N°d'ordre: .....

République Algerienne Démocratique et Populaire
Ministère de l'enseignement supérieur et de la Recherche scientifique

Université Mouloud Mammeri de Tizi Ouzou
Faculté des Sciences
Département de Mathématiques

Mémoire de Master

Filière : Mathématiques
Spécialité : Recherche Opérationnelle

Par

MANSOURI Melissa

Quelques Applications du parcours en largeurs lexicographic(LexBFS)

Soutenue le 5 octobre 2022 devant le jury :

Mme .	OUBAKOUK Lynda	UMMTO	Président du jury
Mme .	AKLOUCHE Fariza	UMMTO	Examinatrice
Mr.	TALEM Djamel	UMMTO	Encadreur

Année Universitaire : 2021/2022

Dédicace

Je dédie cet humble travail à :

Mes chers parents pour leur témoignés ma gratitude et ma reconnaissance pour leur amour, encouragement et sacrifices.

À mes frères et soeurs, mes amis ainsi que toute personne ayant su être une épaule sut laquelle je pouvais me reposer durant tout mon parcours.
. . .

MANSOURI Melissa.

Dédicace

Je dédie cet humble travail à :

Mes chers parents et ma sœur pour leurs encouragements tout au long de mon cursus universitaire.

Mes amis, mes proches et mes camarades pour leur appuyer et leurs présences.

Mon binôme pour son sacrifice et son dévouement ….

AMIAR Youva

Remerciements

On remercie Dieu le tout-puissant de nous avoir donné la santé et la volonté d'entamer et de terminer ce mémoire.

Il nous est agréable d'adresser nos premiers remerciements à notre promoteur Monsieur TALEM Djamel, qui nous a accompagnés tout au long de notre mémoire, avec ses précieux conseils et orientations.

Mes remerciments vont également à nos enseignants pour leurs générosité et grande patience dont ils ont fait preuve pour nous donner la meilleure formation possible durant nous étude.

Ainsi qu'aux membres du jury OUBAKOUK Lynda et AKLOUCHE Fariza pour avoir accepté d'examiner notre travail.

Enfin, nous remercions toute personne ayant contribué de près ou de loin à la réalisation de ce modeste travail.

Tizi-Ouzou, le 27 septembre 2022.

TABLE DES FIGURES

TABLE DES MATIÈRES

INTRODUCTION

Si on vous dis le mot ***problème***, on suis sur qu'il est familiarisé à tout le monde, car il n'y a pas une seule personne n'ayant pas eu à faire aux problèmes difficiles soient-ils ou facile. La première chose que fait l'individu pour résoudre son problème c'est d'analyser les données dont il dispose et de les traiter et les étudier afin d'essayer de trouver la solution optimale : par exemple dans n'importe quel pays dans le monde , l'inflation est le sujet le plus répondu, beaucoup de pères de famille ou simplement des citoyens essayent de gérer leurs budgets de telle sorte à satisfaire toutes leurs besoins quotidiennes. La première étape est d'évaluer le budget en possession et les produits ou les dépenses à effectuer, la deuxième étape consiste à chercher la meilleure solution possible (chercher des magasins qui proposent des promotions afin d'économiser au mieux ses frais...).

Les mathématiciens ont suivi la même logique pour essayer de résoudre des problèmes dans différents domaines comme les sciences, la gestion, l'économie.... La première étape est nommée la modélisation, son objectif est de réécrire le problème réel sous forme d'un problème mathématique pour pouvoir appliquer la deuxième partie qui est l'optimisation où on cherche à trouver les meilleurs solutions possibles dans le cadre d'une maximisation ou d'une minimisation, en utilisant des outils mathématiques, ce concept est une branche des mathématiques qui sera nommée par la suite la **recherche opérationnelle**.

La **recherche opérationnelle (RO)** est née durant la Seconde Guerre mondiale quand on a demandé aux mathématiciens Patrick Blackett, George Danzig d'optimiser les ressources militaires et cela en trouvant des méthodes très efficaces. C'est de là que vient son nom "opérationnelle" du fait qu'elle

ait été utilisée pour des **opérations** militaires. Mais, si on vient à l'origine, on trouvera qu'il y a eu des mathématiciens qui s'en aient servi bien avant, tel que Alcuin ou même Euler sur le problème des sept ponts de konigsberg. Le problème consistait à déterminer s'il existe ou non une promenade dans les rues de la ville de Königsberg permettant, à partir d'un point de départ, aux villageois de passer une seule fois par chacun des ponts et de revenir à leurs point de départ. Ce genre de problème est toujours présent dans nos jours, par exemple, le ramassage de déchets ou une entreprise qui doit livrer son produit sur plusieurs boutiques dans la ville....
L'un des points forts de la recherche opérationnelle est sa capacité de présenter les problèmes de la vie réelle par des modèles mathématiques, sous forme de programme linéaire, matricielle ou par un graphe comme est le cas dans le problème des ponts de Konigsberg.

La théorie des graphes est un outils de modélisation très puissant, elle permis la représentation d'une manière très simple des interactions entre les éléments d'un grand nombre d'objets de même nature ou de natures différentes. Son développement touche diverses disciplines telles que la chimie, la biologie, l'informatique, l'économie ....

Dans ce mémoire, nous nous intéressons aux applications du parcours en largeur lexicographique LexBFS, à savoir son application, pour la reconnaissance de quelques classes de graphes, la résolution efficace de quelques problèmes et pour l'approximation de rayon.
Notre mémoire est structuré comme suit :

Le chapitre 1 contient les notions de base de la théorie des graphes qui son utile pour la compréhension des prochains chapitres ; le chapitre 2 est constitué par quelques notions de la théorie de la complexité algorithmiques qui seront utilisées pour étudier l'efficacité des algorithmes et la classification des problèmes selon leurs difficultés intrinsèque ; le chapitre 3 est consacré à l'étude des différents types de parcours qui sont à la base des algorithmes de graphes ; enfin, le chapitre 4 est dédié aux applications du parcours LexBFS.

CHAPITRE 1

NOTIONS DE BASE

1.1 Premières définitions

Considérons les 5 pays suivants : l'Algérie, le Maroc, la Tunisie, le Niger et la Libye. Le tableau suivant représente les voisins de chacun de ces payés

pays	Fronitéres
Algérie	Maroc,Tunisie,Niger,Libye
Maroc	Algérie
Tunisie	Libye
Niger	Libye
Libye	Tunisie,Niger

Ces frontières peuvent être représentées par un graphe de la manière suivante :
chaque pays est représenté par un point, appelé sommet (on utilise par exemple la première lettre de son nom), et la frontière commune entre deux payés est représenté par un trait continu appelé arête. par exemple l'Algérie et la Tunisie sont représentés par les sommets A et T, et ils sont reliés par une arrête, car il y a une frontière entre ces deux payes. Ainsi, il est facile de voir que la figure 1.1 est définie par deux ensembles : un ensemble de sommets $V = \{A, M, T, N, L\}$ et un ensemble d'arêtes $E = \{AL; AN; AM; AT; NL; TL\}$ et c'est ainsi qu'on déduit un des modèles

de graphe qui est non orienté (dans la suite du chapitre on verra les graphes orientés).

