généraux sur les algorithmes d'approximation. Nous présentons ensuite des heuristiques pour la résolution de ce problème.

5.2 Cycles eulériens et chinois

Comme nous l'avons évoqué dans l'introduction, un cycle eulérien est un cycle qui passe une et une seule fois par chaque arête du graphe pour revenir au sommet initial. Un graphe est alors dit *eulérien* s'il possède un cycle eulérien. Dans la suite nous nous intéressons aux notions de cycles et parcours eulériens et chinois.

5.2.1 Définitions

Le théorème de Euler donne les conditions nécessaires est suffisantes pour l'existence d'un cycle eulérien dans un graphe.

Théorème 5.1 (Théorème de Euler) *Un graphe admet un cycle* eulérien *si et seulement s'il est connexe*[1] *et tous les sommets sont de degré pair.*

Un graphe qui est admet un cycle eulérien est également qualifié d'eulérien.

Par exemple, le graphe donné à la figure 5.2 est un graphe eulérien, puisque le parcours L = (8, 4, 3, 8, 1, 3, 2, 1, 5, 6, 7, 8) utilise bien toutes les arêtes avant de revenir au sommet de départ.

À noter qu'un cycle eulérien peut emprunter plusieurs fois le même sommet.

5.2.2 Recherche de cycle eulérien dans un graphe

Nous faisons bien sûr l'hypothèse que le graphe considéré est non orientée, connexe et n'a aucun sommet de degré impair, condition indispensable pour trouver un cycle eulérien.

1. Voir la définition en 1.6

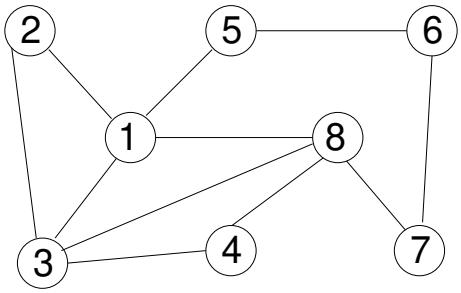


FIGURE 5.2 – Exemple de graphe eulérien

5.2.3 Algorithme de Hierholzer

La recherche d'un cycle eulérien peut se fait grâce à un algorithme par construction décrit par Hierholzer. L'idée est de marquer les arêtes traversées en explorant de proche en proche les arêtes libres, jusqu'à obtenir un cycle. Le cycle est ensuite complété, au besoin, depuis des sommets ayant des arêtes incidentes encore libres. Une constatation permet alors de bien comprendre comment fonctionne l'algorithme : puisque tous les sommets sont de degré pair alors, lorsque l'algorithme arrive à un sommet, il y a forcément une arête non marquée qui permet d'atteindre un autre sommet, sauf pour le sommet de départ dont nous avons déjà utilisé un degré pour le quitter au début. De ce fait, la recherche d'un cycle se termine forcément au sommet de départ. Le cycle trouvé n'est cependant pas forcément un cycle eulérien et il faut relancer la recherche de cycle à partir d'un des sommet ayant encore des arêtes non marquées. Pour le sous-graphe composé des arêtes non marquées et leurs sommets adjacents, la propriété eulérienne est toujours vrai sur les composantes connexes puisque en chaque sommet les arêtes ont été enlevées deux par deux.

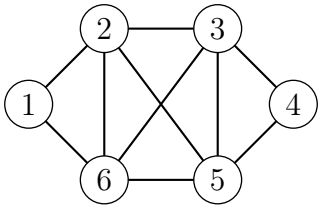


FIGURE 5.3 – Exemple de graphe pour lequel nous cherchons un parcours eulérien

Exemple Soit le graphe donné à la figure 5.3. Un parcours sur ce graphe peut donner le cycle suivant :

1 2 3 4 5 6 1

Ce parcours donne un cycle qui ne contient pas toutes les arêtes et reste bloqué au niveau du deuxième passage sur le sommet 1. Or il existe encore des arêtes non visitées, par exemple au niveau du sommet 2 : (2,6). Il est alors possible de faire un nouveau parcours, à partir de ce sommet :

2 5 3 6 2

Ce deuxième parcours peut ensuite être greffé au premier au niveau du sommet 2. On obtient alors le parcours suivant :

➽ 1 2 5 3 6 2 3 4 5 6 1

Ce parcours est eulérien puisqu'il n'y a plus d'arête non marquée.

Algorithmes L'algorithme **??** permet de trouver un cycle eulérien à partir d'un sommet x du graphe G. Il lance un parcours (en profondeur) récursivement à partir du sommet courant sur toutes les arêtes non marquées incidentes à ce sommet. Toutes les arêtes empruntées sont marquées et les retours des parcours eulériens lancés sur ces sommets voisins sont ajoutés au cycle en construction sur la pile P, les uns après les autres. Le cycle est construit au retour en mettant en liste les sommets empilés.

Algorithme 23 : Recherche d'un cycle eulérien dans un graphe par construction

Données : L : liste des sommets du cycle dans l'ordre du parcours

```
1 Fonction cycleEuler1(G, x)
2 début
3     P.push(x)
4     tant que ∃(x, y), arête non marquée faire
5         marquer((x, y))
6         cycleEuler(G, y)
7     L.add(P.pop())
```

Cet algorithme a ont une complexité linéaire en fonction du nombre d'arêtes.

