



MASTER 2 INFORMATIQUE I2A

Master DVL Master ITVL

FINANCE

HISTOIRE

GÉOGRAPHIE

INFORMATIQUE

MATHÉMATIQUES

SCIENCES POUR L'INGÉNIEUR

FRANÇAIS LANGUE ÉTRANGÈRE

ADMINISTRATION ÉCONOMIQUE ET SOCIALE

DIPLÔME D'ACCÈS AUX ÉTUDES UNIVERSITAIRES



Centre de Télé-enseignement
Universitaire

<http://ctu.univ-fcomte.fr>

FILIÈRE INFORMATIQUE

● VVI9MTGC

Théorie des graphes et combinatoire

Mr PHILIPPE - LAURENT
laurent.philippe@univ-fcomte.fr



UNIVERSITÉ DE
FRANCHE-COMTÉ



UNIVERSITÉ
BOURGOGNE FRANCHE-COMTÉ

Théorie des Graphes et Combinatoire
MASTER INFORMATIQUE AVANCÉE ET APPLICATIONS

Cours



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

Table des matières

1	Graphes et algorithmes	11
1.1	Introduction aux graphes	11
1.1.1	Graphe	12
1.1.2	Diagramme sagittal d'un graphe	12
1.1.3	Boucles et P-Graphes	13
1.1.4	Orientation d'un graphe	14
1.1.5	Sous-graphe et graphe partiel	16
1.1.6	Vocabulaire	16
1.2	Représentation informatique	18
1.2.1	Les matrices d'adjacence	18
1.2.2	Ensembles d'adjacence	19
1.3	Chemins et parcours	22
1.3.1	Chemins et chaînes	22
1.3.2	Mesures dans un graphe	24
1.3.3	Recherche de chemin ou de cycle	25
1.3.4	Parcours	25
1.3.5	Algorithme de parcours en largeur	27
1.3.6	Algorithme de parcours en profondeur	31
1.3.7	Calcul d'un chemin correspondant à un parcours	32
1.4	Recherche des chemins d'un graphe	32
1.5	Plus courts et plus longs chemins	34
1.5.1	Pondérations	34
1.5.2	Algorithme générique matriciel de Floyd-Warshall	35
1.5.3	Algorithme à fixation d'étiquettes de Dijkstra	36
1.6	Connexité et forte connexité d'un graphe	38
1.6.1	Algorithmes de calcul de composantes connexes	40
1.6.2	Algorithme de calcul de composantes fortement connexes	41

1.6.3	Algorithme de Tarjan	42
1.7	Arbres	45
1.8	Recherche d'un arbre couvrant de poids minimal	47
1.9	Caractérisation des graphes	47
1.10	Synthèse	50
2	Programmation Linéaire	51
2.1	Exemple de problème linéaire	51
2.2	Problème général	53
2.2.1	Définition	53
2.2.2	Principe de résolution	53
2.2.3	Applications	55
2.2.4	Algorithme de résolution	55
2.3	Forme canonique et forme standard	55
2.3.1	Forme canonique	55
2.3.2	Conversion d'un programme linéaire sous forme canonique	55
2.3.3	Conversion d'un programme linéaire sous forme standard	56
2.4	Algorithme du simplexe	58
2.4.1	Exemple d'exécution de l'algorithme du simplexe	58
2.4.2	Calcul du pivot	62
2.4.3	Calcul du simplexe	63
2.5	Synthèse	64
3	Programmation dynamique	65
3.1	Introduction	66
3.2	Mémoïsation	66
3.3	Illustration de la programmation dynamique	68
3.3.1	Structure de la solution optimale (1)	69
3.3.2	Solution récursive (2)	69
3.3.3	Solution en programmation dynamique (3)	70
3.3.4	Algorithme de Bellmann - Ford	71
3.4	Bilan	74
3.5	Synthèse	74
4	Programmation gloutonne	77
4.1	Introduction	77

4.2	Recherche d'un arbre couvrant de poids minimal	79
4.3	Problème du choix d'activités	81
4.3.1	Présentation du problème	81
4.3.2	Exemple	82
4.3.3	Solution avec la programmation dynamique	82
4.3.4	Conversion d'une solution du type programmation dynamique en une solution gloutonne	84
4.4	Éléments de stratégie gloutonne	87
4.4.1	Étapes de conception pour les activités	87
4.4.2	Propriétés du choix glouton	88
4.4.3	Sous structure optimale	88
4.4.4	Programmation gloutonne et programmation dynamique	89
4.5	Synthèse	89
5	Recherche de cycle dans un graphe	91
5.1	Introduction	91
5.2	Cycles eulériens et chinois	92
5.2.1	Définitions	92
5.2.2	Parcours eulérien d'un graphe non orienté	94
5.2.3	Algorithme du postier chinois	95
5.3	Cycles hamiltoniens et voyageur de commerce	96
5.3.1	Cycle hamiltonien	96
5.3.2	Problème du voyageur de commerce	97
5.4	Algorithmes d'approximation	98
5.4.1	Définition	98
5.4.2	Facteurs d'approximation	99
5.4.3	Application au problème du voyageur de commerce	99
5.5	Conclusion	102
	Exemples de graphes remarquables	105

Introduction

Présentation du cours

Le cours de Théorie des Graphes et Combinatoire 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 combinatoire¹.

A tous les niveaux de l'informatique, il est nécessaire de modéliser les données et le problème auxquels nous nous intéressons pour le 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 points, la création d'emplois du temps, l'ordonnancement ou l'affectation de tâches, etc. La théorie des graphes est relativement nouvelle en comparaison à d'autres domaines des mathématiques. C'est l'avènement des premiers ordinateurs qui a permis le développement de ce domaine, à la frontière entre les mathématiques et l'informatique, qui reste un domaine de recherche très actif. Les domaines dans lesquels sont utilisés les graphes sont nombreux car il s'agit d'une structure de données ou une structure abstraite générale. 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

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'algorithme 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 solutions naïves sont soit rarement efficaces, soit donnent des résultats non optimaux. Il est alors important

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

de connaître les conditions de résolution de ces problèmes, leur complexité, et les techniques dont nous pouvons disposer 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 (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.

La théorie des 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

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 des graphes quelconques. Il est alors facile d’exécuté l’algorithme sur 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 fil de discussion.

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 : quel que 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 $x = N$

$\{ \dots \}$: 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, inclu dans E ou égal

\cup : 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 dans le cours et qui doit être utilisée dans les rendus.

Pile<type> :

— *push*(type) : empiler

— *pop*() : dépiler

— *peek*() : lire premier élément sans le dépiler

— *boolean empty*() : pile vide

File<type> :

— *add*(type) : ajouter un élément à la fin de la file

— *poll*() : lire et supprimer le premier élément

— *peek*() : lire le premier élément de la file sans le supprimer

— *boolean empty()* : file vide

Liste<type> :

— *add(type)* : ajouter un élément à la fin de la liste

— *type get(int)* : lire, sans supprimer l'élément, l'index donné en paramètre

— *int size()* : taille de la liste

— *set(index, type)* : changer l'élément dont l'index est donné en paramètre

— *type remove(index)* : supprime l'index donné en paramètre

— *boolean empty()* : file vide

— *boolean contains(type)* : l'élément est dans la liste

Pour les listes les indexes commencent en 0.

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 $\{\{Laura, Paul\}, \{Thomas, Paul\}, \{Thomas, Gautier\}, \{Paul, Naddir\}, \dots\}$.

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

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.

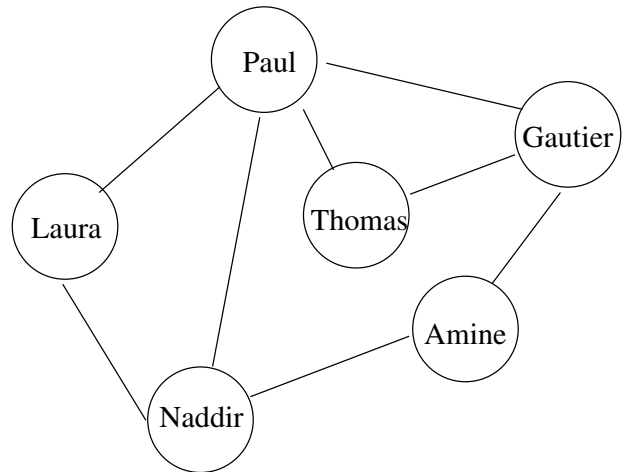
La représentation donnée sur la figure 1.1b est appelée diagramme sagittal. C'est une représentation d'un graphe où les sommets sont des points ou des cercles et les arêtes des liens. C'est la représentation classique d'un graphe sous une forme visuelle.

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 Paul sont en contact

(a) Représentation textuelle des contacts



(b) Représentation graphique des contacts

FIGURE 1.1 – Représentations d’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.

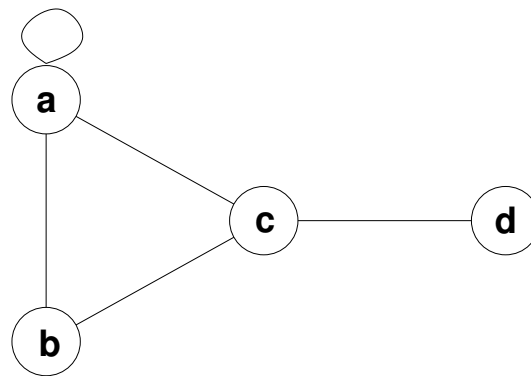


FIGURE 1.2 – Exemple de graphe avec une boucle

Deux sommets d’un graphe peuvent être reliés par plus d’une arête s’il y a plus d’une relation qui les lie, 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.

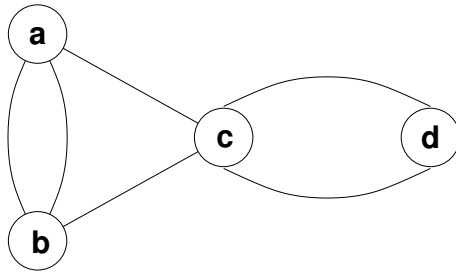


FIGURE 1.3 – Exemple de 2-graphe

Précisions :

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

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

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

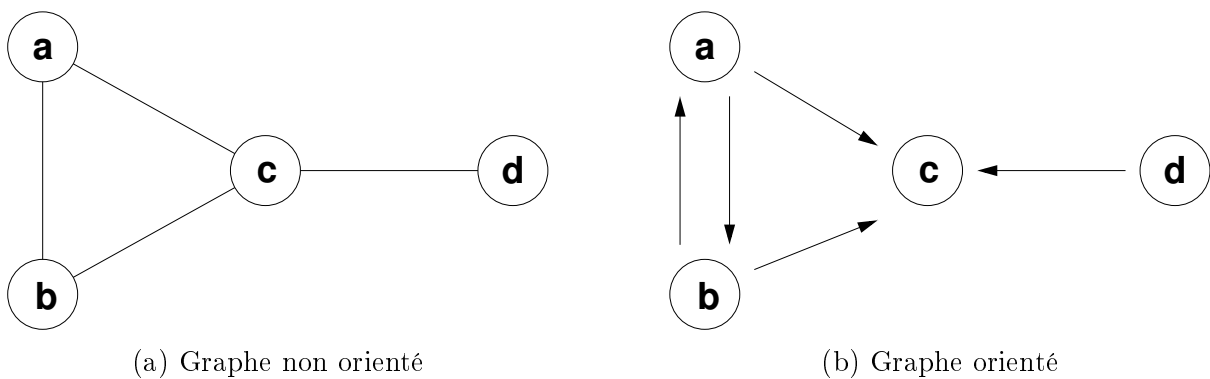


FIGURE 1.4 – Exemples de graphes

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) est différent de l'arc (b, a) .

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

Définition 1.1 : Graphe symétrique

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

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.

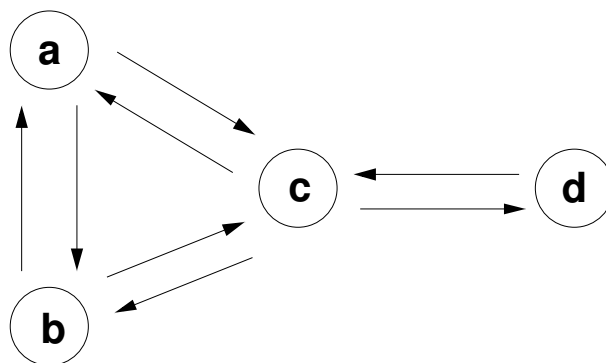


FIGURE 1.5 – Exemple de graphe symétrique

Définition 1.2 : Graphe antisymétrique

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

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

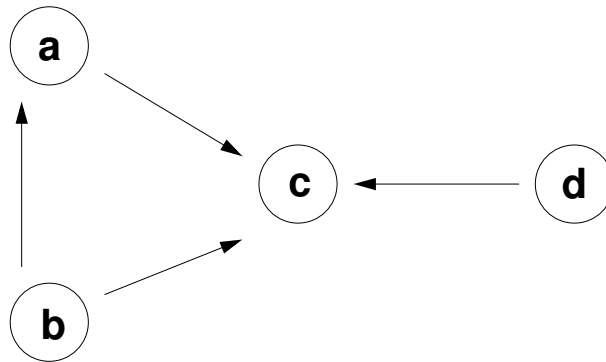


FIGURE 1.6 – Exemple de graphe antisymétrique

La figure 1.6 donne un exemple de graphe antisymétrique puisque aucun arc n'est doublé. La notion de symétrie/antisymétrie ne s'applique pas 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 générique². 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 arc 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 graphe sagittal. Ces deux représentations conviennent bien à l'interprétation humaine mais 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 : listes ou objets.

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 : $M[i][j] = 1$. Pour les graphes non orientés, une arête $\{i, j\}$ est représentée par deux valeurs à 1 dans la matrice d'adjacence : $M[i][j] = 1$ et $M[j][i] = 1$ alors que si le graphe est orienté

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

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

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'adjacence M_{no} suivante. Le graphe exemple donné à la figure 1.4b peut être représenté par la matrice d'adjacence M_o .

$$M_{no} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad M_o = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

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 M_{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 :

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

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.

Cette représentation 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 $M[x][y]$.

Par contre, 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 car elle consomme inutilement une place mémoire importante puisque la matrice M contient dans ce cas majoritairement des zéros. Par exemple, la matrice du graphe donné à la figure 1.4b a un tiers des ses valeurs à 0. Il existe alors des représentations alternatives pour éviter de stocker des informations inutiles.