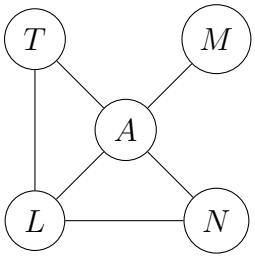

FIGURE 1.1: Graphe des frontières entre pays

Définition 1.1. Un graphe est un couple $G = (V, E)$ formé de deux ensembles, un ensemble de n sommets $V = \{v_1, v_2, ..v_n\}$ et un ensemble de m arêtes $E = \{e_1, e_2, ..e_m\}$.

Une arête e est définie par un couple de deux sommets u et v qui sont ses extrémités et on écrit $e = uv$. Dans ce cas, les sommets u et v sont dits **adjacents** ou **voisins** et l'arête e est dites **incidente** à u et à v. Le nombre de sommets dans G est son **ordre**, le nombre de ses arêtes est sa **taille**. Une **boucle** est une arête dont les extrémités sont confondues. Un graphe est **simple** s'il est sans boucle et entre deux sommets il y a au plus une arête. Le graphe de la figure 1.1 est simple , d'ordre 5 et de taille 6.

Définition 1.2.

1. Si deux sommets u et v sont adjacents dans G, on écrit $u \sim_G v$
2. L'ensemble des voisins d'un sommet v dans un graphe G est

$$N(v) = \{u \in V, u \sim v\}$$

3. Le **degré** d'un sommet v dans G est $d_G(v) = |N(v)|$.
4. Le **degré minimal** de G est $\delta(G) = min\{d(v) : v \in V\}$
5. **Le degré maximal** de G est $\Delta(G) = max\{d(v) : v \in V\}$.

Dans le graphe de la figure 1.1, $N(A) = \{T, M, L, N\}$,$N(M) = \{A\}$, $d(T) = |\{L, A\}| = 2$, $\delta(G) = d(M) = 1$ et $\Delta(G) = d(A) = 4$.

Définition 1.3. Soit $G = (V, E)$ un graphe

1. Le graphe G est **complet** si ses sommets sont deux à deux adjacents.

2. Le graphe G est **biparti** si V peut être partitionné en $\{V_1, V_2\}$ tel que toute arête possède une extrémité dans V_1 et l'autre dans V_2.
3. Le **complémentaire** de G est le graphe $\overline{G}$ ayant les memes sommets et dont les arêtes sont celles qui manquent à G pour qu'il soit complet, c'est-à-dire
$$e \in G \Leftrightarrow e \notin \overline{G}$$

Définition 1.4. Un **digraphe** ou **graphe orienté** est un couple $G = (V, U)$ de deux ensembles, un ensemble de sommets V et un ensemble U dont les éléments sont des **arcs**. Un arc est un couple ordonné de deux sommets, c'est-à-dire, si $a \in U$, alors $a = (u, v)$ où $u, v \in V$. Dans ce cas, u est **l'extrémité initiale** de l'arc a et v est **l'extrémité terminale** de a ; on dit aussi que u est un **prédécesseur** de v et v est un **successeur** de u.

Dans un graphe orienté

1. Le nombre d'arcs sortant d'un sommet x est son **degré extérieur**, il est noté $d^+_G(x) = |\{a \in U, x = I(a)\}|$;
2. Le nombre d'arcs entrant dans x est son **degré intérieur**, il est noté $d^-_G(x) = |\{a \in U, x = T(a)\}|$.

Le degré d'un sommet x est alors la somme de son degré extérieur et son degré intérieur : $d_G(x) = d^+_G(x) + d^-_G(x)$.

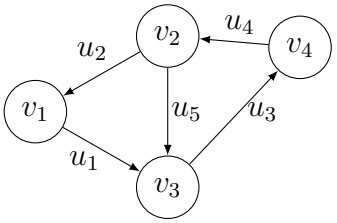

FIGURE 1.2: Graphe orienté

1.2 Sous graphe

Étant donné un graphe $G = (V, E)$, dans de nombreuses applications de la théorie des graphes, on cherche à déterminer si un graphe donné possède un **sous-graphe** (subgraph) avec certaines propriétés voulues. Il y a deux manières naturelles d'obtenir des graphes plus petits à partir de G : par la suppression de sommets ou par la suppression des arêtes.
Plus généralement, un graphe G' est un sous-graphe d'un graphe G si

$V'_G \subseteq V_G$ et $E'_G \subseteq E_G$; G est alors un **sur-graphe** (supergraph) de G'. Ainsi nous disons que G contient G' ou que G' est contenu dans G.

Définition 1.5. Un graphe **partiel** (ou **couvrant**) (spanning graph) d'un graphe $G = (V, E)$ est un sous-graphe $H = (V, E')$ tel que $E' \subset E$, c'est-à-dire H est obtenu à partir de G par suppressions d'arêtes sans toucher aux sommets. Si A est l'ensemble des arêtes supprimées, on écrit $H = G \setminus A = (V, E - A)$ pour désigner le graphe partiel qui en résulte. Par exemple, tout graphe simple est un graphe couvrant d'un graphe complet.

Définition 1.6. Un sous-graphe **induit** (induced subgraph) par un sous-ensemble de sommets S de V_G est le sous-graphe noté G_S dont l'ensemble de sommets est S et l'ensemble d'arêtes est constitué de toutes les arêtes de G qui ont leurs deux extrémités dans A. Le sous-graphe induit G_S est le sous-graphe obtenu par la suppression des sommets de $V_G \setminus S$.