5.2.4 Algorithme de Fleury

L'algorithme de Fleury propose une autre approche pour la recherche d'un cycle eulérien en choisissant de manière prioritaire une arête qui, si elle est enlevée, ne déconnecte pas le graphe restant. En effet si l'arête déconnecte le graphe, il n'est plus possible de revenir sur une partie du graphe qui n'a pas été parcourue.

Exemple Nous reprenons le graphe de la figure 5.3. Un parcours sur ce graphe peut commencer de la manière suivante :

1 2 3 4

Arrivé au sommet 4, la seule arête disponible déconnecte le sommet 4 du graphe. L'arête est tout de même utilisée car nous n'avons pas d'autre choix. Cela ne pose en fait aucun problème puisque toutes les arêtes incidentes au sommet 4 ont été enlevées. Nous continuons donc avec les sommets :

5 6

Arrivé au sommet 6, nous constatons qu'enlever l'arête (6,1) déconnecte le graphe en d'une part le sommet 1 et, d'autre part, les sommets 2, 3, 5 et 6. Il faut donc lui préférer

l'arête (6,2). Le parcours de cette arête permet alors d'ajouter le cycle :

6 2 5 3 6

Avant de finir le cycle pour obtenir :

➽ 1 2 3 4 5 6 2 5 3 6 1

Algorithme L'algorithme 24 lance un parcours récursivement à partir du sommet courant sur toutes les arêtes non marquées incidentes à ce sommet, en choisissant prioritairement les arêtes qui ne déconnectent pas le graphe.

Algorithme 24 : recherche d'un cycle dans un graphe $G(V, E)$ non orienté en marquant les arêtes

Données : L : liste des sommets du cycle dans l'ordre du parcours

```
1  Fonction cycleEuler2(G, x)
2  début
3      L.add(x)
4      tant que ∃(x, y), arête non marquée faire
5          si ∃(x, z) qui ne déconnecte pas le graphe alors
6              marquer((x, z))
7              cycleEuler2(G, z)
8          sinon
9              marquer((x, y))
10             cycleEuler2(G, y)
```

Comme l'algorithme précédent cet algorithme est en temps linéaire. À noter cependant que l'identification des arêtes qui déconnectent le graphe le rend plus coûteux lorsqu'il est implémenté.

5.2.5 Parcours eulérien d'un graphe non orienté

La propriété d'eulérien associée à un cycle peut-être étendue à la notion de parcours. Un parcours est donc *eulérien* dans graphe G s'il passe exactement une et une seule fois par chaque arête de G, sans que le sommet de départ et le sommet d'arrivée soient nécessairement les mêmes.

Le théorème suivant permet de définir les conditions nécessaires est suffisantes pour l'existence d'un parcours eulérien dans un graphe.

Théorème 5.2 (Existence d'un parcours eulérien) *Un graphe admet un parcours* eulérien *si et seulement si il est connexe et au plus deux sommets sont de degré impair.*

L'exemple classique de graphe qui admet un parcours eulérien est celui de "l'enveloppe" donné par la figure 5.4. Il est possible de dessiner sans lever le crayon de la feuille à

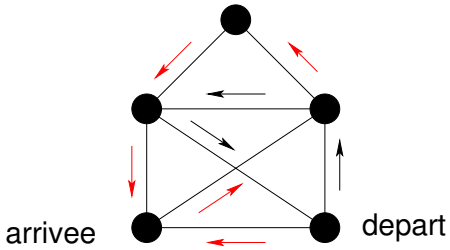


FIGURE 5.4 – Exemple de graphe admettant un parcours eulérien

condition de partir du bon endroit. Ici nous partons, par exemple, du sommet `depart` en suivant d'abord les flèches noires puis les flèches rouges pour arriver au sommet `arrivee`.

Un parcours eulérien commence et fini forcément sur un des deux sommets de degré impair puisque ce sont les seuls auxquels ils restera des arêtes après avoir été traversés autant de fois qu'il est possible. La recherche d'un parcours eulérien peut ainsi se faire de la même manière qu'un cycle, avec un algorithme de Hierholzer ou de Fleury, en partant d'un sommet de degré impair.

5.2.6 Algorithme du postier chinois

Le cadre limité de ce type de parcours restreint son champ d'application et suggère de l'élargir à des graphes généraux. Nous étudions ici la mise en place d'un cycle dans un graphe qui n'est pas eulérien, en s'autorisant à passer plusieurs fois par une arête. Ce cycle est appelé cycle chinois. Une application de ce problème peut être un graphe qui représente le plan d'une ville sur lequel nous souhaitons définir une tournée qui passe dans chacune des rues pour distribuer le courrier.

Si nous définissons un cycle chinois comme étant un cycle parcourant toutes les arêtes d'un graphe quelconque, nous avons alors les propriétés suivantes :

- un graphe non orienté admet toujours un cycle chinois s'il est connexe ;
- un graphe orienté admet un cycle chinois si et seulement si il est fortement connexe.

Le problème du postier chinois (défini par Mei-Ko Kwan en 1962) est alors la recherche d'un cycle chinois qui minimise le poids des arêtes parcourues, ou le nombre d'arêtes parcourues dans le cas où il n'y a pas de poids associé aux arêtes. Il s'agit d'un problème de minimisation ayant de nombreuses d'applications comme la distribution du courrier, la collecte des ordures ménagères, la collecte du lait, le relevé de compteurs, l'inspection de lignes à haute tension, etc. Sans autre contrainte, ces problèmes peuvent être résolus de manière optimale par des algorithmes polynomiaux. Par contre, avec des contraintes supplémentaires, ces problèmes deviennent eux aussi NP-Complets.