1.2.2 Ensembles d'adjacence

Une autre manière de représenter les graphes est de ne traduire que les arêtes existantes dans le graphe. De ce cette manière la consommation mémoire de la mémorisation du graphe est limitée au nécessaire et ne dépend que du nombre de relations. Cela peut se faire sous la forme d'un ensemble d'adjacence pour chaque sommet. Pour l'exemple de la figure 1.4b, il suffit de traduire le fait qu'il existe des arêtes depuis le sommet a (1) vers les sommets b (2) et c (3), des arêtes depuis le sommet b (2) vers les sommets a (1) et c (3), etc.

A noter que la notion d'adjacence (ou de voisinage), telle que nous l'avons définie précédemment ou telle qu'elle est définie par Wikipedia, considère qu'un sommet prédécesseur ou successeur d'un autre en est adjacent. Dans le cas de la mémorisation d'un graphe orienté sous la forme d'ensemble d'adjacence, il est en général pertinent de ne pas stocker les prédécesseurs d'un sommet, par exemple si l'algorithme se contente de suivre les liens dans le sens de leur orientation, puisque l'ensemble définit ainsi implicitement l'orientation des relations. Si l'algorithme nécessite d'avoir accès aux prédécesseurs d'un sommet, la structure d'ensemble d'adjacence est généralement composée de deux ensembles par sommet, un pour les prédécesseurs et un pour les successeurs.

Les ensembles d'adjacence, ou de voisins, peuvent être implémentés de manière diffé-

rentes suivant l'utilisation qui en est faite. Les deux principales représentations sont les tableaux, tableau des successeurs et adresse des premiers successeurs, ou des listes chaînées.

Successeurs/Premier Successeur

Une solution 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 1 à n et nous enregistrons dans le tableau des successeurs (ou file des successeurs *FS*) successivement le numéros 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 l'entier 0 mais on peut aussi utiliser -1 si les sommets sont numérotés de 0 à $n - 1$.

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* ou adresse du premier successeur, dans lequel la i ème 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 plus indispensable. Il est généralement 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 utilisé la valeur contenue dans le tableau des premiers successeurs.

En associant les valeurs 1, 2, 3, 4 successivement aux sommets a, b, c, d , l'exemple du graphe 1.4a conduit aux tableaux FS et APS suivants :

FS :

2	3	0	1	3	0	1	2	4	0	3	0
---	---	---	---	---	---	---	---	---	---	---	---

APS :

0	3	6	10
---	---	---	----

Ce type de représentation permet une manipulation simple et une faible occupation mémoire mais peut manquer de dynamisme si le graphe doit changer. En effet, l'insertion d'éléments est coûteuse dans un tableau. Une autre forme de représentation adaptée, comme ici aux graphes peu dense, doit alors être envisagée. On peut par exemple utiliser des listes chaînées plus dynamiques.

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.

Listes chaînées

Le choix fait classiquement pour ce type de représentation est le suivant. Le plus souvent, ce sont les arêtes qui sont amenées à évoluer, plus que les sommets. Dans ce cas, la structure envisagée se décompose en un tableau de sommets, chaque case correspondant à un sommet pointe sur sa liste des sommets voisins. Cette liste est une liste chaînée.

Il existe une grande souplesse d'implantation pour ce type de représentation, tant au niveau de la structure des informations liées au sommet lui-même, aux voisins (donc à l'arête liant le sommet à son voisin), qu'au niveau de la gestion de cette liste chaînée (liste simplement chaînée, doublement chaînée, doublement chaînée circulaire, doublement chaînée avec sentinelle, etc).

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.

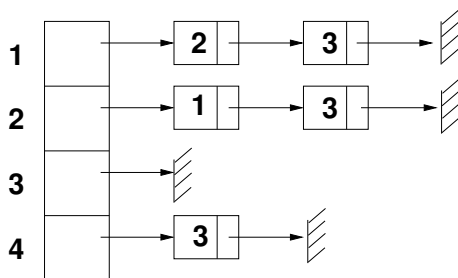


FIGURE 1.7 – Tableau des listes des voisins (ou des suivants) du graphe de la figure 1.4b

Objets

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 JGraphT³ 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 Guava⁴ ou python-graph⁵.

Remarque : *la conception des algorithmes sur les graphes dépend fortement de la structure de données utilisée pour mémoriser le graphe. Comme nous l'avons déjà évoqué certaines structures de données facilitent d'accès aux informations nécessaires pour atteindre notre objectif, d'autres seront plus adaptées à un autre objectif. 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 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.

3. <https://jgrapht.org/>

4. <https://github.com/google/guava/>

5. <https://github.com/pmatiello/python-grap>

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 orientés⁶.

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 une 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 : Une chemin de longueur p est une suite finie (x_0, x_1, \dots, 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.

Formellement : Une chaîne de longueur p est donc une suite finie (x_0, x_1, \dots, 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 $\{x_{i+1}, x_i\} \in E$.

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/Chaîne_\(théorie_des_graphes\)](http://fr.wikipedia.org/wiki/Chaîne_(théorie_des_graphes)))

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 $c = (x_0, x_1, \dots, x_p)$ et $c' = (x_0, x'_1, \dots, x_p)$ 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 orienté⁷) est un chemin (resp. chaîne) fermé de longueur non nulle.

Formellement : Un chemin (resp. chaîne) de longueur $p, p \neq 0$, $c = (x_0, x_1, \dots, x_p)$ 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_\(théorie_des_graphes\)](http://fr.wikipedia.org/wiki/Cycle_(théorie_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 $d_G(c) = 3$.

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 :

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

La démonstration de cette proposition est donnée en exercice, dans la feuille d'exercices associées à ce chapitre. Vous pouvez valider cette propriété expérimentalement sur les graphes donnés dans le cours. Plus particulièrement la somme des degrés du graphe de la figure 1.4a est de 8, ce qui est effectivement pair.

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 chemin 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.

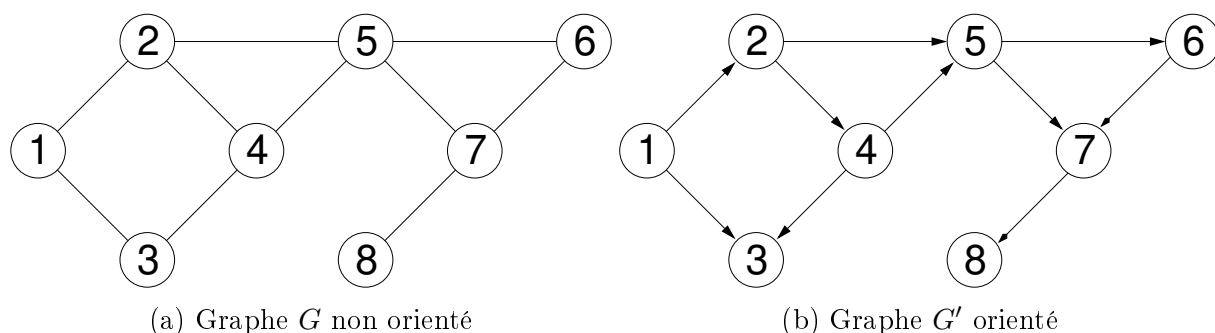


FIGURE 1.8 – Graphes non orienté et orienté

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 $L = (x_1, \dots, x_n)$ 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 j ème élément de cette liste, alors $\forall j \in \{2, \dots, n\}$, $B(L[1, j-1], G) \neq \emptyset \Rightarrow x_j \in B(L[1, j-1], G)$

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). À l'inverse, un sommet fermé est un sommet dont tous les voisins ont déjà été parcourus.

Formellement : Un sommet x d'un parcours $L[1, i] = (x_1, \dots, x_i)$ 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 $L = (x_1, \dots, x_n)$ est dit en largeur si $\forall j \in \{2, \dots, n\}$ 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 $L = (x_1, \dots, x_n)$ est dit en parcours en profondeur si $\forall j \in \{2, \dots, n\}$ 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 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.

Algorithme générique : l'algorithme 1 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 1 : parcours en largeur d'un graphe $G(V, E)$ à partir du sommet s

Données : f : file des sommets en cours **initialisé à** file vide

$couleur[u]$: $u \in \{BLANC, GRIS\}$, **initialisé à** *BLANC*, $\forall u \in V$

1 **Fonction** *parcoursLargeur*(G, s) :

2 **début**

3 $f.add(s)$

4 **tant que** $\neg f.empty()$ **faire**

5 $u \leftarrow f.poll()$

6 **pour chaque** *sommet* v *voisin de* u **faire**

7 **si** $couleur[v] = BLANC$ **alors**

8 $couleur[v] \leftarrow GRIS$

9 $f.add(v)$

10 $couleur[u] \leftarrow NOIR$ // optionel

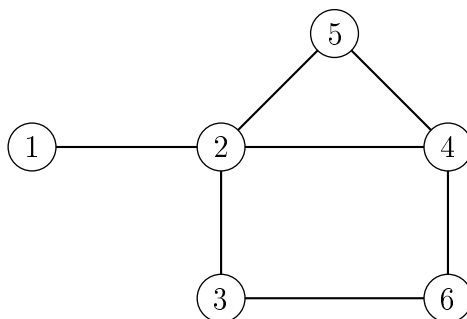


FIGURE 1.9 – Exemple de graphe

La figure 1.10 déroule⁹ une exécution pas à pas de l'algorithme 2 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.

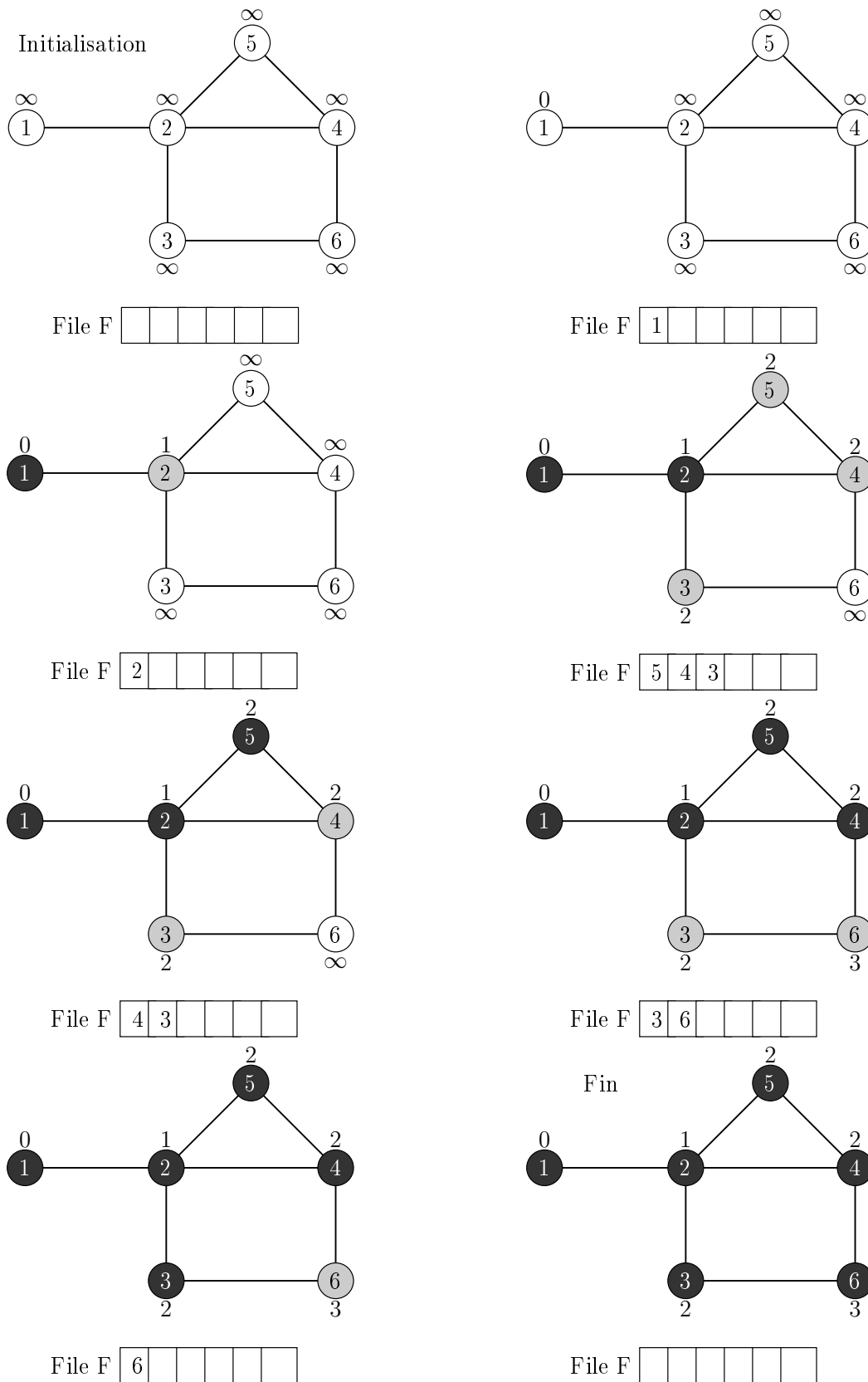


FIGURE 1.10 – Exemple de parcours en largeur d'un graphe à partir du sommet 1

L'algorithme 1 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 2 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 *père* 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 2 : parcours en largeur d'un graphe avec calcul de distance

Données : f : file des sommets en cours **initialisé à** file vide
 $couleur[u]$: $u \in \{BLANC, GRIS, NOIR\}$, **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    $d[s] \leftarrow 0$ 
4    $f.add(s)$ 
5   tant que  $\neg f.empty()$  faire
6      $u \leftarrow f.poll()$ 
7     pour chaque sommet  $v$  voisin de  $u$  faire
8       si  $couleur[v] = BLANC$  alors
9          $couleur[v] \leftarrow GRIS$ 
10         $d[v] \leftarrow d[u] + 1$ 
11         $pere[v] \leftarrow u$ 
12         $f.add(v)$ 
13     $couleur[u] \leftarrow NOIR$ 
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.

À noter également que l'arbre du parcours est conservé dans le sens *fil*s \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.