La figure 1.3 montre un graphe G et les sous graphes H et K. Le sous graphe H n'est pas couvrant, il contient pas le sommet i ; il n'est pas induit car les sommets a et c sont adjacents dans G et ne le sont pas dans H. Le sous graphe K est un graphe induit par l'ensemble de sommets $S = \{a, b, c, d\}$

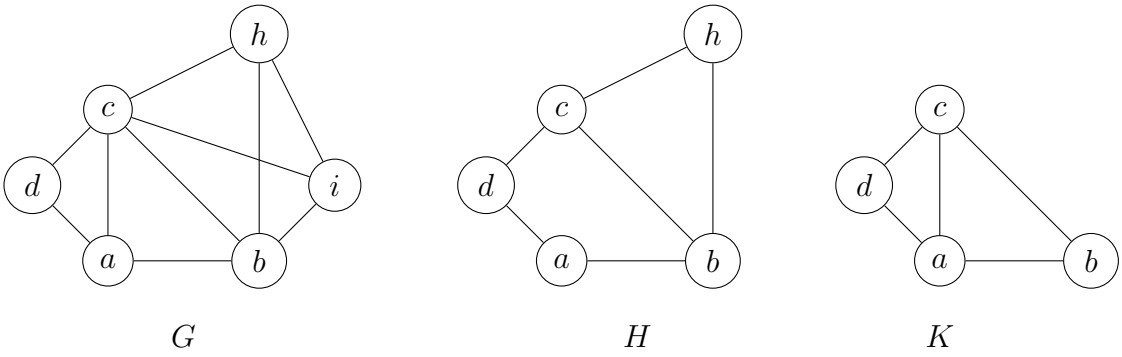

FIGURE 1.3: Le graphe G et les sous graphe H, K.

Définition 1.7. Un **stable** dans G est un ensemble de sommets deux à deux non adjacents, c'est-à-dire un sous-graphe sans arêtes. Une **clique** dans G est un ensemble de sommets deux à deux adjacents, c'est-à-dire un sous-graphe

complet. Une **biclique** dans G est un sous graphe biparti complet. Remarquons qu'une clique dans un graphe est un stable dans son complémentaire et inversement.

1.3 Connexité et forte connexité

Définition 1.8. Une **chaine** dans un graphe $G = (V, E)$ est une sequence d'arêtes $\mu = (x_1, x_2, \ldots, x_k)$ telle que, pour $i = 1 \ldots, k-1$, $x_i x_{i+1}$ est une arête, ça longueur est $l(\mu) = k - 1$. Un **cycle** est une chemin dont les extrémités sont confondues.
Un **chemin** dans un graphe orienté $G = (V, U)$ est une sequence d'arcs $\mu = (x_1, x_2, \ldots, x_k)$ telle que, pour $i = 1 \ldots, k-1$, (x_i, x_{i+1}) est un arc, ça longueur est $l(\mu) = k-1$. Un **circuit** est un chemin dont les extrémités sont confondues.

Une chaine (resp. chemin) est **élémentaire**, si elle (il) passe au plus une fois par chaque sommet, elle (il) est **simple**, si elle (il) passe au plus une fois par chaque arête (resp. arc). Un cycle (resp. circuit) est **élémentaire**, s'il passe au plus une fois par chaque sommet, il est **simple**, s'il passe au plus une fois par chaque arête (resp. arc).

Définition 1.9.

1. Une **Corde** dans un cycle μ est une arête reliant deux sommets non consécutifs dans μ.
2. Un **Trou** est un cycle élémentaire de longueur supérieure ou égale à 4, noté $C_n, n \geq 4$.

Définition 1.10. (Graphe connexe)
Un graphe G est **connexe** si entre deux sommets quelconques, il existe une chaine allant de l'un vers l'autre. Un graphe **non connexe** est constitué par plusieurs **composantes connexes**. Une composante connexe dans un graphe non connexe est un sous graphe induit connexe maximal, c'est à dire n'est pas inclus dans un autre sous graphe induit connexe.

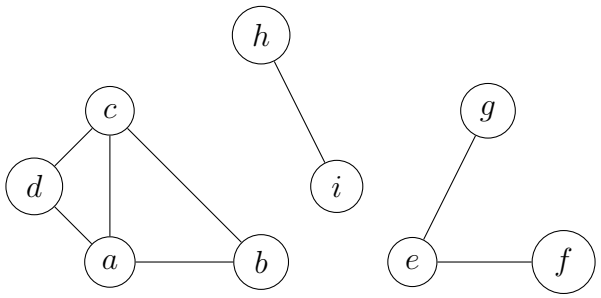

FIGURE 1.4: Graphe non connexe avec trois composantes connexes : $C_1 = \{a, b, c, d\}, C_2 = \{e, f, g\}$ et $C_3 = \{i, h\}$.

Définition 1.11. (Forte connexité)
Un graphe orienté $G = (V, U)$ est **fortement connexe** si pour tout couple de deux sommets x et y, il existe un chemin allant de x à y et un autre chemin allant de y vers x, c'est à dire il existe un circuit passant par les deux sommets. Un graphe non fortement connexe, est constitué par des **composantes fortement connexes**. Une composante fortement connexe (cfc) est un sous-graphe induit fortement connexe maximal, c'est à dire n'est pas inclus dans un sous graphe induit fortement connexe.

1.4 Représentation Matricielle des graphes

Soit G un graphe (orienté ou non) d'ordre n et de taille m. La représentation des graphes en machine (ordinateur) nécessite d'associer à G différents types de matrices, matrice d'adjacence, matrice d'incidence et liste d'adjacence

1.4.1 Matrice d'adjacence

La matrice d'**adjacence** (adjacency matrix) d'un graphe d'ordre n est la matrice carrée A d'ordre $n \times n$ (une ligne pour chaque sommet et une colonne pour chaque sommet) telle que $a_{ij} = 1$ s'il existe une arête (resp. un arc) entre les sommets i et j, et $a_{ij} = 0$ sinon.

$$a_{ij} = \begin{cases} 1 & Si \quad ij \in E(\text{ resp. } (i,j)) \in E) \\ 0 & Sinon \end{cases}$$

Exemple 1.1.
Soient le graphe non orienté G et le graphe H obtenu par l'orientation de G (voir 1.5). Ces deux graphes ont la même matrice d'adjacence. La matrice d'adjacence des deux graphes est la suivante :

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

Remarque 1.1. Notons que la matrice d'adjacence d'un graphe simple est symétrique, ça diagonale est nulle. De plus, une matrice d'adjacence dépend

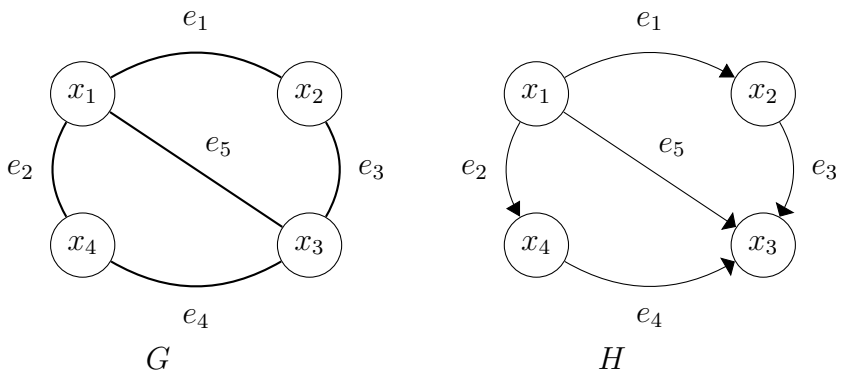

FIGURE 1.5: G graphe non orienté,H graphe orienté

de la numérotation des sommets du graphe qu'elle représente, donc si nous changeons la numérotation des sommets, nous trouverons une autre matrice d'adjacence.