La différence entre un graphe qui admet un cycle eulérien et un graphe qui n'en admet pas est simplement le nombre de sommets de degré impair. Pour pouvoir effectuer un parcours qui visite toutes les arêtes au moins une fois, il faut simplement répliquer des arêtes existantes afin de se placer dans le cadre des hypothèses de la recherche d'un cycle eulérien.

Le problème du postier chinois n'est par conséquent pas la recherche du cycle propre-

ment dit, mais plutôt le problème de l'ajout des arêtes permettant l'existence d'un cycle eulérien.

Puisque la complexité associée à la recherche d'un cycle eulérien est liée à la somme des coûts des arêtes, le coût de recherche d'un cycle chinois est donc le coût de recherche d'un cycle eulérien auquel on ajoute la somme des coûts des arêtes répliquées.

L'algorithme du postier chinois se transforme donc en un problème d'optimisation qui vise à minimiser le coût des arêtes additionnelles. La résolution de ce problème peut se faire grâce à la résolution d'un problème de couplage de poids minimal pour apparier les sommets de degré impair.

5.3 Cycles hamiltoniens et voyageur de commerce

De manière symétrique à la recherche de cycle eulériens et chinois, qui s'appliquent aux arêtes traversées, la recherche de cycles hamiltoniens et le problème du voyageur de commerce s'appliquent aux sommets du graphes que l'on cherche à relier en un seul cycle.

5.3.1 Cycle hamiltonien

Un cycle qui passe une et une seule fois par chacun des sommets d'un graphe est appelé un cycle hamiltonien. Comme pour les graphes eulériens, un graphe qui admet un cycle hamiltonien est appelé graphe hamiltonien. Dans un graphe, il est également possible de rechercher des parcours hamiltoniens, qui n'ont pas forcément le même point de départ et d'arrivée.

Il existe plusieurs théorèmes prouvant que certains types de graphes sont hamiltoniens. Par exemple le théorème de Dirac dit qu'un graphe non orienté ayant au moins 3 sommets et dont le degré de chacun des sommets est supérieur à $n/2$, n étant le nombre de sommets, est hamiltonien. Il n'existe cependant pas de condition nécessaire et suffisante aussi simple et générale que le théorème d'Euler, dans le cas des graphes eulérien, pour prouver le caractère hamiltonien d'un graphe. Pour certains graphes aucun théorème ne permet de savoir s'ils sont hamiltonien ou non. Une solution pourrait alors être de faire une recherche mais le problème de la recherche d'un cycle hamiltonien dans un graphe quelconque est un problème NP-Complet.

Il existe malgré tout un algorithme de programmation dynamique qui apporte une solution en $O(n^2 2^n)$ conçu par Bellman, Held et Karp. *Cet algorithme détermine, pour chaque sous-ensemble S de sommets et chaque sommet v de S, s'il existe une chaîne couvrant exactement les sommets de S et se terminant en v. Pour chaque choix de S et de v, une chaîne existe pour (S, v) si et seulement si v a un sommet voisin w tel qu'une chaîne existe pour $(S - \{v\}, w)$, ce que l'on peut vérifier dans les informations déjà calculées par le programme*[2].

2. Repris de https://fr.wikipedia.org/wiki/Graphe_hamiltonien

5.3.2 Problème du voyageur de commerce

Dans le problème du voyageur de commerce (ou TSP, Travelling Salesman Problem), nous nous intéressons à un problème d'optimisation légèrement différent puisque le graphe est complet et les arêtes sont pondérées. Le problème du voyageur de commerce est alors le suivant : il faut trouver le cycle le plus court reliant toutes les villes sans passer deux fois par la même ville, les villes étant organisées autour d'un graphe complet. Comme la recherche d'un cycle hamiltonien, ce problème est un problème NP-Complet.

Comme dans le cas de la recherche d'une tournée de coût minimal (problème du postier chinois), il existe une simplification du problème initial dans lequel il faut passer au moins une fois dans chaque ville dans un graphe complet. On travaille ensuite grâce à un distancier, comme la matrice des plus courts chemins calculée par exemple grâce à l'algorithme de Floy-Wharshall à condition que la distance vérifie l'inégalité triangulaire.

Étant donné la difficulté de ce problème ($n!$ solutions), il n'est pas envisageable de le résoudre pour obtenir une solution exacte. Il existe en revanche des méthodes approchées dont certaines sont garanties à un facteur constant de l'optimal.

Applications

De nombreux problèmes peuvent se ramener à la recherche d'un cycle de taille minimale passant une et une seule fois par un ensemble de points :

- perçage de circuits imprimés : il s'agit de minimiser le temps de perçage des n trous de la plaque sachant que la vitesse de déplacement de la perceuse est finie. De plus, le déplacement de la perceuse n'est pas contraint, ce qui permet de considérer que les trous sont les sommets d'un graphe complet et les arêtes ont pour poids la durée de déplacement de la perceuse d'un trou à l'autre ;
- fabrication de peintures : le problème consiste en la fabrication de n lots de peinture. Le lot i a une durée f_i de fabrication et le nettoyage des instruments prend une durée $d_{i,j}$ entre le lot i et le lot j. La durée de fabrication totale dépend de l'ordre de fabrication des lots. On peut représenter les lots au sommet d'un graphe ayant comme arêtes les durées de nettoyage d'un lot à l'autre. Ce graphe est complet, les sommets doivent tous être traités une et une seule fois et la durée totale est la somme des fabrications (coût constant) auquel on ajoute le coût de tous les nettoyages. Cela revient donc à résoudre le problème du voyageur de commerce sur ce graphe.