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'arborences 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 3 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 3 : Parcours en profondeur du graphe $G(V, E)$ à partir du sommet s

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

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

De même que l'algorithme 1, l'algorithme 3 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 1 a été utilisé pour calculer les liens $fil\ s \rightarrow p\ ère$ d'un parcours en largeur, l'algorithme 3 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 4 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 4 : 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 ( $v \neq s$ ) faire
6        $p.push(v)$ 
7        $v = pere[v]$ 
8      $l.add(v)$ 
9     tant que  $\neg p.empty()$  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 : $\Gamma^*(x) = \{x\} \cup \Gamma(x) \cup \Gamma(\Gamma(x)) \dots$ 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 un 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 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

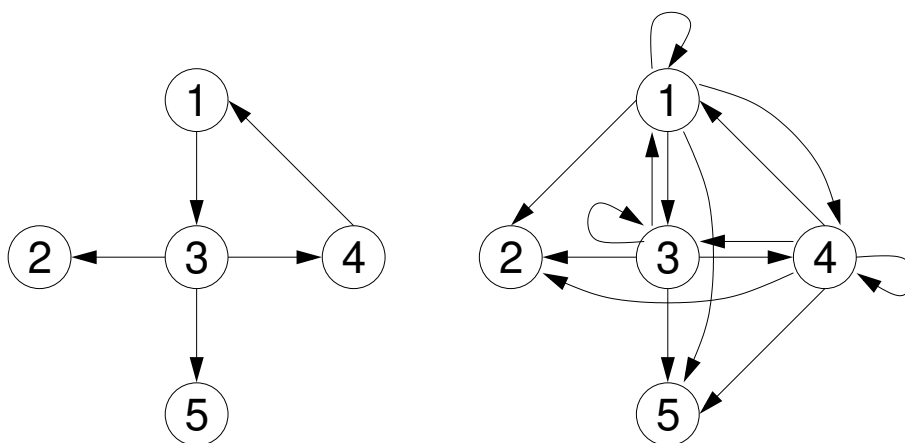


FIGURE 1.11 – Exemple de graphe G associé à sa fermeture transitive G^*

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

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

$$M^* = M + M^2 + M^3 + \dots + M^{n-1}$$

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 M^* 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 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 de villes et des distances entre ces 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

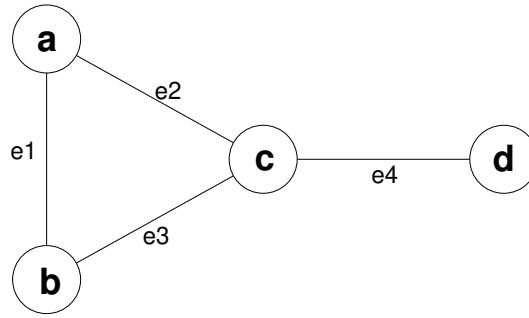


FIGURE 1.12 – Exemple de graphe avec des arêtes pondérées

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'ordonnement (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 M , les cases $M[s, v]$ de la matrice M donnent la distance entre deux sommets s et v connectés directement. Dans ce cas, on prend comme convention $M[s, v] = +\infty$ si les sommets s et v ne sont pas connectés. L'algorithme générique 5 matriciel de Floyd-Warshall permet alors de calculer toutes les distances entre chaque couple de sommets.

Algorithme 5 : Algorithme matriciel de Floyd-Warshall pour un graphe $G(V, E)$

Données : M : 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  $((M[x, z] \neq +\infty) \wedge (M[z, y] \neq +\infty) \wedge$ 
7            $(M[x, z] + M[z, y]) < M[x, y])$  alors
8            $M[x, y] \leftarrow M[x, z] + M[z, y]$ 
9            $PCC[x, y] \leftarrow z$ 

```

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 5 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 5, 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 6 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 6 : 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 $x \forall x \in V$

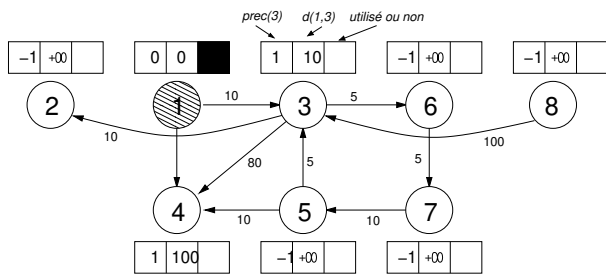
$prec[x]$: le sommet précédent x sur le plus court chemin de x_0 à x

X : l'ensemble des sommets x ayant une valeur pour $l[x]$ définitive

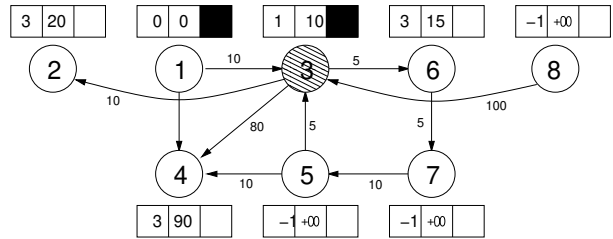
Y : le complémentaire de X dans V ($V = X \cup Y$)

```
1 Fonction plusCourtChDijkstra( $G, x_0$ )
2 début
3    $pcd[x_0] \leftarrow 0$ 
4    $X \leftarrow \{x_0\}$ 
5    $Y = V \setminus \{x_0\}$ 
6   pour chaque sommet  $x$  de  $G \neq x_0$  faire
7      $pcd[x] \leftarrow +\infty$ 
8      $prec[x] \leftarrow -1$ 
9   pour chaque sommet  $y$  voisin de  $x_0$  faire
10     $pcd[y] \leftarrow d(x_0, y)$ 
11     $prec[y] \leftarrow x_0$ 
12  tant que  $\exists y \in Y$  tel que  $pcd[y] < +\infty$  faire
13    prendre  $y \in Y$  tel que  $pcd[y]$  soit minimum
14    pour chaque  $z \in Y \cap \Gamma(y)$  faire
15      si  $pcd(z) > pcd(y) + d(y, z)$  alors
16         $pcd[z] \leftarrow pcd[y] + d(y, z)$ 
17         $prec[z] \leftarrow y$ 
18     $X \leftarrow X \cup \{y\}$ 
19     $Y \leftarrow Y \setminus \{y\}$ 
```

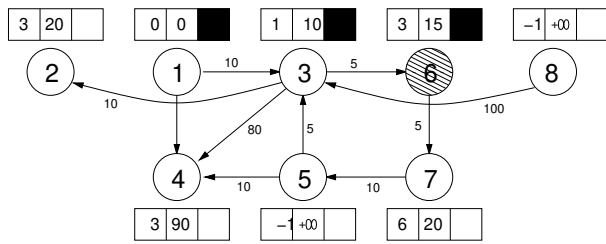
À la fin de l'algorithme, X contient tous les sommets x avec $pcd[x] < +\infty$ et Y contient tous les sommets inaccessibles depuis x_0 . Il est possible de construire les plus courts chemins au départ de x_0 grâce à $prec[]$. La figure 1.13 montre un exemple d'étiquetage des sommets d'un graphe pour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



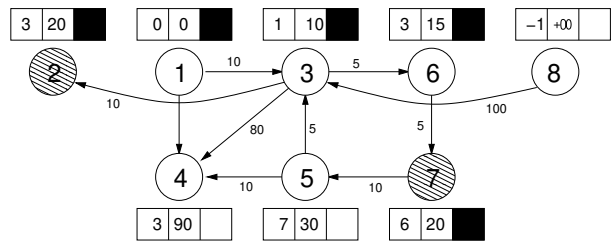
(a) Recherche du plus court chemin à partir de **1**



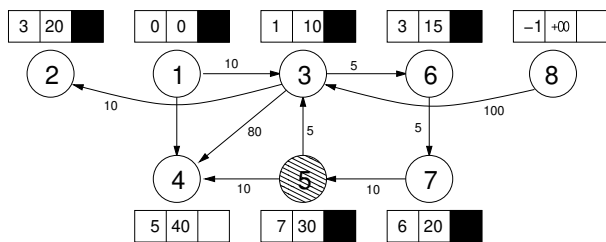
(b) **3** est le sommet le plus proche de **1**. On recalcule les plus courtes distances à **1**. On calcule les distances pour les voisins de **3**.



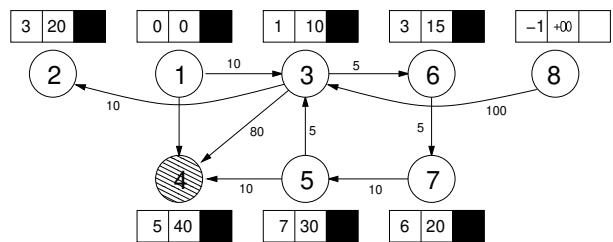
(c) **6** est le plus proche de **1** (non marqué). Modification des valeurs associées au sommet **7**.



(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 3 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.

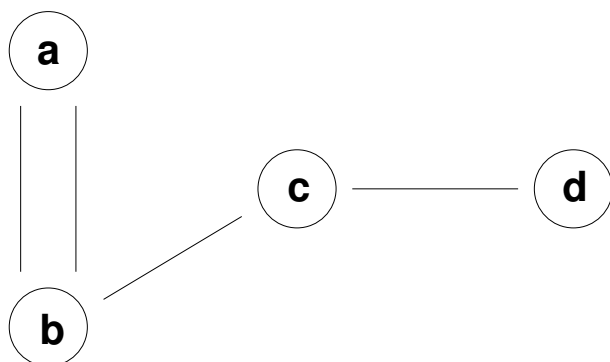


FIGURE 1.14 – Exemples de graphe connexe

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 7 propose, 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 7 : Numérotation en composantes connexes d'un graphe $G(V, E)$

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** *composanteConnexeNOrientee*(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 com-

posantes connexes est ensuite possible. L'algorithme générique 8 propose un calcul de la numérotation en composantes connexes pour un graphe orienté.

Algorithme 8 : Algorithme de numérotation en composantes connexes d'un graphe

Données : $n \leftarrow 0$: numéro de la composante connexe courante

$corresp[u]$: table de correspondance composantes, **initialisée à** $\#u \forall u \in G$

Résultat : $cc[u]$: table des numéros de composantes, **initialisée à** $0 \forall u \in V$

```

1 Fonction composanteConnexeOriente( $G$ )
2 début
3   pour chaque  $x \in G$  non encore visité faire
4      $n \leftarrow n + 1$ 
5     pour chaque sommet  $y$  appartenant au parcours en largeur issu de  $x$  faire
6       si  $cc[y] \neq 0$  alors
7          $corresp[n] \leftarrow cc[y]$  /*  $n > cc[y]$  par construction ne pas poursuivre
8           l'exploration à partir de  $y$  */
9       sinon
10         $cc[y] \leftarrow n$ 
11   pour  $i$  de 1 à la taille de corresp faire  $corresp[i] \leftarrow corresp[corresp[i]]$ 
12   pour chaque sommet  $x$  de  $G$  faire  $cc[x] \leftarrow corresp[cc[x]]$ 

```

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 9 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 9 : 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 composantesFCConnexes( $G$ )
2 début
3   calcul de  $G^{-1}$ 
4   pour chaque  $s \in V$  tel que  $cfc[s] = 0$  faire
5      $parcours \leftarrow$  sommets du parcours de  $G$  à partir  $s$ 
6      $parcours^{-1} \leftarrow$  sommets du parcours de  $G^{-1}$  à partir  $s$ 
7      $ncfc \leftarrow ncfc + 1$ 
8     pour chaque  $x \in parcour \cap parcours^{-1}$  faire  $cfc[x] \leftarrow ncfc$ 
```

Nous cherchons maintenant à évaluer la complexité de cet algorithme. La recherche des sommets accessibles à partir de s (descendants de s) coûte $O(N + M)$ (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 (lignes 26 et 27), 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 petite 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.

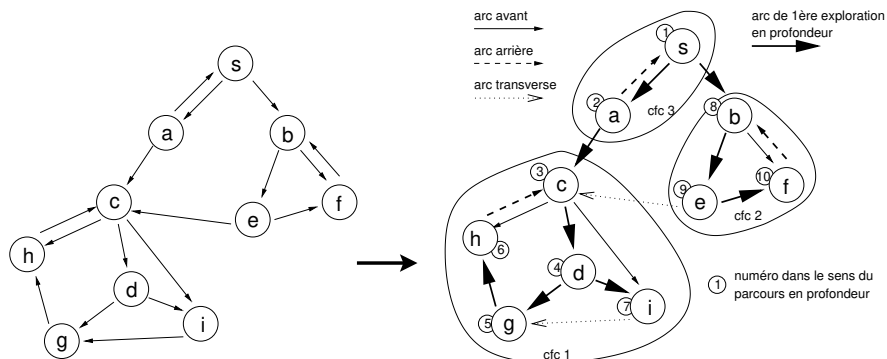


FIGURE 1.15 – Numérotation en composantes fortement connexes d'un graphe $G(V, E)$ par l'algorithme de Tarjan