1.4.2 Matrice d'incidence

La **matrice d'incidence** (incidence matrix) d'un graphe simple est la matrice $B = (n \times m)$, une matrice ayant n ligne et m colonne, c'est à dire une ligne pour chaque sommet et une colonne pour chaque arête telle que :

1. pour un graphe orienté, $b_{ij} = 1$ si le sommet i est l'extrémité initiale de l'arc j, $b_{ij} = -1$ si i et l'extrémité terminale de l'arc j et $b_{ij} = 0$ sinon.
2. pour un graphe non orienté, $b_{ij} = 1$ si le sommet i est l'extrémité de l'arête j, $b_{ij} = 0$ sinon.

Pour un graphe orienté, la matrice d'incidence est définie :

$$m_{ij} = \begin{cases} 1 & Si \quad x_i \text{ est l'extrémité initiale de } \mathrm{e}_j; \\ -1 & Si \quad x_i \text{ est l'extrémité terminale de } \mathrm{e}_j; \\ 0 & Si \quad x_i \text{ n'est pas une extrémité de } \mathrm{e}_j. \end{cases}$$

Pour un graphe non orienté, la matrice d'incidence est définie par :

$$m_{ij} = \begin{cases} 1 & Si \quad x_i \text{ est une extrémité de } \mathrm{e}_j; \\ 0 & Sinon. \end{cases}$$

Exemple 1.2. La matrice d'incidence associée au graphe orienté H de la figure 1.5 est la suivante :

$$\begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{array} \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & e_3 & e_4 & e_5 \end{array} \\ \begin{pmatrix} +1 & +1 & 0 & 0 & +1 \\ -1 & 0 & +1 & 0 & 0 \\ 0 & 0 & -1 & -1 & -1 \\ 0 & -1 & 0 & +1 & 0 \end{pmatrix} \end{array}$$

La matrice d'incidence associée au graphe non orienté G de la figure 1.5 est la suivante :

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

CHAPITRE 2

NOTIONS SUR LA COMPLEXITÉ ALGORITHMIQUE

La théorie de la complexité est une branche des mathématiques et d'informatique utilisée pour étudier l'efficacité des algorithmes et classer les problèmes algorithmiques (problème résolu par un algorithme) selon la complexité des meilleurs algorithmes qui les résout.

2.1 Problème de décision et problème d'optimisation

Un **problème** est une question générale, c'est-à-dire qui s'applique à un ensemble d'éléments appelés **instances** ou **données** du problème.

Exemple 2.1.

1. Déterminer si un entier naturel est premier est appelé
   PROBLÈME DE PRIMALITÉ
   Donnée : Un entier N ;
   Question : N est-il premier ?
2. Calcule R une clique avec un maximum de sommets dans un graphe est appelé
   PROBLÈME DE CLIQUE MAXIMUM
   Donnée : Un graphe $G = (V, E)$;
   Question : Trouver une clique maximum dans G ?

Dans le premier problème les instances sont des entiers naturels et dans le second les instances sont les graphes.

D'après cet exemple, nous distinguons au moins deux types de problèmes, problème dont la réponse est soit oui soit non, et problème qui consiste à maximiser ou minimiser un certain paramètre.

Définition 2.1.

1. Un **problème de décision** consiste à répondre par oui ou par non à une question. Un problème de décision peut etre vu comme une application sur X à valeur dans $\{0, 1\}$ telle que : $f(x) = 1$ si x vérifie une certaine propriété P et $f(x) = 0$ sinon.[2]
2. Un **problème d'optimisation** consiste à trouver la solution $x^* \in X$ qui optimise un certain paramètre dont la valeur est calculée par une fonction f sur l'ensemble X,

   $$f(x^*) = min_{x \in X} f(x) \text{ ou } f(x^*) = max_{x \in X} f(x)$$

   où Xest un ensemble des solutions réalisables, x^* est une solution optimale et f est une fonction objective.

Remarque 2.1. Notons qu'à tout problème de décision correspond un problème d'optimisation. par exemple la version optimisation du problème de la clique est
Donnée : Un graphe G et un entier k.
Question : Existe-t-il dans G une clique avec au moins k sommets ?

2.2 Complexité des algorithmes

La théorie de la complexité comme on l'a déjà dit s'intéresse aux problèmes résolubles par des algorithmes.

Définition 2.2. Un **algorithme** est une suite d'opérations élémentaires dites instructions, qui a pour but de trouver un résultat à partir des données connues du problème.

Chaque algorithme est caractérisé par deux fonctions de complexité

Définition 2.3. La *fonction de complexité spatiale* donne le nombre de cases mémoires occupées par les données manipulées par l'algorithme au cours de son exécution et la *fonction de complexité temporelle* qui correspond au nombre d'opérations élémentaires effectuées au cours de son exécution.

Dans la suite, nous nous intéressons qu'à la complexité temporelle qui s'exprime en fonction de la taille des instances.

Définition 2.4. On appelle taille d'une instance I l'entier noté $|I|$ qui mesure le nombre de cases mémoires nécessaires pour la représentation de I par l'ordinateur. La taille d'un graphe s'exprime par le nombre n de ses sommets et le nombre m des ses arêtes.

Notons que le temps d'exécution d'un algorithme est proportionnel avec la taille des instances, il peut aussi varier sur des instances de même taille. Par exemple, pour chercher l'existence d'un élément dans un tableau contenant n entiers, on peut le trouver dans la première position donc la valeur de la complexité est de 1 ; mais, on peut le trouver à la dernière position et donc la valeur de la complexité est n. Ainsi pour que notre calcul de complexité soit fiable, on définit généralement la complexité en considérant la pire instance possible parmi toutes les instances de taille n.

Définition 2.5. Soient P un problème et A un algorithme qui résout P. Notons $\mathcal{I}_{(P,n)} = \{I/I \text{ est une instance de } P \text{ et } |I| = n\}$ et φ_A est une application qui a toute instance I fait associer le temps d'exécution de A sur I. La fonction de complexité de l'algorithme A est une application c_A définie sur l'ensemble des entiers naturels par :

$$c_A(n) = max_{\mathcal{I}_{(P,n)}} \varphi(n).$$

$c_A(n)$ est le nombre maximum d'opérations élémentaires prises par A sur les instances de taille n.