Heuristiques gloutonnes

Plusieurs stratégies peuvent être imaginées pour parcourir tous les sommets en faisant le trajet le moins long. Lorsqu'on ajoute un sommet on prend :

- le plus proche voisin du dernier sommet ajouté. Le cycle se construit suivant une grande chaîne que l'on referme une fois le dernier sommet ajouté. Il s'agit d'un heuristique très simple mais non garantie ;
- le sommet conduisant à la plus proche insertion. Il s'agit d'ajouter le sommet dans un cycle en construction tel qu'il soit le plus proche d'un des sommets parmi tous

les sommets hors du cycle ;
— le sommet conduisant à la plus lointaine insertion. Voir l'explication précédente ;
— le sommet conduisant à la meilleure insertion, c'est-à-dire l'insertion du sommet augmentant le moins le cycle en cours de construction.

La dernière stratégie a une complexité en $O(M^3)$ et donne les meilleurs résultats en moyenne sur des graphes quelconques. Nous verrons d'autres stratégies garanties donnant encore de meilleurs résultats.

5.4 Algorithmes d'approximation

Parmi les problèmes d'optimisation, nombreux sont les problèmes NP-Complets. Le temps d'exécution d'un algorithme explorant les solutions pour déterminer la solution optimale croît alors de manière exponentielle par rapport à la taille des données du problème. De ce fait, en dehors des petites instances où il est possible de faire le calcul, il n'est pas possible de déterminer la solution optimale et trouver une solution approchée de bonne qualité est un enjeu important. Par exemple, le problème du voyageur de commerce a $n!$ solutions de parcours pour n villes. Il est donc possible de calculer la solution optimale pour une dizaine de villes, en comparant les 3628800 solutions, mais guère plus. Pour des problèmes de plus grande taille il faut utiliser des heuristiques, c'est-à-dire des méthodes de calcul donnant des solutions dont on espère qu'elles soient de bonne qualité. La question qui se pose est alors de savoir comment mesurer la qualité de ces solutions puisque notre objectif est de trouver la meilleure possible. Il nous faudrait pouvoir connaître leur distance à la solution optimale, que nous ne connaissons malheureusement pas.

5.4.1 Définition

Les algorithmes d'approximation donnent des solutions dont la distance à la solution optimale peut-être garantie par un facteur. C'est-à-dire que la plus mauvaise de leur solution pour une instance du problème donné ne sera pas pire que ce facteur, appelé facteur d'approximation.

La difficulté consiste à trouver une méthode pour laquelle ce facteur peut-être prouvé mathématiquement. Il s'agit de montrer que, quelque soit l'instance du problème, le rapport de la solution trouvée et la solution optimale peut être borné par une valeur. A noter que ce facteur peut aussi être considéré en moyenne ou aux limites.

La classe des problèmes d'optimisation pour lesquels il existe un algorithme polynomial d'approximation à un facteur borné est notée APX. Ainsi si l'algorithme d'approximation s'exécute en temps polynomial, le problème d'optimisation associé est dans la classe APX.

Dans la suite nous définissons les facteurs d'approximation des algorithmes d'approximation puis nous illustrons ce qu'est un algorithme d'approximation.

5.4.2 Facteurs d'approximation

Dans un problème d'optimisation, un facteur d'approximation, aussi appelé borne de performance, d'un algorithme d'approximation est le coefficient garanti du rapport entre la valeur d'une solution et la valeur de la solution optimale. Toutes les solutions trouvées par l'algorithme d'approximation ne sont pas plus loin de la solution optimale que le facteur d'approximation.

Nous définissons les grandeurs suivantes :

- Soit C^* le coût, ou la valeur, de la solution optimale (maximisation ou minimisation) ;
- Soit C le coût, ou la valeur, de la solution approximative ;
- Soit $\rho(n)$ le facteur d'éloignement de la solution calculée par l'algorithme d'approximation et de la solution optimale.

S'il s'agit d'un problème de maximisation, nous aurons $0 < C < C^*$ et nous nous intéressons au ratio C^*/C. A l'inverse, s'il s'agit d'un problème de minimisation, nous aurons $0 < C^* < C$ et nous nous intéressons au ratio C/C^*.

Il est alors possible d'écrire la relation suivante entre les différentes grandeurs précédemment définies :

$$1 \leq \max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) < \rho(n)$$

Nous pouvons également définir une erreur relative $\varepsilon(n)$, ou borne inférieure relative, définie par :

$$\frac{|C - C^*|}{C^*} < \varepsilon(n)$$

Nous illustrons la notion d'algorithme d'approximation dans la suite avec l'exemple du voyageur de commerce.

5.4.3 Application au problème du voyageur de commerce

Le problème du voyageur de commerce peut être modélisé par un graphe $G(V, E)$ non orienté avec une fonction de coût $c(i, j)$ associée à toutes les arêtes $(i, j) \in E$ telle que $c(i, j) \geq 0$. Dans ce cas une tournée est un sous-ensemble $A \subset E$ des arêtes.

➽ Nous cherchons une tournée dans G de coût minimal $c(A)$, définie par :

$$c(A) = \sum_{(u,v)\in E} c(u, v)$$

La fonction de coût respecte l'inégalité triangulaire, c'est à dire que passer par un point intermédiaire ne diminue pas le coût, ou bien la suppression d'une étape n'augmente jamais le coût :

$$c(u, w) \leq c(u, v) + c(v, w)$$

En exploitant cette propriété il est possible de trouver un algorithme d'approximation avec un facteur 2.