Algorithme 10 : 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  $s \in G$  tel que  $nv[s] = 0$  faire
3      $npp++$  ;
4      $nv[s] \leftarrow npp$ ;
5      $pPetit[s] \leftarrow npp$ ;
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  $\neg q.empty()$  faire
9        $x \leftarrow q.peek()$  ;
10      si  $x$  n'a plus de voisin à découvrir alors
11        si  $pPetit[x] = nv[x]$  alors
12          /*  $x$  est l'entrée de la cfc courante */
13           $ncfc++$  /* nouvelle composante fortement connexe */
14          répéter
15             $y \leftarrow p.pop()$  ;
16             $cfc[y] \leftarrow ncfc$ 
17          jusqu'à  $y = x$ ;
18           $x \leftarrow q.pop()$  ;
19          si  $\neg q.empty()$  alors
20            /* propagation du num du plus vieil ancêtre commun */
21             $pPetit[q.peek()] \leftarrow \min(pPetit[p.peek()], pPetit[x])$ 
22          sinon
23             $y \leftarrow voisinSuivant[x]$  /* 1er voisin non visité depuis  $x$  */
24            si  $nv[y] = 0$  alors
25              /* 1ère rencontre avec ce sommet */
26               $npp++$  ;
27               $nv[y] \leftarrow npp$ ;
28               $pPetit[y] \leftarrow npp$  /* car il n'y a pas forcément de retour vers la
29                cfc de  $x$  */
30               $p.push(y)$  /*  $y$  est dans la cfc courante */
31               $q.push(y)$  /* il faudra parcourir les voisins de  $y$  */
32            sinon
33              si  $(nv[y] < nv[x]) \wedge (y \in p)$  alors
34                /*  $y$  visité avant  $x$  et  $\in$  à la cfc courante.  $(x, y)$  est un arc
35                  transverse ou arrière dans la même cfc */
36                 $pPetit[x] \leftarrow \min(pPetit[x], nv[y])$ 
```

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, connexe¹¹ et acyclique.

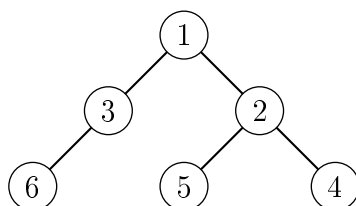


FIGURE 1.16 – Exemple d'arbre

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.

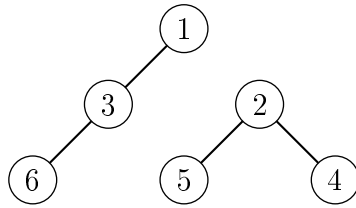


FIGURE 1.17 – Exemple de forêt

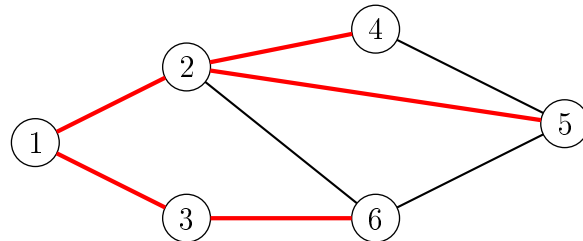


FIGURE 1.18 – Exemple d'arbre couvrant

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 enraciné¹²) est un arbre dont on distingue un sommet particulier, la racine.

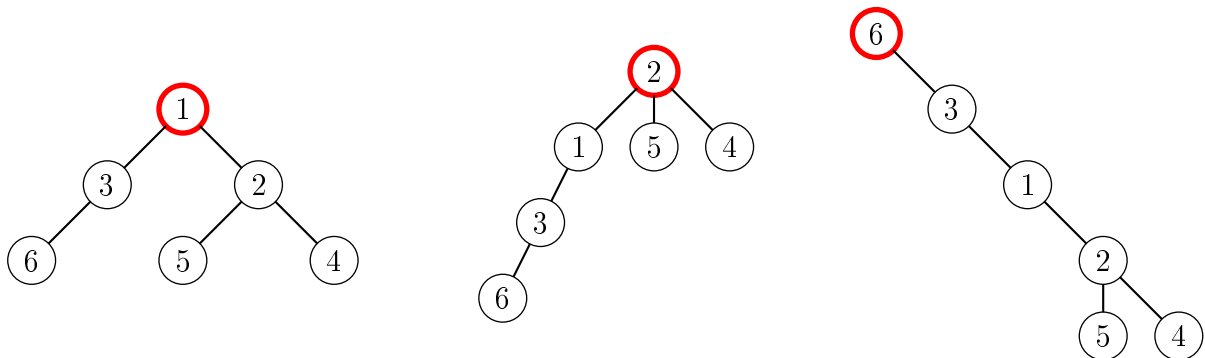


FIGURE 1.19 – Exemples d'arborescences

La figure 1.19 montre différentes 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. À 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 (ou 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.

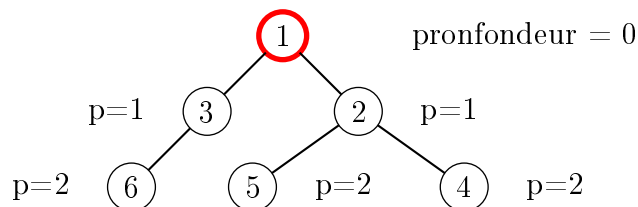


FIGURE 1.20 – Profondeurs dans une arborescence

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 seront 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.

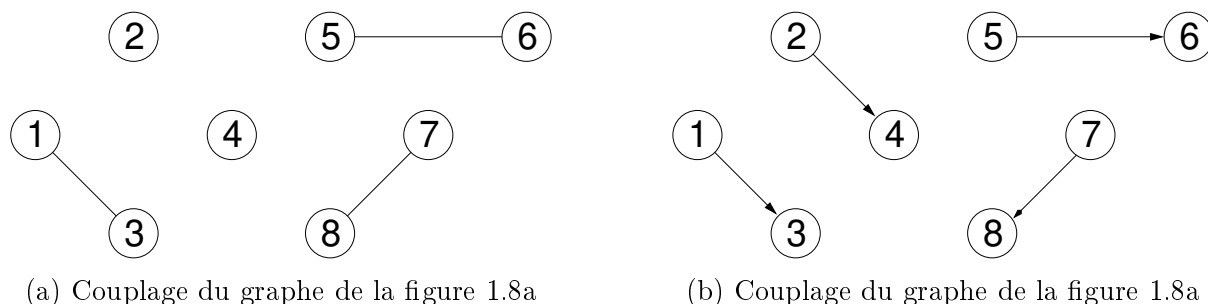


FIGURE 1.21 – Exemples de couplages

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 : Graphe biparti

Un graphe $G(V, E)$ est *biparti* s'il existe une partition¹³ de l'ensemble des sommets V en deux sous-ensembles (classes) V_1 et V_2 telles que tout élément de E a une extrémité dans V_1 et l'autre dans V_2 .

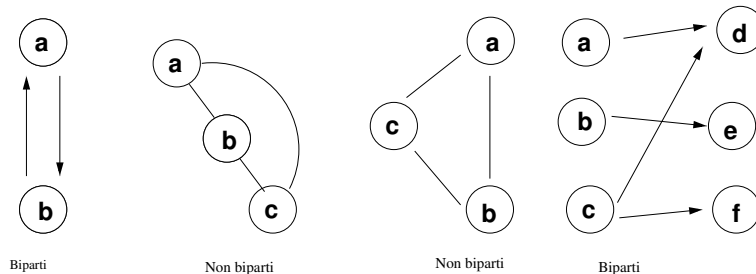


FIGURE 1.22 – Exemple de graphes bipartis et non-bipartis

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.

A noter qu'il est toujours possible de définir une partition en deux de l'ensemble des sommets, même si le graphe n'est pas biparti. Dans ce cas certains arcs ont leurs deux extrémités dans un même sous-ensemble, cette partition définit alors une coupe.

Définition 1.37 : 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 $(p = q = 1)$.

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

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 .

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'ordonnement, 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.

Les chapitres suivants du cours abordent la résolution de certains problèmes combinatoires. Les problèmes combinatoires sont des problèmes dont les solutions sont discrètes, indépendantes les unes des autres. Par exemple, trouver un plus court chemin entre deux sommets u et v dans un graphe est un problème combinatoire car il consiste, à la base¹⁴, 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. A noter que l'ensemble des solutions d'un problème combinatoire est, en général, un ensemble fini. Par exemple, il y a un nombre fini de chemins qui relient deux sommets, si nous excluons les chemins avec boucles qui ne présentent bien sûr pas d'intérêt puisqu'il ajoutent une distance inutile. Si c'est 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 une feuille 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 certaine valeur de n .

La résolution de problèmes combinatoire est souvent en lien avec le 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 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 ou trouver une approche permettant de réduire le nombre de solutions à explorer, mais ce n'est pas toujours possible.

14. 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.

Chapitre 2

Programmation Linéaire

La programmation linéaire a pour but de résoudre un problème d'optimisation avec *maximisation* ou *minimisation* d'une fonction objectif fidèlement à 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 d'une fonction linéaire. 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.

La programmation linéaire en général ne fait pas partie de l'optimisation discrète si les valeurs des solutions sont des valeurs réelles car l'ensemble des solutions est continu. Par contre la programmation linéaire en nombres entiers fait, elle, partie de l'optimisation discrète 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 réelles dont la résolution est polynomiale.

2.1 Exemple de problème linéaire

Enfin de bien comprendre ce qu'est un problème d'optimisation linéaire nous commençons par un exemple. Le problème traité vise à maximiser le nombre de voix récoltées lors d'élections tout en minimisant les sommes engagées dans des actions.

Dans la circonscription concernée par ce scrutin, il existe trois zones bien distinctes :

1. le centre ville : 100 000 inscrits
2. la banlieue : 200 000 inscrits
3. la campagne : 50 000 inscrits

L'objectif à atteindre est la majorité des voix dans chacune des trois zones afin de remporter les élections.

Nous supposons qu'il existe un certain nombre d'actions pouvant influencer le vote des électeurs en fonction de l'argent investi sur les thèmes suivants :

- construction de routes (1);

- lutte contre l'insécurité (2);
- subvention aux agriculteurs (3);
- amélioration de la qualité de l'eau (4).

Pour chacun de ces thèmes, on compte un certain nombre de voix *gagnées/perdues* en fonction des sommes investies. Le tableau suivant donne le nombre de voix gagnées (nombre positif) ou perdues (nombre négatif) par 1 000€ dépensés sur cette action.

thèmes	ville	banlieue	campagne
les routes	-2 000	+5 000	+3 000
l'insécurité	+8 000	+2 000	-5 000
subventions	0	0	+10 000
amélioration de l'eau	10 000	0	-2 000

Le problème est donc de connaître la somme minimale à investir pour que le nombre de voix en faveur du candidat soit de :

- 50 000 en ville ;
- 100 000 en banlieue ;
- 25 000 à la campagne.

Nous pouvons essayer de tâtonner pour chercher à garantir les contraintes liées au succès de l'élection en minimisant les sommes engagées. L'exemple suivant donne le nombre de votes par secteur en fonction des sommes investies sur les différents thèmes :

- | | | |
|---|---|---|
| <ul style="list-style-type: none"> — 20 000 € pour le thème (1); — 0 € pour le thème (2); — 4 000 € pour le thème (3); — 9 000 € pour le thème (4); | ⇒ | <ul style="list-style-type: none"> — 50 000 votes en ville ; — 100 000 votes en banlieue ; — 82 000 votes à la campagne. |
|---|---|---|

⇒ 33 000 € en investissement.

Dans cette solution nous voyons que le nombre de voix à atteindre est correct pour la ville et la banlieue mais qu'il serait possible d'investire moins puisque le nombre de voix de la campagne dépasse l'objectif. L'exploration des valeurs possibles pour les sommes engagées devrait donc nous permettre d'apporter une meilleure solution. Cette exploration n'est cependant pas si simple puisque le résultat dépend de l'ensemble des paramètres et pour être sûr d'atteindre l'objectif optimal, nous devrions explorer toutes les valeurs possibles, de manière combinatoire.

Pour trouver cette solution, il est cependant possible de formaliser le problème sous forme mathématique :

- soit x_1 la somme dépensée pour le thème (1);
- soit x_2 la somme dépensée pour le thème (2);
- soit x_3 la somme dépensée pour le thème (3);
- soit x_4 la somme dépensée pour le thème (4);

En conséquence, les contraintes de réussite pour le candidat sont les suivantes, pour les 3 secteurs :

$$\left\{ \begin{array}{l} \text{contraintes en ville} \\ \text{contraintes en banlieue} \\ \text{contraintes à la campagne} \end{array} \right. \quad \begin{array}{r} -2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50 \\ 5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100 \\ 3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25 \end{array}$$

Et la fonction à minimiser est $x_1 + x_2 + x_3 + x_4$ avec $x_i \geq 0$ pour $i = 1, 2, 3$ et 4 .

► L'ensemble de ces équations forment un **programme linéaire**.

2.2 Problème général

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 $f(x_1, x_2, \dots, x_n) \leq b$ ou $\geq b$ (inégalité au sens large en programmation linéaire), où f est une fonction linéaire donc de la forme : $f(x_1, x_2, \dots, x_n) = \sum_{j=1}^n a_j x_j$
- Un objectif d'optimisation exprimé sous la forme d'une fonction linéaire $f(x_1, x_2, \dots, x_n)$.

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 se fait par la méthode du *simplexe* qui utilise la forme standard.

2.2.2 Principe de résolution

Soit l'exemple de programme linéaire à 2 variables suivant :

$$\left\{ \begin{array}{l} \text{maximiser} \quad \quad \quad x_1 + x_2 \\ \text{avec les contraintes} \quad 4x_1 - x_2 \leq 8 \\ \quad \quad \quad \quad \quad 2x_1 + x_2 \leq 10 \\ \quad \quad \quad \quad \quad 5x_1 - 2x_2 \geq -2 \\ \text{et } x_1, x_2 \geq 0 \end{array} \right.$$

Il est possible de résoudre ce programme linéaire graphiquement car il n'a que 2 dimensions. Les contraintes permettent de définir des partitions du plan en 5 demi-plans. En faisant l'intersection de ces demi-plans compatibles avec les contraintes, une zone *réalisable* se définit. 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 2.1 illustre cette résolution. Les solutions de ce programme linéaire sont $x_1 = 2$ et $x_2 = 6$. 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.

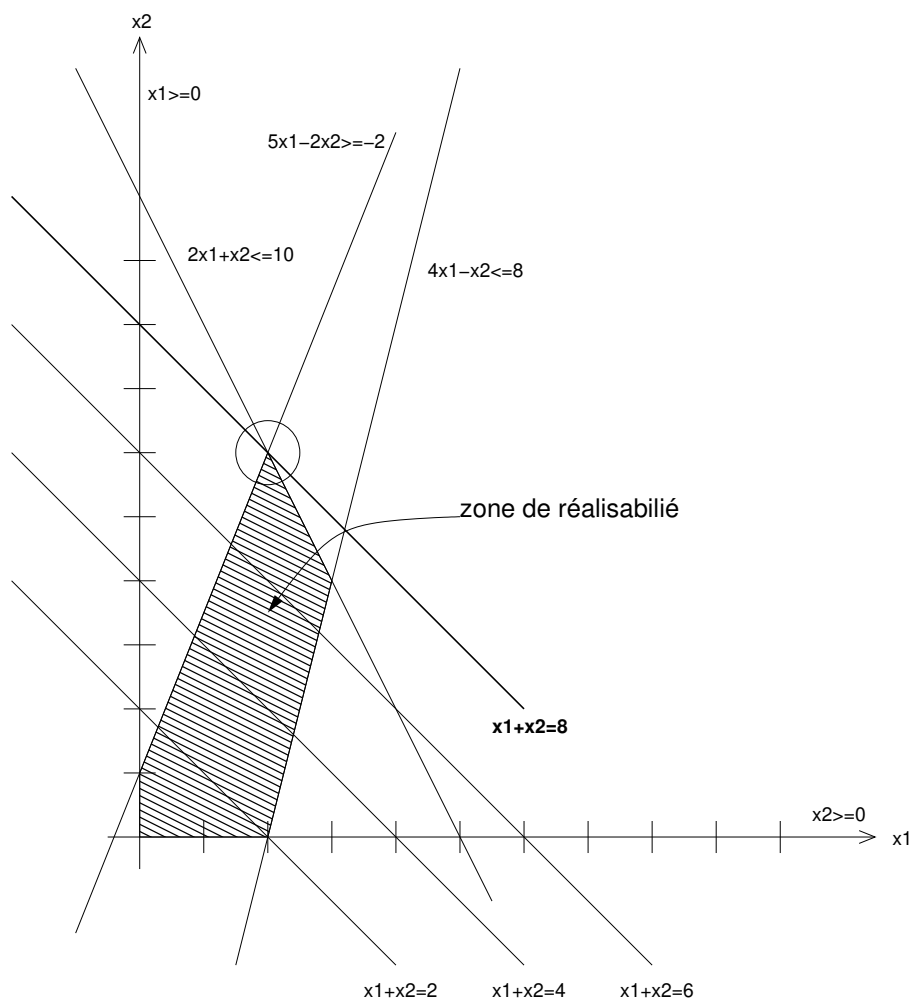


FIGURE 2.1 – Résolution graphique d'un programme linéaire à deux dimensions

En fait, la démarche est la même pour un programme linéaire avec un plus grand nombre de variables, même si dans ce cas, une représentation graphique n'a plus de sens.

La région de réalisabilité est formée, comme il a été suggéré précédemment, par l'intersection des demi-espaces définis dans les contraintes du programme linéaire. Cette région convexe est appelée *simplexe*. 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 , 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.

2.2.3 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 ;
- ...

2.2.4 Algorithme de résolution

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.

L'algorithme de *l'ellipsoïde* est linéaire. Par contre 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.

2.3 Forme canonique et forme standard

2.3.1 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) :

$$\left\{ \begin{array}{l} \text{maximiser} \\ \text{avec les contraintes} \\ \text{et les contraintes de positivité} \end{array} \quad \begin{array}{l} \sum_{j=1}^n c_j x_j \text{ (fonction objectif)} \\ \sum_{j=1}^n a_{ij} x_j \leq b_i \text{ avec } i = 1, \dots, m \\ x_j \geq 0 \text{ avec } j = 1, \dots, n \end{array} \right.$$

2.3.2 Conversion d'un programme linéaire sous forme canonique

Soit le programme linéaire suivant à transformer sous forme canonique :

$$\left\{ \begin{array}{l} \text{minimiser} \\ \text{avec les contraintes} \end{array} \quad \begin{array}{r} -2x_1 + 3x_2 \\ x_1 + x_2 = 7 \\ x_1 - 2x_2 \leq 4 \\ x_1 \geq 0 \end{array} \right.$$

Minimiser la fonction objectif : Minimiser une fonction linéaire est équivalent à maximiser la fonction linéaire opposée. Il suffit donc de tout multiplier par -1 .

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 :

$$\left\{ \begin{array}{l} \text{maximiser} \\ \text{avec les contraintes} \\ \text{et} \end{array} \right. \begin{array}{r} 2x_1 - 3x'_2 + 3x''_2 \\ x_1 + x'_2 - x''_2 = 7 \\ x_1 - 2x'_2 + 2x''_2 \leq 4 \\ x_1, x'_2, x''_2 \geq 0 \end{array}$$

Transformation des égalités Si 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 :

$$x_1 + x'_2 - x''_2 = 7 \Rightarrow \left\{ \begin{array}{l} x_1 + x'_2 - x''_2 \leq 7 \\ -x_1 - x'_2 + x''_2 \leq -7 \end{array} \right.$$

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

$$\left\{ \begin{array}{l} \text{maximiser} \\ \text{avec les contraintes} \\ \text{et} \end{array} \right. \begin{array}{r} 2x_1 - 3x_2 + 3x_3 \\ x_1 + x_2 - x_3 \leq 7 \\ -x_1 - x_2 + x_3 \leq -7 \\ x_1 - 2x_2 + 2x_3 \leq 4 \\ x_1, x_2, x_3 \geq 0 \end{array}$$

2.3.3 Conversion d'un programme linéaire 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 2.3.2 s'écrit donc comme suit :

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

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 :

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

Forme générale d'un programme linéaire standard

On pose N l'ensemble des indices des variables *hors base* ($|N| = n$) et B l'ensemble des indices des variables de *base* ($|B| = m$). On pose également la matrice A comme coefficients a_{ij} des variables hors base et les vecteurs c et b pour les coefficients c_j de la fonction objectif et pour les constantes b_i des égalités linéaires. Ainsi, de manière générale, un programme linéaire s'écrit de la manière suivante :

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

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

Exemple

Soit le programme linéaire suivant :

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

On a alors pour (N, B, A, b, c, v) les formes suivantes :

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

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

2.4 Algorithme du simplexe

Cet algorithme 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 qui ont un coefficient positif 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, 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.

2.4.1 Exemple d'exécution de l'algorithme du simplexe

Soit le programme suivant écrit sous sa forme canonique :

$$\left\{ \begin{array}{l} \text{maximiser} \\ \text{avec les contraintes} \end{array} \right. \begin{array}{r} 3x_1 + x_2 + 2x_3 \\ x_1 + x_2 + 3x_3 \leq 30 \\ 2x_1 + 2x_2 + 5x_3 \leq 24 \\ 4x_1 + x_2 + 2x_3 \leq 36 \\ \text{et} \quad x_1, x_2, x_3 \geq 0 \end{array}$$

La forme standard de ce programme linéaire est la suivante :

$$\left\{ \begin{array}{l} z = 3x_1 + x_2 + 2x_3 \\ x_4 = 30 - x_1 - x_2 - 3x_3 \\ x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \\ x_6 = 36 - 4x_1 - x_2 - 2x_3 \end{array} \right.$$

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

Solution de base

Pour trouver les solutions de base, il faut mettre 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 :

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

Dans ce cas, la valeur de l'objectif est $z = 0$ et $\overline{x_i} = b_i \forall i \in B$. 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 avoir un ensemble de variables de base différent de celui de la solution de base sans changer le programme linéaire initial, il faut le réécrire dans une autre forme. Comme il a déjà été précisé dans la présentation de l'algorithme en début de ce paragraphe, on itère sur ces transformations jusqu'à ce qu'il ne soit plus possible d'améliorer la fonction objectif.

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 :

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

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

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

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 :

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

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 :

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

On passe donc x_3 du côté des variables de base et on passe x_5 du côté 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 :

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

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

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

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 :

$$x_2 > 132 \Rightarrow x_1 < 0$$

$$x_2 > 4 \Rightarrow x_3 < 0$$

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 :

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

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'issue 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 :

$$\underbrace{x_1 + x_2 + x_3}_{12} \leq 30$$

$$12 \leq 30$$

NB : Attention les solutions ne sont pas toujours entières.

2.4.2 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 11 illustre le calcul des nouveaux indices.

Algorithme 11 : Algorithme du pivot pour la réécriture de la nouvelle forme du programme linéaire compte tenu des deux variables entrante et sortante.