La complexité d'un algorithme est donnée par la nature de la fonction de sa complexité temporelle, et elle est donnée par notation grand "O" dont la définition est

Définition 2.6. Soient f et g deux fonctions $f; g : \mathbb{N} \to \mathbb{R}_+$.
On dit que $f = O(g)$(et on lit f est un grand O de g) lorsqu'il existe un entier n_0 et une constante réelle c tel que pour tout $n \geq n_0, f(n) \leq cg(n)$. Ceci signifie que g devient plus grande que f à partir d'un certain entier n_0.

Exemple 2.2. Soit $f(n) = 6n^4 - 2n^3 + 5$. Pour $n_0 = 1$ et pour tout $n \geq n_0$,on a

$$6n^4 - 2n^3 + 5 \leq 6n^4 - 2n^4 + 5n^4 = 13n^4$$

Donc, en prenant $c = 13$, on a $f = O(n^4)$. Autrement dit, à un facteur constant prés, $f(n)$ ne croît pas plus rapidement que n^4. Il est facile de voir qu'un polynôme $p(n)$ de degré k est toujours un $O(n^k)$.

Définition 2.7. On dit qu'un algorithme A est **efficace** si sa fonction de complexité est majorée par un polynôme, c'est-à-dire il existe k tel que $c_A = O(n^k)$.

Types de complexité

- $O(1)$: complexité constante.
- $O(\log n)$: complexité logarithmique.
- $O(n)$: complexité linéaire.
- $O(n \log n)$: complexité quasi linéaire.
- $O(n^2)$: complexité quadratique.
- $O(2^n)$: complexité exponentielle.
- $O(n!)$: complexité factorielle.

2.3 Complexité des problèmes

Dans la suite de ce mémoire, sauf indication contraire, tous les problèmes considérés sont de décision. Ces problèmes seront classés dans des classes selon leurs difficultés intrinsèques. La complexité d'un problème est donnée par la complexité du meilleur algorithme le résolvant.

2.3.1 Les classe P et NP

Définition 2.8. La complexité d'un problème est la complexité du meilleur algorithme qui permet de le résoudre .Si cet algorithme est polynomial le problème est dit facile.

Définition 2.9. La classe $\mathcal{P}$ est l'ensemble de tous les problèmes de décision qu'on peut résoudre par un algorithme de complexité polynomiale. La classe $\mathcal{NP}$ est l'ensemble de tous les problèmes pour lesquels toute solution proposée est vérifiable par un algorithme polynomiale.

Remarque 2.2. Il est facile de voir que $\mathcal{P} \subseteq \mathcal{NP}$. En effet, si une solution est calculable par un algorithme polynomiale alors elle est vérifiable en temps polynomiale. C'est la vérification de l'inclusion inverse qui pose problème, jusqu'à présent, personne n'a pu confirmer ou infirmer sa vérification, elle constitue l'un des problèmes ouverts les plus importants de notre époque.

2.3.2 La classe NP-complet

La communauté scientifique s'accordent sur l'hypothèse que $\mathcal{P} \neq \mathcal{NP}$. Sous cette hypothèse il existe dans $\mathcal{NP}$ une autre classe contenant les problèmes les plus difficiles. La définition suivante est utile pour définir cette nouvelle classe.

Définition 2.10. Soient p, q deux problèmes de décision et I_p (resp. I_q) leurs ensembles des instances. On dit que p se transforme (ou se réduit) polynômialement en q s'il existe une application $f : I_p \to I_q$, calculable en un temps polynomial et transformant toute instance x de p en une instance $f(x)$ de q admettant la même réponse que x.
On écrit alors $p \leq q$, et qui signifie que le problème p n'est pas plus difficile que le problème q.

Exemple 2.3. Considérons le problème du couplage (matching problem) dans un graphe. Un **couplage** est un ensemble d'arêtes M qui sont deux à deux disjointes. La version "décision" de ce problème est

---

Instance : un graphe $G = (V, E)$ et un entier positif k.
Question : Existe-t-il un couplage M de cardinalité $|M| \geq k$?

Ce problème se réduit en un temps polynomial au problème de la clique comme suit : soit $H = \big(V(H), E(H)\big)$ le graphe dont les sommets sont les arêtes de G, et deux sommets de H sont adjacents si et seulement si les arêtes qui leurs correspondent dans G sont disjointes. Il est facile de se convaincre que le graphe H peut être calculé en un temps polynomial et que le graphe G admet un couplage de taille k si et seulement si le graphe H admet une clique de taille k.

Remarque 2.3. Sachant que toute clique dans un graphe G est un stable dans son complémentaire G^c, et tout stable dans G est une clique dans G^c. Ceci montre que le problème de la clique peut se réduire en un temps polynomial au problème du stable, et vice-versa. Ainsi, ces deux problèmes ont une même difficulté. En général, deux problèmes p, q sont dits équivalents (en termes de la difficulté) si $p \leq q$ et $q \leq p$. On écrit alors $p \sim q$.

Définition 2.11. Un problème de décision q est dit $NP - complet$ si :

1. $q \in NP$.
2. $\forall p \in NP, p \leq q$.

Proposition 2.1. *si p se réduit à q et s'il existe un algorithme efficace pour q, alors il existe un algorithme efficace pour p.*

En 1971, Cook a montré que la classe $\mathcal{NP}$-complet n'est pas vide, il a montré qu'elle contient au moins le problème SAT dont la définition est la suivante.

Soit $X = \{x_1, ..., x_n\}$ un ensemble de variables booléennes (où x_i vaut vrai ou faux) et $\bar{X} = \{\bar{x}/x \in X\}$ (où $\bar{x}$ est la négation de x).

- Une assignation pour X est une application $T : X \cup \bar{X} \to \{vrai, faux\}$ vérifiant : $T(x) = vrai$ si et seulement si $T(\bar{x}) = faux$.
- On appelle les éléments de $X \cup \bar{X}$ des littéraux sur X.
- Une clause sur X est une disjonction de littéraux, exemple $(x_1 \vee \bar{x_2} \vee x_3\ )$. Cette clause est dite satisfaite par une assignation si et seulement si au moins un de ses littéraux est vrai.
- une famille M de clauses sur X est satisfaisable si et seulement s'il existe une assignation qui satisfait en même temps toutes ses clauses.

Définition 2.12.

le problème de satisfaisabilité (SAT) :

- **Donnée :** Un ensemble X de variables et une famille M de clauses.
- **Question :** Existe-t-il une fonction d'assignation T permettant de satisfaire en même temps toutes les clauses de M ?

Théorème 2.2. *Le problème de la satisfaisabilité est $NP-complet$[11].*

Remarque 2.4. Après la démonstration de ce théorème beaucoup d'autres problèmes ont pu être énoncé comme $NP-complet$ et ça ont utilisant SAT pour montre leur $NP-completude$.