Nous procédons de la manière suivante, l'exemple donné à la figure 5.5 illustre cette stratégie :

— recherche d'un arbre de recouvrement de poids minimal avec l'algorithme de Prim (figure 5.5.a),
— parcours en profondeur d'abord de l'arbre. Nous faisons un cycle en parcourant deux fois (aller puis retour) chaque arc (figure 5.5.b),

Nous rappelons qu'un arbre couvrant de poids minimal utilise toujours les arêtes de poids minimal pour relier deux sommets. Ainsi, du fait de l'inégalité triangulaire, le poids de l'arbre recouvrant minimal $ACPM(G)$ est inférieur au coût du cycle optimal C^*. Or notre stratégie ne fait que doubler la valeur de $ACPM(G)$ en parcourant deux fois chaque arête et nous avons $ACPM(G) \leq C^* \leq C \leq 2 \times C^*$. Il s'agit bien d'un algorithme d'approximation avec un facteur 2. La solution optimale peut être trouvée, par exemple dans le cas d'un graphe composé de 2 sommets.

A noter que, en supprimant les sommets intermédiaires déjà visités par le parcours en profondeur, nous n'allongeons pas le cycle mais nous réduisons le parcours, toujours en nous reposant sur l'inégalité triangulaire.

Dans l'exemple de la figure 5.5, la solution approchée est calculée de la manière suivante :

1. construction de l'arbre de recouvrement de poids minimal de racine ***a*** ;
2. parcours préfixé complet de l'arbre :

   ➽ ***a b c b h b a d e f e g e d a***

3. élimination des doublons en prenant le prochain sommet dans la liste précédente si celui-ci n'a pas été encore parcouru, etc :

   ➽ ***a b c h d e f g***

La complexité de l'algorithme permettant de calcul d'une valeur approchée de la tournée optimale d'un voyageur de commerce est $\Theta(|E|) = \Theta(|S|^2)$.

Heuristique de Christofides Il existe encore une autre heuristique qui améliore la borne sur la garantie de la solution approchée calculée (3/2). Le principe de cet autre heuristique est de trouver un couplage minimal entre les sommets de degré impair dans l'arbre recouvrant calculé à la première étape. La recherche d'un parcours eulérien est alors possible et améliore ce que l'inégalité triangulaire avait permis de réduire par rapport au passage deux fois par chaque arêtes.

Définir un algorithme d'approximation pour un problème consiste donc à trouver une méthode de calcul puis à prouver que les solutions trouvées sont à un facteur de l'optimal. A noter que cela ne présage pas de la qualité de l'algorithme, le facteur d'approximation pouvant être important. Dans certains cas il peut être nécessaire d'avoir recours à un algorithme d'approximation si nous avons besoin d'une solution garantie