```

1   $(N, B, A, b, c, v)$  : le programme linéaire courant sous sa forme standard
2   $l$  : l'indice de la variable sortante
3   $e$  : l'indice de la variable entrante
4  retourner  $(\widehat{N}, \widehat{B}, \widehat{A}, \widehat{b}, \widehat{c}, \widehat{v})$ 
5  Fonction pivot( $N, B, A, b, c, v, l, e$ )
6  début
   |   /* coefficients de l'équation pour la nouvelle variable de base */
7   |    $\widehat{b}_e \leftarrow b_l/a_{le}$ 
8   |   pour  $j \in N \setminus \{e\}$  faire
9   |   |    $\widehat{a}_{ej} \leftarrow a_{lj}/a_{le}$ 
10  |    $\widehat{a}_{el} \leftarrow 1/a_{le}$ 
   |   /* coefficients des contraintes restantes */
11  |   pour  $i \in B \setminus \{l\}$  faire
12  |   |    $\widehat{b}_i \leftarrow b_i - a_{ie} \times \widehat{b}_e$ 
13  |   |   pour  $j \in N \setminus \{e\}$  faire
14  |   |   |    $\widehat{a}_{ij} \leftarrow a_{ij} - a_{ie} \times \widehat{a}_{ej}$ 
15  |   |   |    $\widehat{a}_{il} \leftarrow a_{ie} \times \widehat{a}_{el}$ 
   |   /* calcul de la fonction objectif */
16  |    $\widehat{v} \leftarrow v + c_e \times \widehat{b}_e$ 
17  |   pour  $j \in N \setminus \{e\}$  faire
18  |   |    $\widehat{c}_j \leftarrow c_j - c_e \times \widehat{a}_{ej}$ 
19  |    $\widehat{c}_l \leftarrow -c_e \times \widehat{a}_{el}$ 
   |   /* ensemble des variables de base et hors base */
20  |    $\widehat{N} \leftarrow N \setminus \{e\} \cup \{l\}$ 
21  |    $\widehat{B} \leftarrow B \setminus \{l\} \cup \{e\}$ 

```

2.4.3 Calcul du simplexe

L'algorithme 12 montre comment on itère sur la réécriture du programme linéaire jusqu'à ce que la fonction objectif ne puisse plus être augmentée.

Algorithme 12 : Algorithme du calcul du simplexe.

```
1  $(A, b, c)$  : le programme linéaire courant sous sa forme canonique
2 retourner  $(\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n)$ 
3 Fonction simplexe( $A, b, c$ )
4 début
5    $(N, B, A, b, c, v) \leftarrow \text{initialiseSimplexe}(A, b, c)$ 
6   tant que  $\exists j \in N$  vérifiant  $c_j > 0$  faire
7     choisir  $e \in N$  avec  $c_e > 0$ 
8     pour chaque  $i \in B$  faire
9       si  $a_{ie} > 0$  alors
10        |  $\Delta_i \leftarrow b_i/a_{ie}$ 
11       sinon
12        |  $\Delta_i \leftarrow +\infty$ 
13     choisir  $l \in B$  qui minimise  $\Delta_i$ 
14     si  $\Delta_i = +\infty$  alors
15       | retour "non borné"
16     sinon
17       |  $(N, B, A, b, c, v) \leftarrow \text{pivot}(N, B, A, b, c, v, l, e)$ 
18   pour  $i$  de 1 à  $n$  faire
19     si  $i \in B$  alors
20       |  $\bar{x}_i \leftarrow b_i$ 
21     sinon
22       |  $\bar{x}_i \leftarrow 0$ 
```

2.5 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 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.

Par exemple, 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 fixé¹ 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 pour 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 solution peut-être calculée à partir de sous-solutions. Par exemple, dans la recherche d'un plus court chemin

Or, 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.

Ce principe a été introduit par Richard Bellman et utilisé dans l'algorithme de Bellman-Ford pour la recherche d'un plus court chemin. Il est généralement appelé principe d'optimalité de Bellman. 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 . 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. En effet, 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 du graphe grandit. Pour résoudre de problème la programmation dynamique utilise la technique de 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 posé le problème suivant² : "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 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 cette définition nous pouvons écrire l'algorithme 13 qui donne un programme récursif de calcul de la suite de Fibonacci.

Algorithme 13 : 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 **FiboModifRec(5)**, nous obtenons l'arbre suivant :

La figure 3.1

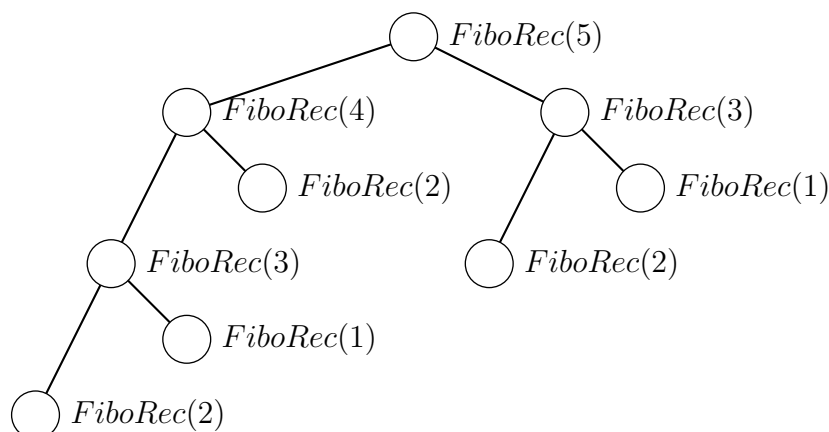


FIGURE 3.1 – Arbre du calcul de l'appel **FiboModifRec(5)**

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 *FiboModifRec(1)* et *FiboModifRec(2)*, 2 pour *FiboModifRec(3)*, 4 pour *FiboModifRec(4)*, 8 pour *FiboModifRec(5)*, et donc 2^{n-2} pour *FiboModifRec(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 mémorisation 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.

Algorithme 14 : 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    $FiboVal[1] \leftarrow 1$ 
5    $FiboVal[2] \leftarrow 1$ 
6   pour  $i \leftarrow 3$  à  $n$  faire  $FiboVal[i] \leftarrow FiboVal[i - 1] + FiboVal[i - 2]$ 
7   retourner  $FiboVal[n]$ 
```

L'algorithme 14 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 *tabloVal*.

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 15.

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

Algorithme 15 : Calcul de la plus courte distance par récurrence

```
1 Fonction  $PCDRec(u, v)$   
2  $d[\ ][\ ]$  : tableau des distances  
3 début  
4   si  $u = v$  alors retourner 0  
5    $pcd \leftarrow \infty$   
6   pour chaque sommet  $w$  voisin de  $v$  faire  
7     si  $pcd > PCDRec(u, w) + d(w, v)$  alors  $pcd \leftarrow PCDRec(u, w) + d(w, v)$ 
```

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 .

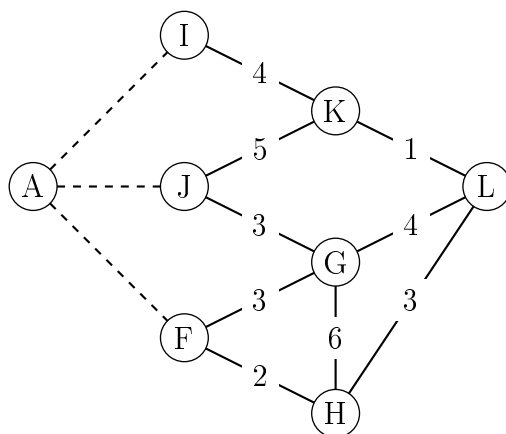


FIGURE 3.2 – Exemple de graphe avec des distances

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 mémorisation 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 14.

Pour la 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 16 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 sommets du graphe mais peut aussi s'arrêter avant si la file qu'il utilise est vide.