CHAPITRE 3

PARCOURS DE GRAPHES

L'exploration de graphe est une démarche très importante pour mettre en évidence les propriétés de celui ci ou son appartenance à une classe de graphe particulière, montrer qu'un graphe est connexe, sans cycle ou planaire.... Pour cela il existe beaucoup d'algorithmes de parcours qui permettent de visiter les sommets et arêtes du graphe accessible à partir d'un sommet initial. Parmi ces parcours, nous distinguons le parcours en largeur BFS, le parcours en profondeur DFS et parcours en largeur lexicographique LesBFS.

Définition 3.1. Un **parcours** du graphe $G = (V, E)$ est un algorithme qui consiste à explorer les sommets de G un par un à partir d'un sommet initial, qu'on appelle racine ou source noté s.

Au cours du parcours , un sommet peut être dans l'une des deux situations :

1. Un sommet est **non exploré** s'il n'est pas encore visité.
2. Un sommet est **exploré** s'il est déjà visité.

Un sommet i exploré est dit :

1. **fermé :** si tous ses voisins ont été explorés.
2. **ouvert :** si au moins un de ses voisins n'a pas encore été exploré.

Définition 3.2.

1. Une **File** est une structure de données basée sur le principe **premier entré, premier servi** "FIFO" (First in, First out), ce qui signifie que les Premiers éléments à ajouter à la file seront les premiers à sortir.

2. Une **Pile** est une structure de données basées sur le principe **dernier entré premier sorti** "LIFO" (last in, first out)ce qui signifie que le dernier élément ajouté à la pile sera le premier à être sortie.

3.1 Parcour en largeur

L'algorithme de parcours en **largeur**"BFS" (Breath First Search) permet de parcourir un graphe à partir d'un sommet initial s, puis on explore ses successeurs, après les successeurs des successeurs jusqu'à avoir exploré tous les sommets du graphe G. La structure de données utilisée est la file. Les étapes du parcours sont résumées comme suit :

1. Mettre le sommet initial dans la file ;
2. Retirer le sommet de la tête de la file pour l'examiner (i.e exploré ses voisins non explorés) ;
3. Mettre tous ses non explorés dans la queue de la file ;
4. Si la fille n'est pas vide, aller en 2.

Algorithme 3.1. ***Parcours en largeur*** BFS
Données : *Un graphe* $G = (V, E)$ *et un sommet de départ* s.
Sortie : *Une arborescence des plus couts chemins par rapport à* s.

1. *Pour tout* $x \neq s$ *faire*
2. $couleur[x] = jaune, parent[x] = nil,\ d(x, s) = \infty$;
3. $couleur[s] = vert; parent[s] = nil; d(s, s) = 0$;
4. $F = \phi$ //*File vide*
5. *ajouter* (F, s)// *mettre* s *dans la file*
6. *tant que* $F \neq \phi$ *faire*
7. $x =$ *extraire tête* (F)//
8. *pour chaque* $y \in N(x)$ *faire*// $N =$ *l'ensemble des voisin*
9. *si* $couleur[y] = jaune$ *alors*
10. $couleur[y] = vert$
11. $d(s, y) = d(s, x) + 1$
12. $parent[y] = x$
13. *ajouter* (F, y)
14. $couleur[x] = rouge$

complexité 1. Soit un graphe G connexe. Au cours de l'exploration de ce graphe chaque sommet est examiné au plus deux fois, une fois pour le mettre dans la file et la seconde fois pour le retirer. Chacune de ces opérations peuvent être exécutées en $O(1)$. Une arête est utilisée au plus une fois pour aller d'un sommet père vers son fils, mais jamais le contraire. Puisque le nombre de sommets et d'aretes sont respectivement n, m, la complexité du parcours BFS est en $O(m+n)$.

Exemple 3.1. Dans cet exemple on va voir le déroulement de l'algorithme du parcours en largeur

Le sommet est dans la file (sommet exploré)

Le sommet n'est pas dans la file (sommet non exploré)

Le sommet est sorti de la file (tous ses voisins sont explorés)

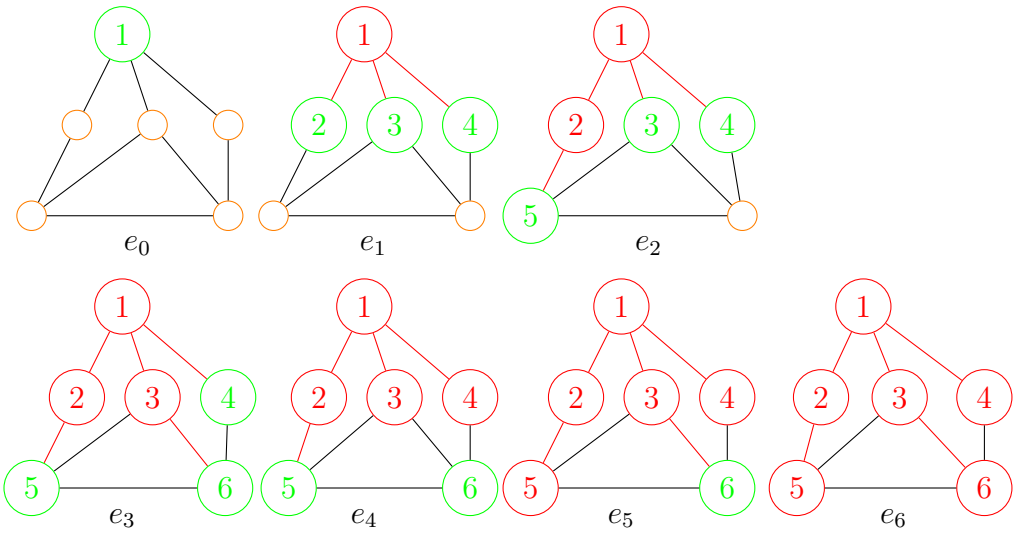

FIGURE 3.1: Illustration d'un parcours en largeur.

3.2 Parcours en profondeur DFS

Le parcours DFS utilise la stratégie suivante : en arrivant à un sommet u il traite l'un de ses voisins qui n'est pas encore visité, soit le sommet v. Avant de traiter le voisin suivant de u, il traite d'abord tous les descendants du

sommet v. Le même traitement sera appliqué à tous les sommets accessibles depuis la source. Pour cela, on utilise la structure de données **pile** (stack). Les étapes du parcours sont résumées ainsi :

1. On choisit le sommet initial s.
2. On choisit un de ses voisins y.
3. Si le sommet y n'est pas fermé, on relance le parcours à partir de ce sommet.

L'exploration s'arrête quand tous les sommets sont fermés. Nous utilisons les couleurs suivantes : un sommet est blanc s'il n'est pas découvert, en gris s'il est ouvert et en vert s'il est fermé.

Exemple 3.2. Dans cet exemple, nous montrons le déroulement de l'algorithme de parcours en profondeur.