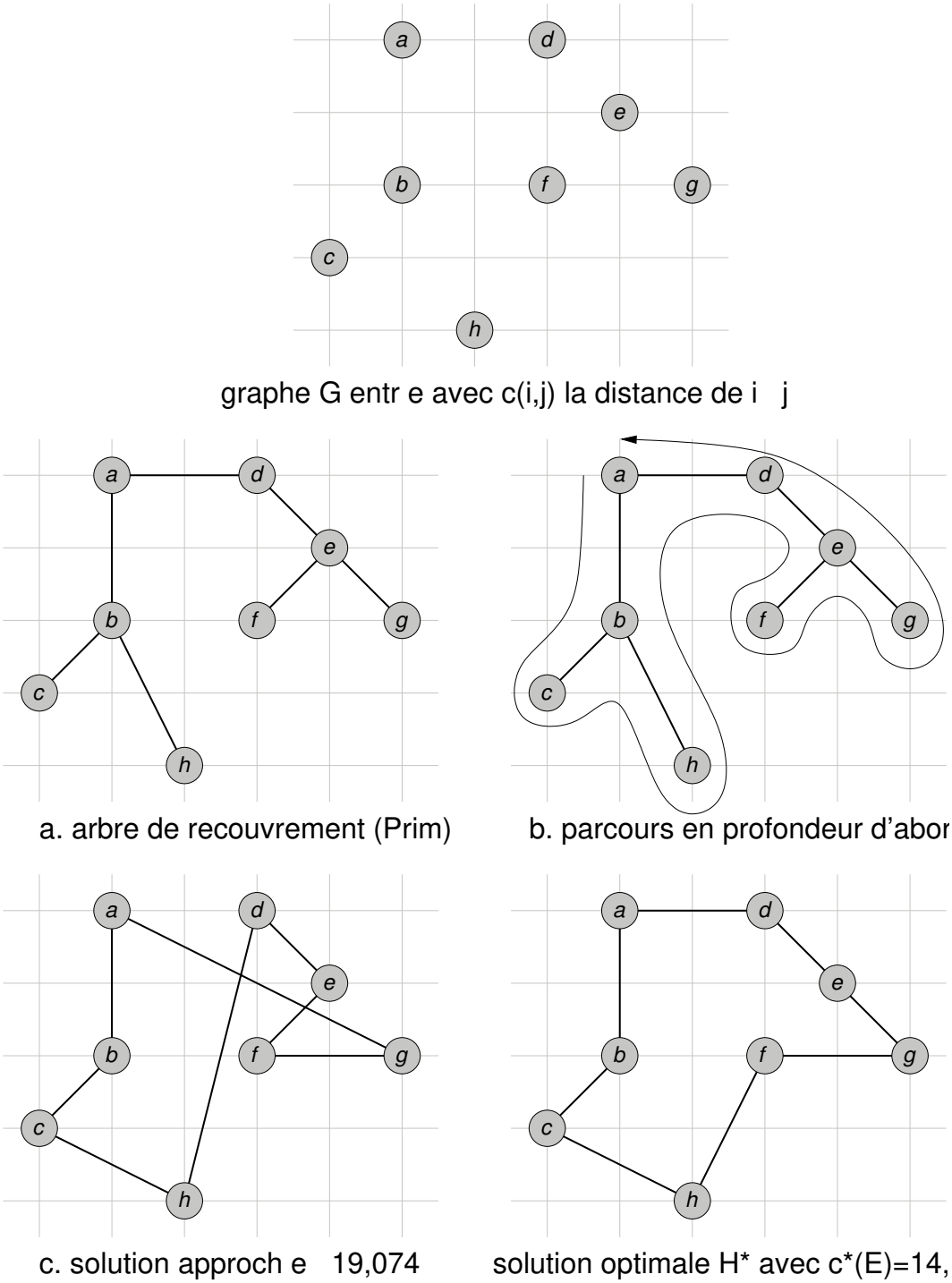


FIGURE 5.5 – Construction d'une solution approchée du problème du voyageur de commerce en utilisant l'inégalité triangulaire

5.5 Conclusion

À l'issue de ce chapitre, et après avoir traité les exercices qui vont avec, vous devez être capables de :

- Comprendre des algorithmes d'optimisation sur les graphes ;
- Comprendre des algorithmes complexes sur les graphes ;
- Concevoir des algorithmes d'optimisation sur les graphes ;

Ce chapitre a été l'occasion de montrer deux problèmes aux énoncés peu éloignés mais avec des performances de résolution très différentes. La recherche d'un cycle eulérien dans un graphe eulérien est polynomial alors que la recherche d'un cycle hamiltonien de longueur minimale dans un graphe complet est NP-Complet. On comprend ainsi l'importance de disposer d'algorithmes performants pour la résolution de problèmes d'optimisation en général et de calcul sur les graphes en particulier. Ces deux domaines se rejoignent souvent, car de nombreux problèmes d'optimisation se modélisent à l'aide d'un graphe. Dans le cas où une approche naïve conduit à une complexité exponentielle, nous avons évoqué différentes pistes pour travailler à son amélioration avec le recours à la programmation linéaire, dynamique et gloutonne. Cependant, comme nous le voyons dans ce dernier cas traité, lorsqu'aucune de ces techniques ne donne de résultats, un algorithme approché doit être imaginé avec, si possible, la définition d'une garantie de cette solution.

De nombreuses techniques existent pour rechercher une solution approchée d'un problème difficile. Ces techniques travaillent souvent à partir d'une première solution à améliorer. Ainsi, l'algorithmique génétique, les méthodes tabou et de recuit simulé donnent de très bons résultats pour la résolution de problèmes d'optimisation. L'étude de ces techniques dépasse le cadre de ce cours, mais peut faire l'objet d'une recherche personnelle dans les prochains travaux que vous pourrez avoir à traiter.

Annexes

Exemples de graphes remarquables

Nous présentons dans cette partie quelques structures de graphes ayant une géométrie ou des propriétés remarquables.

Graphe complet K_n

La définition d'un graphe complet a déjà été donnée page 50. Soit K_n le graphe complet à n sommets :

- degré : $\delta = \Delta = n - 1$
- nombre de sommets : n
- nombre d'arêtes : $\frac{n(n-1)}{2}$
- diamètre : 1

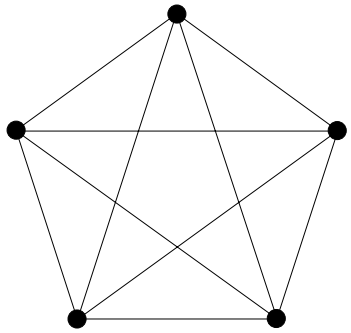


FIGURE 6 – Exemple de graphe complet K_5

La figure 6 donne un exemple de graphe complet.

Anneau R_n

- degré : $\delta = \Delta = 2$
- nombre de sommets : n
- nombre d'arêtes : n
- diamètre : $\lfloor \frac{n}{2} \rfloor$

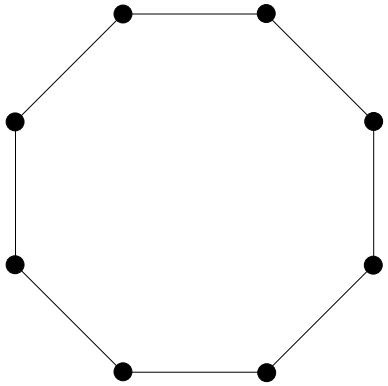


FIGURE 7 – Exemple d'anneau R_8

Comme sont nom l'indique, dans un graphe en anneau, les sommets sont reliés entre eux pour former un anneau. Plus formellement, un anneau R_n de dimension n est un graphe tel que le sommet i est connecté aux sommets $(i-1)$ mod n et $(i+1)$ mod n pour $i = 0, \ldots, n-1$.

La figure 7 donne un exemple de graphe en anneau.