Algorithme 16 : Calcul de la plus courte distance entre deux sommets d'un graphe

Données : n : le nombre de sommets du graphe

$d(x, y)$: la distance entre x et y , $\forall \{x, y\} \in E$

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 PCD( $u, v$ )
2 début
3    $pcd[u] \leftarrow 0$ 
4   pour chaque sommet  $w \neq u$  de  $V$  faire  $pcd[w] \leftarrow +\infty$ 
5    $nIter \leftarrow 1$  /*                               nombre d'itérations */
6    $f.add(u)$ 
7   tant que  $(nIter < n) \wedge (f.get() \neq null)$  faire
8      $sommetCourant \leftarrow f.get()$ 
9     pour chaque sommet  $w$  voisin de  $sommetCourant$  faire
10      si  $pcd[w] > pcd[sommetCourant] + d(sommetCourant, w)$  alors
11         $pcd[w] \leftarrow pcd[sommetCourant] + d(sommetCourant, w)$ 
12      if  $\neg f.contains(w)$  then  $f.add(w)$ 
13       $nIter \leftarrow nIter + 1$ 
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 16, 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 17 : 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    $pcd[u] \leftarrow 0$ 
4    $prec[u] \leftarrow -1$ 
5   pour chaque sommet  $x \neq u$  de  $V$  faire
6      $pcd[x] \leftarrow +\infty$ 
7      $prec[x] \leftarrow -1$ 
8    $nIter \leftarrow 1$  /*          nombre d'itérations          */
9    $modif \leftarrow VRAI$  /*   VRAI si une valeur de  $pcd[y]$  a été modifiée   */
10  tant que  $(nIter < n) \wedge modif$  faire
11     $modif \leftarrow FAUX$ 
12    pour chaque arête  $\{x, y\}$  de  $G$  faire
13      si  $pcd[y] > pcd[x] + d(x, y)$  alors
14         $pcd[y] \leftarrow pcd[x] + d(x, y)$ 
15         $prec[y] \leftarrow x$ 
16         $modif \leftarrow VRAI$ 
17     $nIter \leftarrow nIter + 1$ 

```

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 17 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.

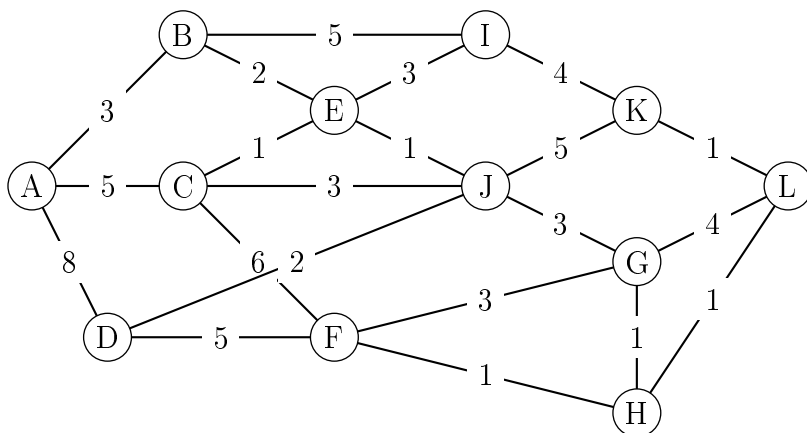


FIGURE 3.3 – Exemple de graphe avec des distances

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 Bilan

Sur cet exemple nous pouvons constater que la programmation dynamique apporte une solution élégante et beaucoup 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. Par exemple, dans les cas de la plus courte distance, il s'agit de montrer que la solution optimale dépend uniquement des sous-solutions des voisins du sommet v .
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).
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 est combinatoire, c'est-à-dire qu'à partir d'un état initial, puis des états intermédiaires, plusieurs possibilités doivent être explorées pour générer la solution. Par exemple, dans une recherche dans un graphe, à partir d'un sommet nous avons le choix entre tous ses voisins, puis à partir des voisins nous avons à nouveau le choix entre tous les voisins, ... Au final, un chemin est le choix d'une suite de voisins parmi l'ensemble des suites possibles ou toutes les combinaisons de voisins possibles. De plus, puisque nous traitons un problème d'optimisation, nous cherchons la meilleure solution. Pour garantir qu'une solution est optimale, il est possible d'explorer l'ensemble des combinaisons pour générer toutes les solutions possibles, les évaluer puis choisir la meilleure. Puisque le problème est combinatoire, la génération de l'ensemble des solutions peut être très coûteuse. Il est donc nécessaire d'établir des stratégies pour réduire le nombre de solutions à explorer.

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 glouton consiste à réaliser le choix qui optimise l'objectif en ne tenant compte que de la situation courante. Par exemple, si nous cherchons 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 donc de manière incrémentale, de choix en 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.

A titre d'exemple, nous considérons le problème de la coloration d'un graphe 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

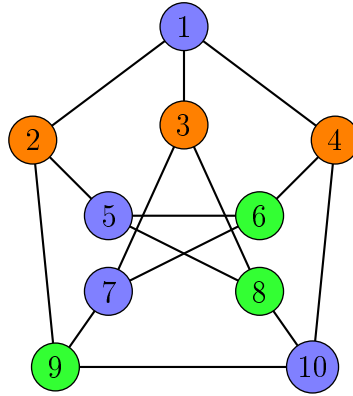
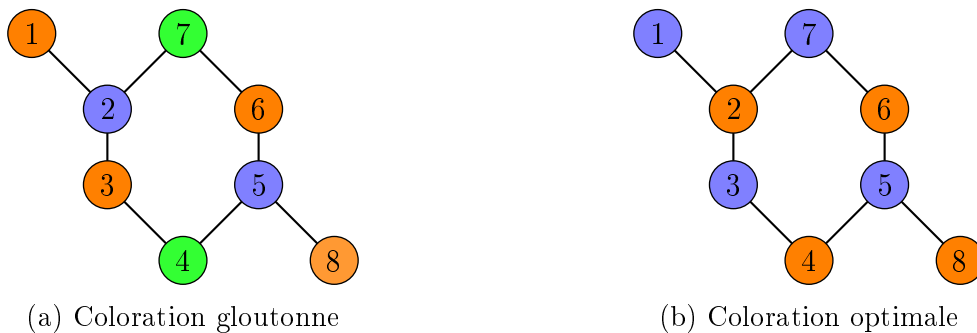


FIGURE 4.1 – Coloration gloutonne du graphe de Petersen à partir du sommet 1

de trouver le nombre minimal de couleurs pour réaliser la coloration d'un graphe est un problème classique d'optimisation combinatoire. Sur la figure 4.1 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 évidemment pas possible dans ce cas.

Dans le cas de la coloration minimale, une stratégie gloutonne consiste à considérer 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. La figure 4.1 donne une coloration obtenue par un algorithme glouton sur un graphe de Petersen. Les sommets ayant tous le même nombre de voisins, nous traitons les sommets 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.

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 verrons le cas de l'arbre couvrant de poids minimal au début de ce chapitre pour illustrer ce propos.



(a) Coloration gloutonne

(b) Coloration optimale

FIGURE 4.2 – Sous-optimalité de l'algorithme glouton de coloration

Cependant les algorithmes gloutons ne donnent pas forcément une solution optimale. C'est le cas pour la coloration de graphes que nous venons de voir. Sur la figure 4.2a, l'algorithme glouton commence par le sommet 2 et utilise le même ordre des couleurs que pour la coloration du graphe de Petersen. Le résultat de l'algorithme glouton donne 3 couleurs alors que le graphe admet une coloration optimale à 2 couleurs comme le montre la figure 4.2b. 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 nécessaire 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 gloutonne n'offre pas toujours des solutions optimale mais elle a un intérêt car elle repose sur des algorithmes simples, polynomiaux qui gèrent 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. C'est le cas que nous traiterons dans la seconde partie de ce chapitre avec le problème du choix des activités.

4.2 Recherche d'un 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 étudions 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 $G = (V, E)$ 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 $w(T)$ est minimal. Le poids de l'arbre est calculé avec : $w(T) = \sum_{(u,v) \in T} w((u,v))$. Ici w est une fonction de pondération des arêtes du graphe.

Sans réflexion préliminaire, trouver une solution 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, c'est-à-dire une stratégie qui à chaque étape fait le choix le moins coûteux, garantisse un résultat optimal.

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 18. 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 18 : Algorithme de Borůvka : recherche de ACPM dans un graphe