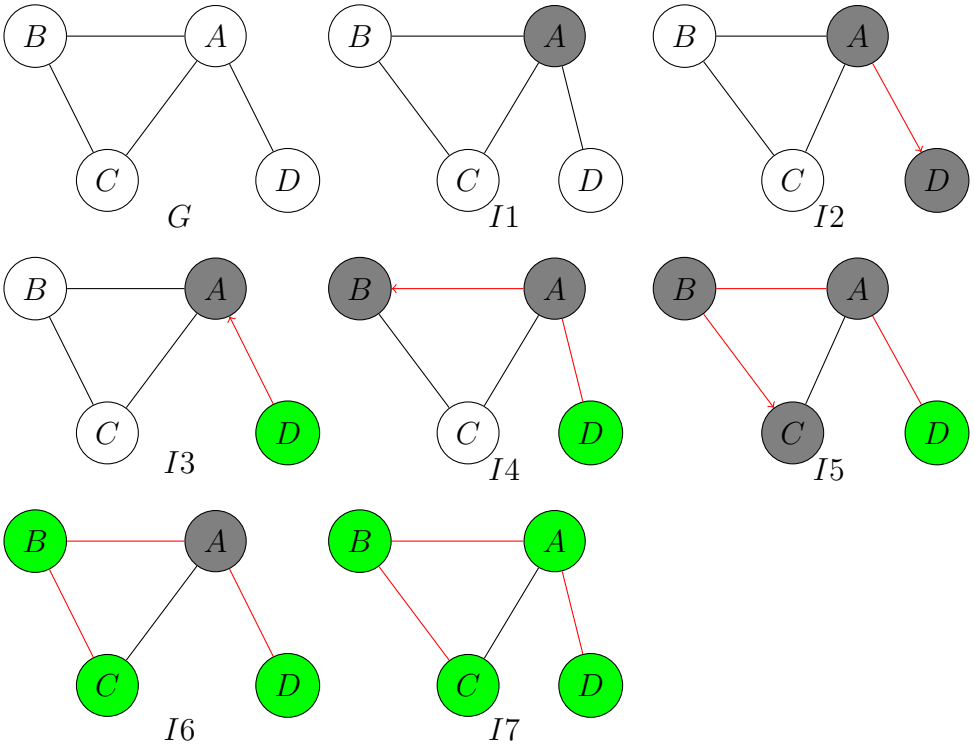

FIGURE 3.2: Illustration d'un parcours en profondeur

Complexité. La complexité du parcours en profondeur est en $O(n+m)$.

3.3 Parcours en largeur lexicographique (LexBFS)

Pour la reconnaissance de certains graphes, les mathématiciens on définit l'algorithme de LexBFS (Lexicographic Breath First search) qui est basé sur la construction d'un ordre sur les sommets d'un graphe $G = (V, E)$ en utilisant la notion d'ordre lexicographique. Parmi les classes de graphe que l'on reconnaît efficacement grâce à lexBFS, on trouve les graphes triangulés, les graphes d'intervalles ....

Définition 3.3. Soient $L_1 = (a_1, a_2, ...a_k)$, $L_2 = (b_1, b_2, ...b_s)$ deux vecteurs dont les coordonnées sont des entiers naturels. On dit que L_1 est inférieur lexicographiquement à L_2 et l'on écrit $L_1 <_{lex} L_2$ si et seulement si

1. il existe $j \leq min(k, t)$ tel que $a_i = b_i$ pour $i < j$ et $a_j < b_j$ ou

2. $k \leq l$ et $a_i = b_i$ pour $i = 1...k$.
   Pour tout vecteur $L = (a_1, a_2, ...a_k)$, si a est un entier alors $L + a = (a_1, a_2, ...a_k, a)$.

3.3.1 Principe de LexBFS

Soit un graphe $G = (V, E)$ et $|V| = n$.

1. On va associer à chaque sommet de G une étiquette (un vecteur) $L(v)$ qui est vide au départ. À l'instant ou cette étiquette n'est plus vide (ou court de l'exécution du parcours) les coordonnées de l'étiquette seront ordonnées par un ordre décroissant, si $L = (a_1, a_2, ...a_k)$, alors $(a_1 \leq a_2 \leq .... \leq a_k)$.
2. On énumère les sommets de G de n à 1, à partir d'un sommet quelconque $s = \sigma(n) = v_n$ (s est le sommet initial, il va prendre le numéro n).
3. On ajoute n à toutes les étiquettes des sommets qui appartiennent à $N(s)$.
4. Après, énumérer le sommet qui a $L(v)$ maximale lexicographiquement.
5. On fait ces étapes jusqu'à avoir énuméré tous les sommets de G.

Algorithme 3.2. *(parcours en largeur lexicographique LexBFS)*
Données : *un graphe $G = (V, E)$ et un sommet initial s.*
Résultat : *un ordre total σ de V.*

1. *$L(s) = \{n\}$; $L(v) = \emptyset$ pour $v \neq s$*

2. *pour i de n à 1 faire*
3. *choisir un sommet v d'étiquette lexicographique maximale.*
4. $\sigma(i) = v$
5. *pour chaque sommet non numéroté w de $N(v)$ faire*
6. $L(w) = L(w) + i$.

Remarque 3.1. Au cours de l'énumération des sommets, si deux sommets non numérotés ont une même étiquette maximale, le choix de l'un ou l'autre n'importe pas.

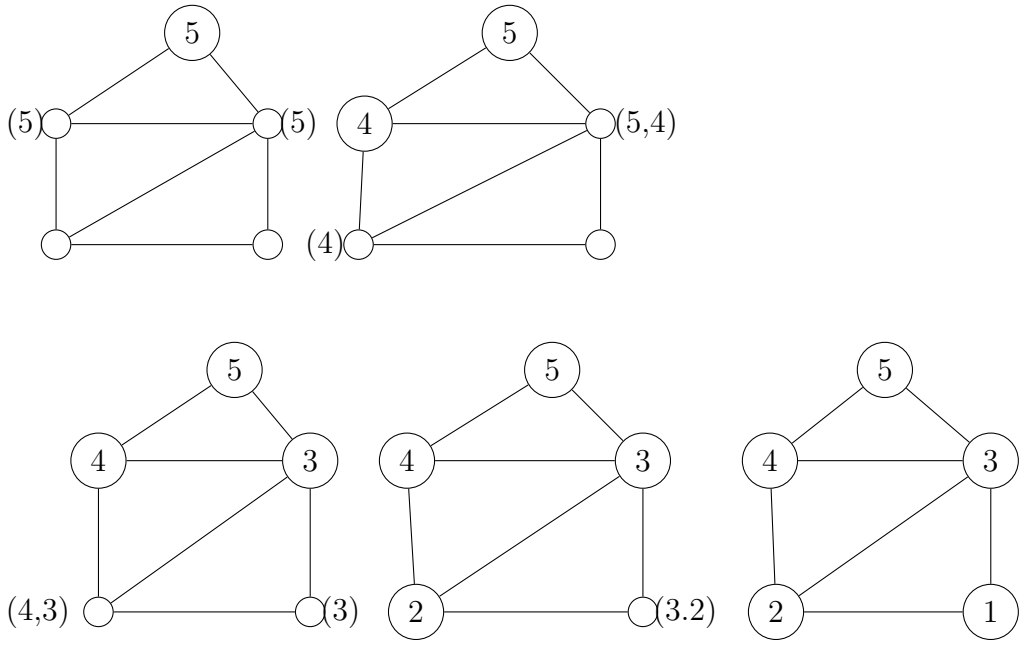

FIGURE 3.3: Déroulement du LexBFS

Exemple 3.3.