Hypercube H_n

Un hypercube est un graphe qui étend la notion de cube à la dimension n. Un cube possède trois dimensions, 8 sommets et 12 arêtes qui permettent de relier chaque sommet à 3 autres. Un hypercube possède 2^n sommets qui sont reliés à n autres sommets de manière régulière. Ainsi, un hypercube H_n, de dimension n, est un graphe (non orienté) dont les sommets sont tous les mots de longueur n sur un alphabet binaire. Le sommet $x_1x_2 \ldots x_i \ldots x_n$ est relié avec le sommet $x_1x_2 \ldots \bar{x}_i \ldots x_n$.

- degré : $\delta = \Delta = n$
- nombre de sommets : $N = 2^n$
- nombre d'arêtes : $n2^{n-1}$
- diamètre : $n = \log_2(N)$

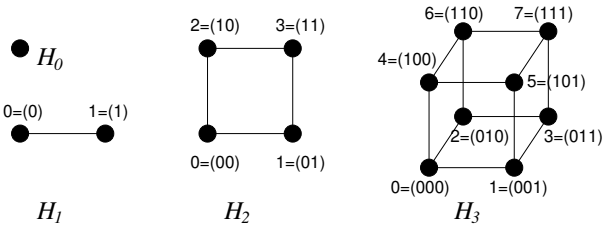


FIGURE 8 – Exemple d'hypercubes H_0, H_1, H_2, H_3

La figure 8 donne un exemple de graphe en hypercube. Le calcul du nombre d'arêtes se prouve par récurrence : remarquez que pour un nombre de sommets de 2^n, il faut considérer les arcs de 2^{n-1} sommets, car chaque sommet a n arcs. Il est possible d'extraire des sous-graphes remarquables dans l'hypercube comme les anneaux, les grilles ou les arbres.

Cube connected cycle CCC_n

- degré : $\delta = \Delta = 3$
- nombre de sommets : $N = n.2^n$
- nombre d'arêtes : $3n.2^{n-1}$
- diamètre : $2n - 1 + \max(1, \lfloor \frac{n-2}{2} \rfloor)$

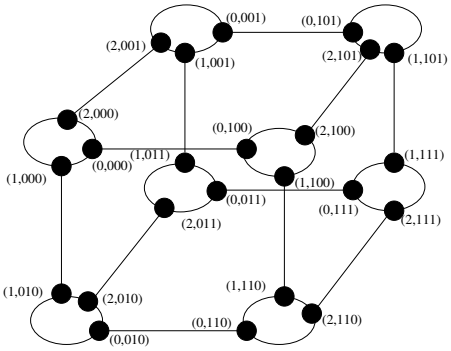


FIGURE 9 – Cube connected cycle CCC_3

Dans un cube connected cycle, les mots de longueur n sont préfixés par un nombre variant de 0 à $n-1$.

La figure 9 donne un exemple de graphe en cube connected cycle. Le diamètre est démontrable facilement avec une borne de $2n$. Pour une sensibilisation, il faut le vérifier sur une dimension plus élevée, $n = 8$ par exemple.

Grille $M(n_1, n_2, \ldots, n_n)$

La grille est un graphe où les sommets sont répartis à intervalle régulier sur des axes verticaux et horizontaux et forment un rectangle. Les arêtes relient les sommets à leur voisins sur les axes. La grille de dimension k est notée $M(n_1, n_2, \ldots, n_k)$, c'est le produit cartésien de k chaînes de n_i sommets avec $i = 1, \ldots, k$.

- degré : $\delta = k$, $\Delta = 2k$
- nombre de sommets : $N = \prod_{i=1}^{k} n_i$
- nombre d'arêtes : $\sum_{i=1}^{k} \frac{(n_i-1)N}{n_i} = \sum_{i=1}^{k}(n_i - 1).\frac{N}{n_i}$
- diamètre : $\sum_{i=1}^{k}(n_i - 1)$

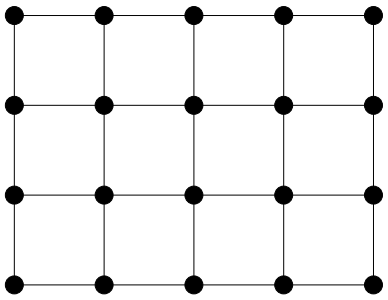


FIGURE 10 – Exemple de grille $M(4, 5)$

La figure 10 donne un exemple de graphe en grille. Sur cette figure le nombre d'arêtes de la grille est $M(4, 5) = (n_1 - 1).\frac{N}{n_1} + (n_2 - 1).\frac{N}{n_2} = (4 - 1).\frac{20}{4} + (5 - 1).\frac{20}{5} = 15 + 16 = 31$

Grille torique $TM(n_1, n_2, \ldots, n_k)$

Une grille torique est une grille à laquelle sont ajoutés des arêtes entre les sommets du bord. Il s'agit du produit cartésien de k cycles Cp_i avec $i = 1, \ldots, k$.

- degré : $\delta = \Delta = 2k$
- nombre de sommets : $N = \prod_{i=1}^{k} n_i$
- nombre d'arêtes : $\sum_{i=1}^{k}(\prod_{i=1}^{k} n_i) = \sum_{i=1}^{k} N = nN$
- diamètre : $\sum_{i=1}^{k} \lfloor \frac{n_i}{2} \rfloor$

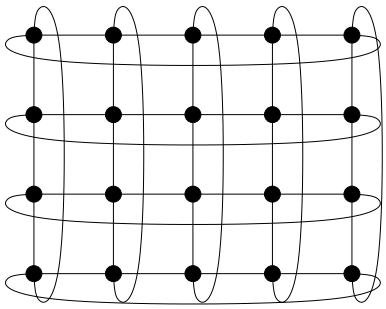


FIGURE 11 – Exemple de grille $TM(4, 5)$

La figure 11 donne un exemple de graphe en grille torique obtenue à partir de la grille de l'exemple précédent.