```

1  Données :
2  Graphe  $G = (V, E)$ 
3   $w(x) : E \rightarrow \mathbb{R}$  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   $F \leftarrow (V, \emptyset)$ 
11   $cc \leftarrow \text{composanteConnexeNOrientee}(F)$ 
12   $nbcc \leftarrow cc.\text{longueur}$ 
13  tant que  $nbcc > 1$  faire
14  |   pour  $i$  de 1 à  $nbcc$  faire  $ppa[i] \leftarrow \infty$ 
15  |   pour  $e = \{u, v\} \in E$  faire
16  |   |   si  $cc[u] \neq cc[v]$  alors
17  |   |   |   si  $w(e) < w(ppa[cc[u]])$  alors  $ppa[cc[u]] \leftarrow e$ 
18  |   |   |   si  $w(e) < w(ppa[cc[v]])$  alors  $ppa[cc[v]] \leftarrow e$ 
19  |   |   pour  $i$  de 1 à  $nbcc$  faire
20  |   |   |   si  $ppa[i] \neq \infty$  alors  $F \leftarrow F \cup e$ 
21  |   |    $cc \leftarrow \text{composanteConnexeNOrientee}(F)$ 
22  |   |    $nbcc \leftarrow cc.\text{longueur}$ 

```

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 règle 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 $O(M \times \text{Log}(N))$, 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 $O(M \times \text{Log}(N))$.
- 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 $O(M + N \times \text{Log}(N))$.

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.

4.3 Problème du choix d'activités

Le problème du choix d'activité est un problème NP-Complet. C'est-à-dire qu'il n'existe pas de solution ayant une complexité polynomiale pour le résoudre. Ici nous montrons que la stratégie gloutonne peut aussi s'appliquer au cas d'un problème complexe.

4.3.1 Présentation du problème

Le problème du choix d'activité est un problème d'ordonnancement. Les données de ce problème sont les suivantes :

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

- différentes activités et les traite une par une ;
- Si a_i est sélectionnée alors $[s_i, f_i[$ est la plage de réservation de la ressource pour le traitement de a_i ;
 - Les activités a_i et a_j sont dites compatibles si $f_i \leq s_j$ ou $f_j \leq s_i$;
- Le problème du choix des activités est le suivant : il s'agit de choisir le sous ensemble S d'activités compatibles le plus grand possible.

4.3.2 Exemple

Soit le tableau 4.1 qui donne les s_i et les f_i pour les activités a_1 à a_{11} .

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

TABLE 4.1 – Date de début et de fin des activités

À partir de ces données nous pouvons constater que :

- Le sous ensemble $\{a_3, a_9, a_{11}\}$ est correct mais n'est pas le plus grand possible ;
- Par contre $\{a_1, a_4, a_8, a_{11}\}$ est plus grand (taille maximale) ;
- Et $\{a_2, a_4, a_9, a_{11}\}$ est un autre sous ensemble de taille maximale.

4.3.3 Solution avec la programmation dynamique

Il est possible de formuler le problème pour une résolution avec la programmation dynamique :

On exprime les solutions optimales de sous problèmes pour former la solution optimale du problème initial. Pour cela il faut examiner l'espace des sous problèmes.

Soit l'ensemble $S_{i,j}$ défini comme suit :

$$S_{i,j} = \{a_k \in S \mid f_i \leq s_k < f_k \leq s_j\}$$

Il s'agit de l'ensemble des activités qui peuvent commencer après l'activité a_i et qui doivent terminer avant a_j .

- Si $a_k \in S_{i,j}$ alors a_k est compatible avec a_i et a_j .
- Soit les activités fictives a_0 et a_{n+1} avec $f_0 = 0$ et $s_{n+1} = +\infty$.
On a donc $S_{0,n+1} = S$ avec $0 \leq i$ et $j \leq n + 1$.

Résolution sur l'intervalle $[i, j]$

- On suppose que les activités sont triées :

$$f_0 \leq f_1 \leq f_2 \leq \dots \leq f_n < f_{n+1}$$

- $S_{i,j} = \emptyset$ si $j \leq i$. En effet, si $a_k \in S_{i,j}$ telle que $j \leq i$ avec a_i suit a_j on aurait :
 $f_i \leq s_k \leq f_k \leq s_j < f_j$.
 Donc $f_i < f_j$ ce qui est impossible si a_i suit a_j .
- Si on a les activités triées par ordre monotone croissant des heures de fin, on cherche un sous ensemble de taille maximale d'activités mutuellement compatibles prises dans $S_{i,j}$ pour $0 \leq i < j \leq n + 1$, les autres $S_{i,j}$ étant vides.

La sous structure du problème de choix d'activité est :

- Soit $S_{i,j}$ non vide avec $a_k \in S_{i,j}$ telle que

$$f_i \leq s_k < f_k \leq s_j$$

- on a donc la génération des deux sous problèmes $S_{i,k}$ et $S_{k,j}$ tels que les activités qui les composent appartiennent à $S_{i,j}$. On a donc :

$$S_{i,j} = S_{i,k} \cup S_{k,j} \cup \{a_k\}$$

Donc le nombre d'activités compatibles avec $S_{i,j}$ est :

$$|S_{i,j}| = |S_{i,k}| + |S_{k,j}| + 1$$

Sous structure optimale du problème De ces expressions on peut déduire la sous structure optimale de ce problème :

- Soit $A_{i,j}$ une solution optimale de $S_{i,j}$ telle que $a_k \in A_{i,j}$
- Alors les solutions $A_{i,k}$ de $S_{i,k}$ et $A_{k,j}$ de $S_{k,j}$ utilisées dans la solution optimale $A_{i,j}$ de $S_{i,j}$ doivent être optimales.
- Si $A'_{i,k}$ de $S_{i,k}$ est une solution qui contient plus d'activités de $A_{i,k}$ alors on peut remplacer $A_{i,k}$ par $A'_{i,k}$ dans $A_{i,j}$ pour donner une autre solution pour $S_{i,j}$ avec plus d'activités que dans $A_{i,j}$, ce qui est une contradiction.
- On peut reprendre le même raisonnement avec le $A'_{k,j}$ de $S_{k,j}$.

Utilisation de la sous structure optimale On utilise la sous structure optimale définie précédemment pour construire une solution optimale à partir des solutions optimales des sous problèmes :

- Si $S_{i,j} \neq \emptyset$ alors $\exists a_k \in S_{i,j}$ une solution optimale contient en elle des solutions optimales aux sous problèmes $S_{i,k}$ et $S_{k,j}$.
- On peut donc construire une solution optimale (taille maximale d'activités mutuellement compatibles de $S_{i,j}$) en divisant le problème en deux sous problèmes :
 - trouver les sous ensembles de taille maximale d'activités mutuellement compatibles pour $S_{i,k}$ et $S_{k,j}$ en trouvant $A_{i,k}$ et $A_{k,j}$.

$$A_{i,j} = A_{i,k} \cup A_{k,j} \cup \{a_k\}$$

La solution optimale pour le problème entier est une solution de $S_{0,n+1}$.

Solution récursive La définition récursive d'une solution optimale en fonction des sous problèmes optimaux est typiquement la démarche utilisée pour construire une solution basée sur la programmation dynamique :

- Soit $C[i, j]$ le nombre d'activités d'un sous-ensemble de taille maximale d'activités mutuellement compatibles de $S_{i,j}$.

$$C[i, j] = 0 \Rightarrow S_{i,j} = \emptyset$$

- Soit $S_{i,j} \neq \emptyset$ et $a_k \in S_{i,j}$
 - Si $C[i, k]$ et $C[k, j]$ sont respectivement le nombre d'activité d'un sous ensemble de taille maximale d'activités mutuellement compatibles respectivement des ensembles $S_{i,k}$ et $S_{k,j}$, alors $C[i, j]$ s'écrit de la manière suivante :

$$C[i, j] = C[i, k] + 1 + C[k, j]$$

Or il faut connaître la valeur de k . Il faut faire le test pour les $j - i - 1$ valeurs possibles pour k . En effet $k = i + 1, \dots, j - 1$.

- On a donc l'expression suivante pour la définition récursive de la solution optimale :

$$C[i, j] = \begin{cases} 0 & \text{si } S_{i,j} = \emptyset \\ \max_{i < k < j, a_k \in S_{i,j}} \{C[i, k] + 1 + C[k, j]\} & \text{si } S_{i,j} \neq \emptyset \end{cases}$$

Compte tenu du cours sur la programmation dynamique, nous avons réuni tous les conditions pour que l'utilisation de la programmation dynamique soit possible. La complexité de la résolution de ce problème en utilisant une programmation récursive est exponentielle alors qu'avec une programmation ascendante de ce problème donne une solution optimale en temps polynomiale. Nous sommes bien dans un cas de figure ressemblant fortement au calcul du meilleur parenthésage du chapitre traitant de la programmation dynamique.

En revanche, il est peut être en possible de réduire la complexité de ce calcul avec l'utilisation de la programmation gloutonne. Cela nous permettrait de ne pas avoir à tester tous les k valeurs possibles au niveau de tous les sous problèmes envisagés (tous les $[i, j]$).

4.3.4 Conversion d'une solution du type programmation dynamique en une solution gloutonne

Nous avons caractérisé le problème récursivement. Les conditions étant réunies pour utiliser la programmation dynamique pour résoudre ce problème, il existe un algorithme ascendant pour construire la solution optimale en temps polynomiale (programmation *bottom up*).

Si nous souhaitons réduire encore la complexité de ce calcul, sans pré-calculer tous les sous-problèmes, il faut envisager une programmation descendante pour la résolution de ce calcul, avec une seule branche dans cette descente. Dans le cas contraire, le calcul serait exponentiel. Les points de choix existent au niveau de chaque ajout d'activité, et s'il est possible de choisir la bonne activité à coup sûr, alors l'algorithme de construction de la solution optimale devient linéaire. C'est l'existence du choix déterministe qui rend cette construction optimale possible. Il s'agit du choix glouton.

Théorème 4.1

Soit le sous problème non vide $S_{i,j}$ et soit a_m l'activité de $S_{i,j}$ ayant l'heure de fin la plus précoce :

$$f_m = \min \{ f_k \mid a_k \in S_{i,j} \}$$

Alors

1. l'activité a_m est utilisée dans un certain sous ensemble de taille maximale d'activités mutuellement compatibles $S_{i,j}$.
2. le sous problème $S_{i,m}$ est vide, de sorte que choisir a_m fait du sous problème $S_{m,j}$ le seul sous problème susceptible d'être non vide.

Le théorème permet de donner l'existence du choix glouton en limitant le choix de tous les k possible à un seul. Les conditions de la programmation gloutonne, descendante sur une branche unique sont donc réunies (programmation *top down*).

► Le choix glouton est ici de maximiser le temps restant non planifié en privilégiant le choix des activités finissant les plus tôt.

Preuve. Lors de cette preuve, on montre d'abord le point 2, puis le point 1 du théorème précédent :

2. Supposons le contraire, c'est à dire que $S_{i,m}$ est non vide avec a_m l'activité terminant au plus tôt parmi les activités de $S_{i,j}$.

Dans ce cas, $\exists a_k \mid f_i \leq s_k < f_k \leq s_m < f_m$

alors a_k appartient aussi à $S_{i,j}$

or a_k a une heure de fin plus précoce que celle de a_m , ce qui est en contradiction avec le choix de a_m . Ainsi $S_{i,m}$ est bien vide si l'on choisit a_m de cette manière.

1. Soit $A_{i,j}$ un sous ensemble de taille maximale d'activités mutuellement compatibles de $S_{i,j}$. Les activités de $A_{i,j}$ sont triées dans l'ordre monotone croissant des heures de fin des activités de $S_{i,j}$.

Soit a_k la première activité de $A_{i,j}$,

— si $a_k = a_m$ alors a_m appartient bien à un sous ensemble de taille maximale d'activités mutuellement compatibles $S_{i,j}$ ($A_{i,j}$)

— si $a_k \neq a_m$ on définit l'ensemble $A'_{i,j}$ tel que

$$A'_{i,j} = A_{i,j} - \{a_k\} \cup \{a_m\}$$

D'après la forme de $A_{i,j}$, les activités de $A'_{i,j}$ sont disjointes. De plus, l'activité a_m est la première à terminer avec $f_m \leq f_k$ par définition et on peut noter que la taille des ensembles $A_{i,j}$ et $A'_{i,j}$ sont les mêmes ($|A_{i,j}| = |A'_{i,j}|$).

► Nous pouvons donc conclure que $A'_{i,j}$ est un sous-ensemble de taille maximale d'activités mutuellement compatibles de $S_{i,j}$ qui contient a_m . \square

Le théorème permet bien de réduire, comme annoncé précédemment, à la fois le nombre de choix possibles sur les sous problèmes à examiner et ainsi que le nombre de sous problèmes. Dans le cas de la programmation dynamique, il y avait $j - i - 1$ possibilités pour le problème $S_{i,j}$ et fallait envisager tous les alors $S_{i,j}$ possibles. Or, grâce à ce théorème qui rend possible l'utilisation du choix glouton, il y a au plus n sous problèmes, avec un seul choix par sous problème. Ce choix est de prendre l'activité qui termine en

premier et de considérer ensuite toutes les activités ayant une date de début supérieure ou égale à la date de fin de l'activité choisie. Le nombre maximal de sous problème existe lorsqu'aucun recouvrement n'existe entre les activités.

La résolution se fait donc par une résolution descendante. pour résoudre $S_{i,j}$, on choisit a_m , puis on cherche à résoudre $S_{m,j}$ sans que $S_{i,m}$ n'ait besoin d'être résolu avant. On se contente donc de passer en revue toutes les activités une seule fois.

Les algorithmes 19 et 20 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 19 : Algorithme récursif glouton choisissant un nombre optimal d'activités mutuellement compatibles parmi un ensemble d'activités ordonnées

```

1 Les tableaux ci-dessous sont tels que les activités soient triées dans l'ordre
  croissant des dates de fin
2  $s$  : le tableau des dates de début des activités
3  $f$  : le tableau des dates de fin des activités
4  $i$  : numéro de la première activité parmi celles à choisir
5  $n$  : numéro de la dernière activité parmi celles à choisir
6 Fonction activiteChoixGloutonR( $s, f, i, n$ )
7 début
8    $m \leftarrow i + 1$ 
   /* Recherche de la première activité compatible avec  $a_i$  sur  $S_{i,n+1}$ .
      $a_i$  a déjà été choisie */
9   tant que  $(m \leq n) \wedge (s[m] < f[i])$  faire
10  |  $m \leftarrow m + 1$ 
11  si  $m \leq n$  alors
12  |  $retour \leftarrow \{a_m\} \cup activiteChoixGloutonR(s, f, m, n)$ 
13  sinon
14  |  $retour \leftarrow \emptyset$ 

```

Algorithme 20 : Algorithme itératif glouton choisissant un nombre optimal d'activités mutuellement compatibles parmi un ensemble d'activités ordonnées

```
1 Les tableaux ci-dessous sont tels que les activités soient triées dans l'ordre
  monotone croissant des dates de fin
2  $s$  : le tableau des dates de début des activités
3  $f$  : le tableau des dates de fin des activités
4 Fonction activiteChoixGloutonI( $s, f$ )
5 début
6    $n \leftarrow \text{longueur}(s)$ 
7    $A \leftarrow \{a_1\}$  /*  $a_1$  fait forcément partie de la solution étant donnée
   le tri sur les activités */
8    $i \leftarrow 1$ 
   /* Recherche des premières activités compatibles avec les  $a_i$  déjà
   choisies */
9   pour  $m$  de 2 à  $n$  faire
10    si  $f[i] \leq s[m]$  alors
11       $A \leftarrow A \cup \{a_m\}$ 
12       $i \leftarrow m$ 
13  retour  $\leftarrow A$ 
```

4.4 Éléments de stratégie gloutonne

Si 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. Il faut en effet être capable de montrer que le choix glouton est un choix valide pour la construction d'une solution optimale globale.

4.4.1 Étapes de conception pour les activités

Pour établir le cadre de l'utilisation d'un algorithme glouton pour résoudre un problème de manière optimale, il est nécessaire de montrer qu'il conduit inévitablement à une solution optimale. Dans le cadre de l'exemple précédent, nous avons établi les étapes suivantes :

1. déterminer la sous structure optimale du problème ;
2. développer une solution récursive ;
3. faire la preuve que le choix glouton est optimal ;
4. faire la preuve que tous les sous problèmes qui résultant du choix glouton sont vides, sauf un ;
5. construire un algorithme récursif ;
6. construire un algorithme itératif.

Il n'est pas toujours nécessaire de faire toutes ces étapes. Il est néanmoins indispensable de trouver une sous structure avec un choix glouton qui ne laisse plus qu'un sous problème

à résoudre de manière optimale. 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.

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

4.4.2 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 montrer 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.

4.4.3 Sous structure optimale

Propriété 4.2

Une sous structure est optimale si une solution optimale du sous 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.

4.4.4 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.

4.5 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 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.

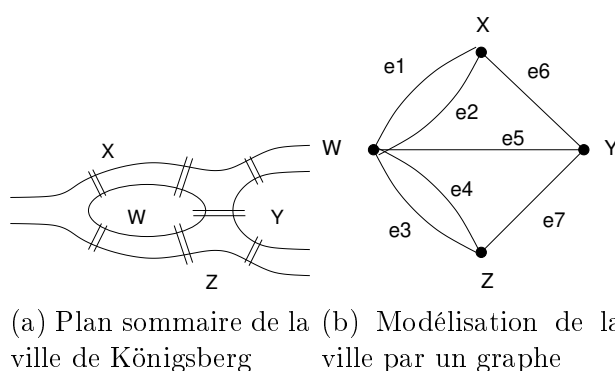


FIGURE 5.1 – Problème du parcours unique de tous les ponts de la ville de Königsberg

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 d'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 par toutes les arêtes 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 à la notion de parcours eulérien et chinois.

5.2.1 Définitions

Le théorème de Euler donne les conditions nécessaires et 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 et tous les sommets sont de degré pair.*

Un graphe qui 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.

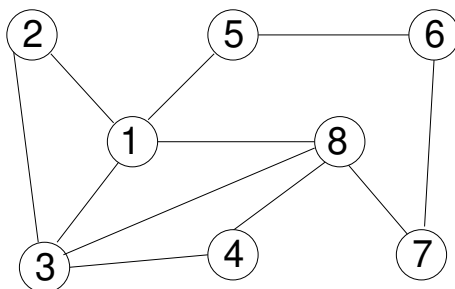


FIGURE 5.2 – Exemple de graphe eulérien

Cette définition 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 et 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 connexe¹ et au plus deux sommets sont de degré impair.*

À noter que ce parcours peut emprunter plusieurs fois le même sommet et que ses sommets de départ et d'arrivée peuvent être distincts. Nous pouvons également parler de chaîne eulérienne. L'exemple classique de graphe qui admet une chaîne eulérienne est celui de "l'enveloppe" (figure 5.3) qu'il est possible de dessiner sans lever le crayon de la feuille à condition de partir du bon endroit.

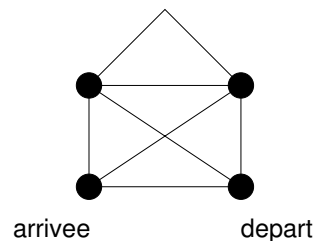


FIGURE 5.3 – Exemple de graphe admettant un parcours eulérien

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. 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 s'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.

Dans le cas où il n'y a pas de poids associé aux arêtes nous minimisons le nombre d'arêtes parcourues.

Il s'agit d'un problème de minimisation ayant de nombreuses 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.

1. Voir la définition en 1.6.

5.2.2 Parcours eulérien d'un graphe non orienté

La résolution de ce problème se fait grâce à un algorithme par construction dont nous décrivons le fonctionnement dans ce paragraphe. Nous faisons ici l'hypothèse que le graphe courant est un graphe eulérien, donc aucun sommet n'est de degré impair.

L'idée est de marquer les arêtes traversées en explorant de proche en proche les arêtes libres. Il ne reste ensuite, qu'à compléter ce parcours depuis des sommets ayant des arêtes incidentes encore libres. Pour ce faire, il existe deux manières de procéder : soit on cherche un parcours partiel que l'on complète ensuite en ajoutant d'autres cycles partiels, soit on exploite toutes les possibilités de parcours de toutes les arêtes éligibles au niveau de chaque sommet parcouru. Dans tous les cas, l'algorithme de calcul d'un cycle eulérien est récursif.

Exemple Soit le graphe donné à la figure 5.4 et stocké en mémoire grâce à des listes d'adjacence triées.

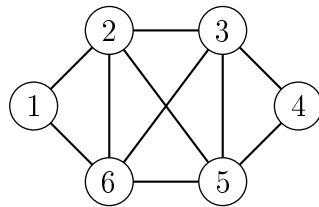


FIGURE 5.4 – Exemple de graphe pour lequel nous cherchons un parcours eulérien

Le parcours en profondeur sur ce graphe peut donner la suite des sommets suivants, en supposant que les sommets sont triés par numéro de sommet :

1 2 3 4 5 6 1

Ce parcours donne un cycle qui ne parcourt 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 une nouvelle exploration, 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 donc le parcours eulérien suivant :

➡ 1 2 5 3 6 2 3 4 5 6 1

Algorithmes Les algorithmes 21 et 22 illustrent les idées données précédemment. Le premier lance un parcours 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, les uns après les autres. Le deuxième algorithme reprend une idée un peu

différente par rapport à la recherche des arêtes oubliées par le parcours en cours. Il s'agit de rechercher un cycle et de relancer ensuite la recherche d'un cycle au niveau de tous les sommets du cycle résultat, et ce récursivement.

Algorithme 21 : 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 euler1( $G, x$ )
2 début
3   ajoutEltList( $L, x$ )
4   tant que  $\exists y \in \Gamma(x)$  tel que l'arête  $(x, y)$  est non marquée faire
5     si le graphe des arêtes non marquées -  $(x, y)$  reste connexe alors
6       marquer( $(x, y)$ )
7       ajoutEltList( $L, euler1(G, y)$ )
8   retourner  $L$ 

```

Algorithme 22 : recherche d'un cycle dans un graphe $G(V, E)$ non orienté avec relance

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

```

1 Fonction euler2( $G, x$ )
2 début
3   ajoutEltList( $L, x$ )
4   si  $x$  est un sommet isolé ( $\Gamma(y) = \emptyset$ ) alors retourner  $L$ 
5   sinon
6      $y \leftarrow x$ 
7     tant que  $\Gamma(y) \neq \emptyset$  faire
8       choisir  $z \in \Gamma(y)$ 
9       supprimer l'arête  $(y, z)$ 
10       $y \leftarrow z$ 
11      ajoutEltList( $L, y$ )
12     /*          soit  $k$  la taille du cycle  $L$  courant          */
13     retourner  $euler2(G, L(1)), euler2(G, L(i)), \dots, euler2(G, L(k))$ 

```

Les algorithmes présentés ici ont une complexité linéaire en fonction du nombre d'arêtes.

5.2.3 Algorithme du postier chinois

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 appairer 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-Complexe.

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 programme².*

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

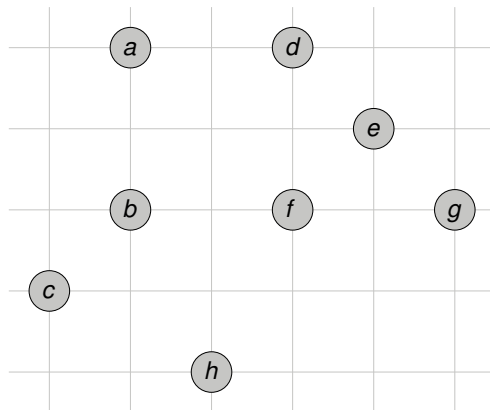
3. élimination des doublons en prenant le prochain sommet dans la liste précédente si celui-ci n'a pas été encore parcouru, etc :

➤ a b c h d e f g

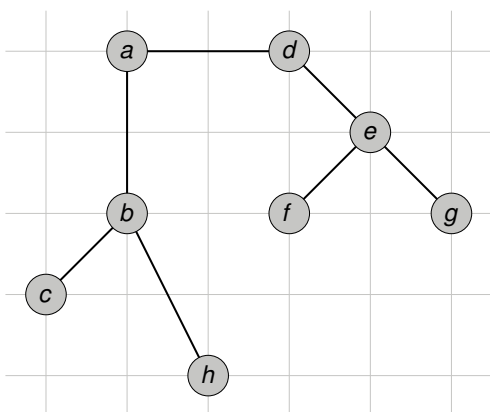
La complexité de l'algorithme permettant de calcul d'une valeur approchée de la tournée optimale d'un voyageur de commerce est $\Theta(|E|) = \Theta(|S|^2)$.

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.

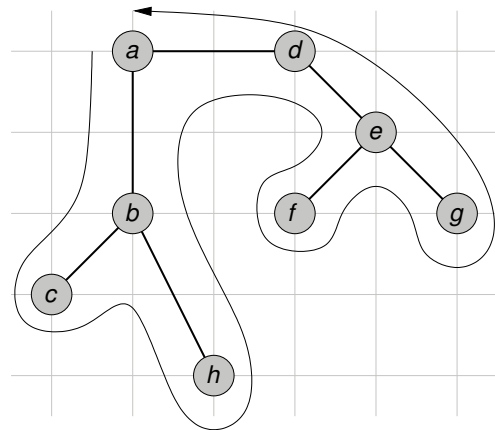
Définir un algorithme d'approximation pour un problème consiste donc à trouver une méthode de calcul puis à prouver que les solutions trouvées sont à un facteur de l'optimal. A noter que cela ne présage pas de la qualité de l'algorithme, le facteur d'approximation pouvant être important. Dans certains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



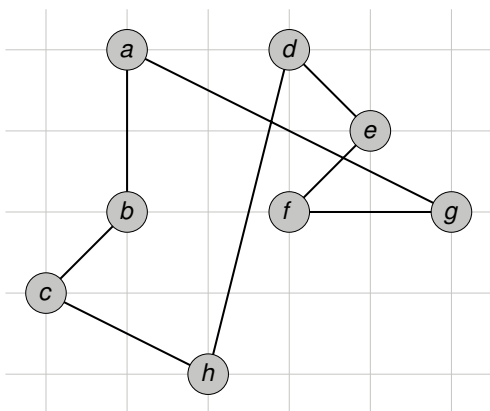
graphe G entrée avec $c(i,j)$ la distance de i à j



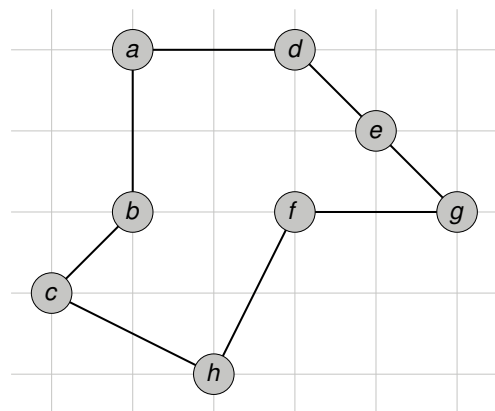
a. arbre de recouvrement (Prim)



b. parcours en profondeur d'abord



c. solution approchée à 19,074



solution optimale H^* avec $c^*(E)=14$,

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 48. 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

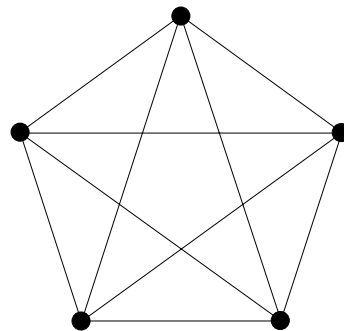


FIGURE 6 – Exemple de graphe complet K_5

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 : $\lfloor \frac{n}{2} \rfloor$

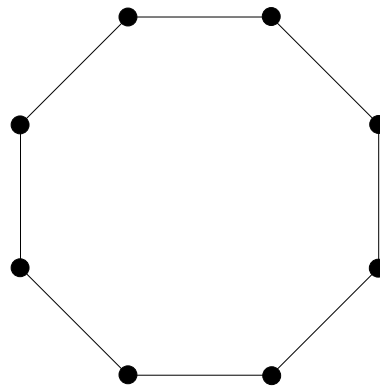


FIGURE 7 – Exemple d'anneau R_8

Comme son 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 $(i - 1) \bmod n$ et $(i + 1) \bmod n$ pour $i = 0, \dots, n - 1$.

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 $x_1x_2 \dots x_i \dots x_n$ est relié avec le sommet $x_1x_2 \dots \bar{x}_i \dots x_n$.

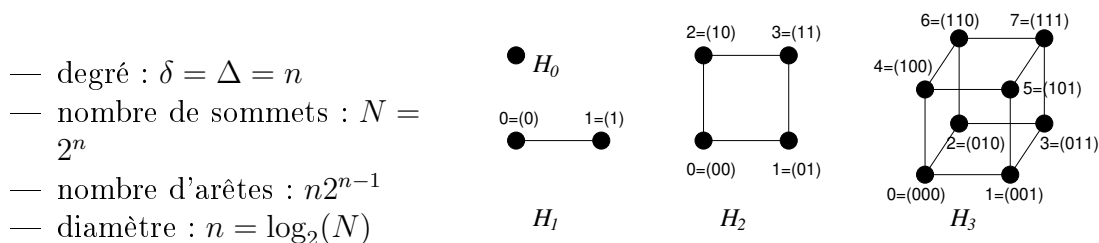


FIGURE 8 – Exemple d'hypercubes H_0, H_1, H_2, H_3

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 : $N = n \cdot 2^n$
- nombre d'arêtes : $3n \cdot 2^{n-1}$
- diamètre : $2n - 1 + \max(1, \lfloor \frac{n-2}{2} \rfloor)$

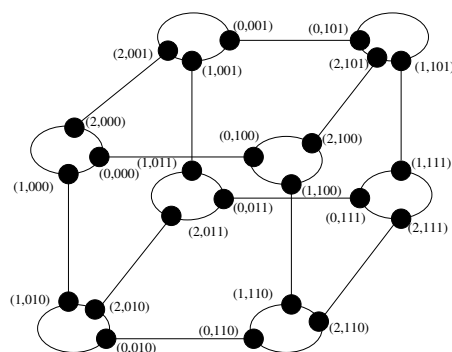


FIGURE 9 – Cube connected cycle CCC_3

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 $M(n_1, n_2, \dots, n_n)$

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 $M(n_1, n_2, \dots, n_k)$, c'est le produit cartésien de k chaînes de n_i sommets avec $i = 1, \dots, k$.

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

$$\sum_{i=1}^k \frac{(n_i-1)N}{n_i} = \sum_{i=1}^k (n_i - 1) \cdot \frac{N}{n_i}$$
- diamètre : $\sum_{i=1}^k (n_i - 1)$

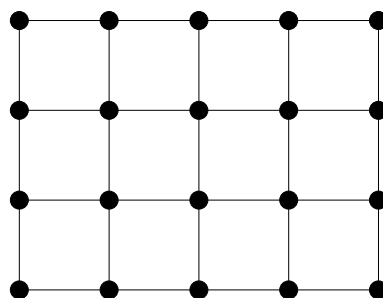


FIGURE 10 – Exemple de grille $M(4, 5)$

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

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

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 $i = 1, \dots, k$.

- 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 : $\sum_{i=1}^k \lfloor \frac{n_i}{2} \rfloor$

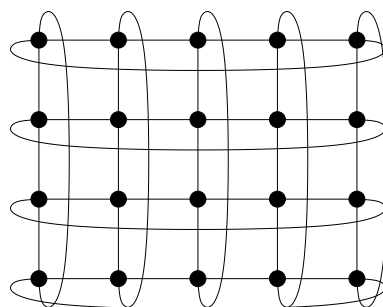


FIGURE 11 – Exemple de grille $TM(4, 5)$

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 $x_1x_2 \dots x_n$ est relié avec les sommets $x_1x_2 \dots \bar{x}_n$ (arête échange), $x_2x_3 \dots x_nx_1$ 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 : $2n - 1$

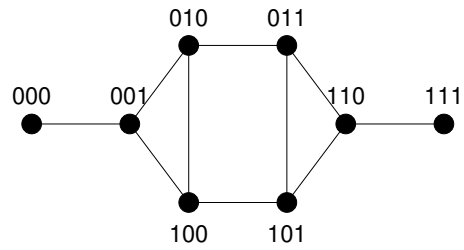


FIGURE 12 – Exemple de Shuffle Exchange SE_3

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 : $n + \lfloor \frac{n}{2} \rfloor$

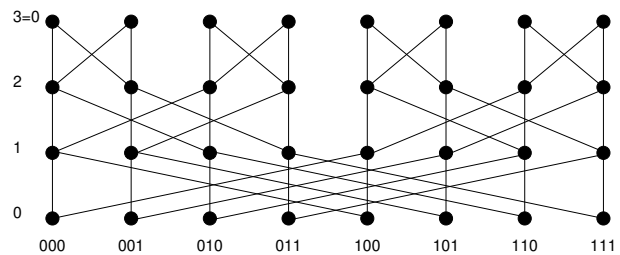


FIGURE 13 – Exemple de Butterfly BF_3

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 : $\lfloor \frac{3(n-1)}{2} \rfloor$

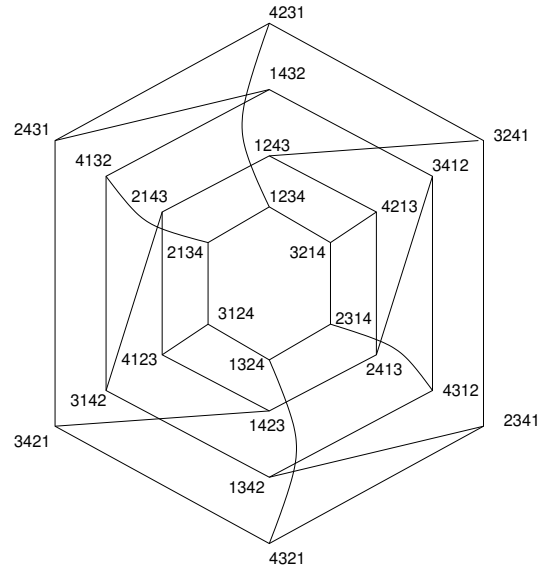


FIGURE 14 – Star Graph SG_4

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

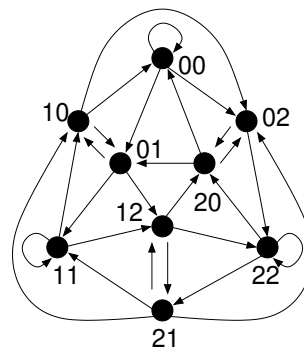


FIGURE 15 – Graphe de De Bruijn $B_{3,2}$