Shuffle Exchange SE_n

Le SE_n de dimension n a pour sommets les mots binaires de dimension n. Le sommet $x_1x_2 \ldots x_n$ est relié avec les sommets $x_1x_2 \ldots \bar{x}_n$ (arête échange), $x_2x_3 \ldots x_nx_1$ et

$x_n x_1 x_2 \dots x_{n-1}$ (arêtes décalage).

- degré : $\delta = 1$, $\Delta = 3$
- nombre de sommets : $N = 2^n$
- nombre d'arêtes :
  $3 \times 2^{n-1} - 3 + (n \bmod 2)$
- diamètre : $2n - 1$

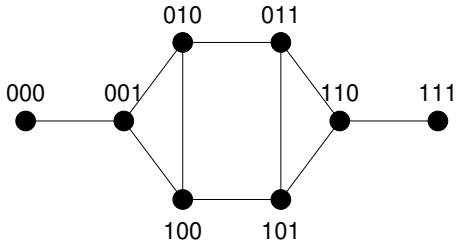


FIGURE 12 – Exemple de Suffle Exchange SE_3

Butterfly BF_n

Le BF_n de dimension n est le graphe dont les sommets sont en bijection avec les couples (l, x) où l est le niveau tel que $0 \leq l < n$ et x un mot binaire de longueur n qui correspond au numéro de rangée.

Le sommet $(l, x_0 x_1 \dots x_{n-1})$ est connecté avec les sommets $((l+1) \bmod n, x_0 x_1 \dots x_{n-1})$ et $((l+1) \bmod n, x_0 x_1 \dots \bar{x}_l \dots x_{n-1})$.

- degré : $\delta = \Delta = 4$
- nombre de sommets : $N = n2^n$
- nombre d'arêtes : $n2^{n+1}$
- diamètre : $n + \lfloor \frac{n}{2} \rfloor$

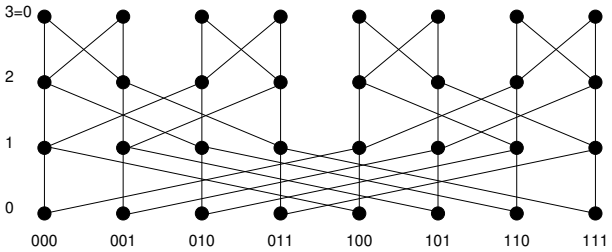


FIGURE 13 – Exemple de Butterfly BF_3

Star Graph SG_n

Le SG_n est un Star Graph de dimension n si ses sommets sont des permutations d'un ensemble à n éléments ($n \geq 2$). Le sommet $x_1 x_2 \dots x_n$ est connecté avec les sommets $x_i x_2 \dots x_{i-1} x_1 x_{i+1} \dots x_n$ avec $i = 2 \dots n$.

De Bruijn $B_{d,n}$

Le $B_{d,n}$ est un graphe de De Bruijn, orienté, tel que les sommets sont dans $\{0, \dots, d-1\}^n$. Le sommet $(x_1 \dots x_n)$, nombre en base d, est connecté avec le sommet $(y_1 \dots y_n)$ si et seulement si $x_2 \dots x_n = y_1 \dots y_{n-1}$.

— degré : $\delta = \Delta = n - 1$
— nombre de sommets : $N = n!$
— nombre d'arêtes : $\frac{(n-1)n!}{2}$
— diamètre : $\lfloor \frac{3(n-1)}{2} \rfloor$

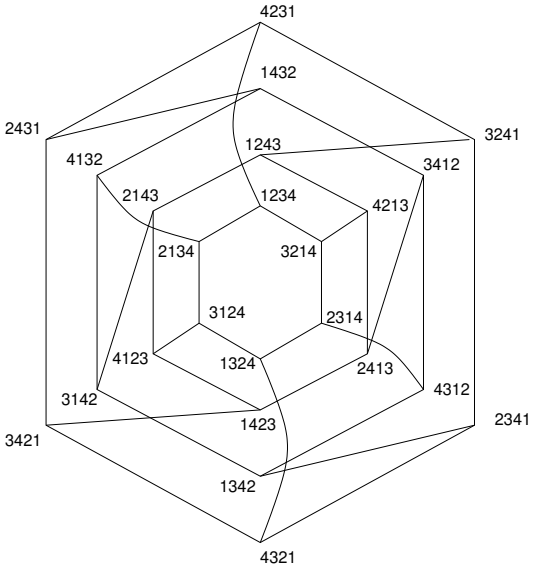


FIGURE 14 – Star Graph SG_4

— degré : $\delta = \Delta = 2d$
— nombre de sommets : $N = d^n$
— nombre d'arêtes : d^{n+1}
— diamètre : $2n - 1$

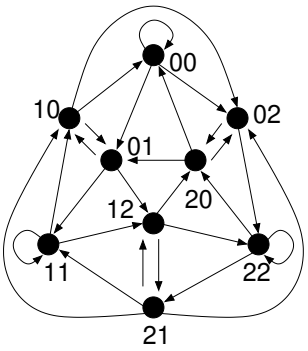


FIGURE 15 – Graphe de De Bruijn $B_{3,2}$