Définition 3.4. Soient u et v deux sommets dans un graphe ordonné par LexBFS σ, la notation $u \longrightarrow v$ signifie que $uv \in E$ et $u <_\sigma v$; dans ce cas, on dit que v est un voisin supérieur immédiat de u. L'ensemble des voisins supérieurs immédiats de u et $N^+(u) = \{v \in V, u \longrightarrow v\}$.

Lemme 3.1. *[15] Soit v un sommet dans un graphe biparti $G = (V, E)$ muni d'un ordre lexBFS $\sigma = (v_1, v_2, \ldots, v_n)$. Alors :*

$$L(v) = \{j / v \longrightarrow v_j\}.$$

CHAPITRE 4

APPLICATIONS DU PARCOURS LEXBFS

Le parcours LexBFS peut être utilisé pour la reconnaissance des graphes triangulés [15], la reconnaissance des graphes d'intervalles et l'orientation transitive [12]. Il peut aussi être utilisé pour le calcul de diamètre et aussi pour résoudre efficacement les problèmes de coloration, de la clique et du stable dans la classe des triangulés....

4.1 Graphe triangulé

Définition 4.1. Un graphe **triangulé ou cordal** (chordal graph) est un graphe sans trou.

Sans doute, la classe des graphes triangulés est l'une des classes les plus importantes en théorie des graphes et en algorithmique, elle a été introduite en 1958 par Hajnal et al. La reconnaissance de cette classe peut se faire en un temps linéaire, soit en utilisant l'algorithme LexBFS [15] ou en utilisant l'algorithme (maximum cardinality search MCS) [14].

Plusieurs problèmes qui sont NP-complet, comme les problèmes du stable, de la clique et de coloration ..., deviennent polynomiaux quand ils sont considérés dans cette classe de graphes [14]. Il est claire que tout sous graphe induit d'un graphe sans trou est sans trou, d'où

Proposition 4.1. *Tout sous-graphe induit d'un graphe triangulé est triangulé. Donc les graphes triangulés sont héréditaires.*

Définition 4.2. On dit qu'un sommet x est **simplicial**, si l'ensemble de ses voisins $N(x)$ induit une clique. Ainsi, $N[x] = N(x) \cup \{x\}$ est une clique maximale, et elle est l'unique clique maximale contenant x.

Définition 4.3. Soit $\sigma = (x_1, x_2, ..., x_n)$ un ordre sur V. On dit que σ est un ordre d'élimination simplicial ou ordre d'élimination parfait (o.e.p) si pour $i = 1...n$, x_i est un sommet simplicial dans le sous graphe G_i induit par les sommets $x_i, x_{i+1}...x_n$.

Le théorème suivant caractérise les graphes triangulés, et il est à la base d'un algorithme de reconnaissance de cette classe de graphes.

Théorème 4.2. *[6, 7, 8] Soit $G = (V, E)$ un graphe. Les propriétés suivantes sont équivalentes :*

1. *G est triangulé.*
2. *G admet un ordre d'élimination simplicial.*
3. *Tout ordre LexBFS sur les sommets de G est un ordre d'élimination simplicial.*

Corollaire 4.3. *Soit $G = (V, E)$ un graphe ordonné par Lex-BFS $\sigma = (v_1, v_2, ...v_n)$.*
Pour que G soit triangulé il faut et il suffit que, si $u < v < w$ tels que $u \to v$, $u \to w$, alors $v \to w$.

Démonstration. La condition est nécessaire :
Soient G un graphe triangulé et u, v, w trois sommets tels que $u < v < w$, $u \to v$ et $u \to w$.
Soit G_u le sous-graphe induit par les sommets qui sont supérieures ou égales à u. Puisque $\sigma = LexBFS$, alors le sommet u est simplicial dans G_u (théorème 4.2). Il en résulte que si $u \to v$, $u \to w$ alors $vw \in E$, et puisque $v < w$, alors $v \to w$.

La condition est suffisante :
Par absurde, supposons que G n'est pas triangulé, donc G contient un trou C_k, $k \geq 4$. Posons $C_k = (x_1, x_2, x_3..., x_{k-1}, x_k)$ tel que x_1 est le plus petit sommet pour σ sur C_k, donc $x_1 \to x_2$ et $x_1 \to x_k$. Mais ceci implique, d'après l'hypothèse, que $x_2 \to x_k$ si $x_2 < x_k$ ou $x_k \to x_2$ si $x_k < x_2$. Dans les deux cas $x_2x_k \in E$, c'est à dire x_2x_k est une corde pour C_k. Absurde, car C_k est un trou. □

Définition 4.4. Dans un graphe G ordonné par $LexBFS$, pour tout sommet v, on note $m(v)$ le plus petit des voisins supérieurs de v, c'est à dire $m(v) = \min\{w, v \to w\}$.

Corollaire 4.4. *Si $G = (V, E)$ un graphe ordonné par LexBFS $\sigma = (v_1, v_2, ...v_n)$. Alors*
Pour que G soit triangulé il faut et il suffit que, $\forall u, v \in V$ tels que $u < v$, $m(u) \neq v$, si $u \to v$, alors $m(u) \to v$.

Algorithme 4.1. *Reconnaissance d'un graphe triangulé*
Donnée : *Un graphe $G(V, E)$ et $\sigma = LexBFS$ sur V.*
Sortie : *Vrai si et seulement si $\sigma = (v_1...v_n) = o.e.s.$*
Début

1. *Pour $i = 1...n$ faire*
2. *Pour $y \in N^+(v_i) \setminus \{m(v_i)\}$ faire*
3. *Si $y \notin N^+(m(v_i))$, alors ;*
4. *Retourner FAUX*
5. *Fin si*
6. *Fin pour*
7. *Fin pour*
8. *Retourner VRAI.*

fin

Complexité : La complexité de cet algorithme est en $O(m)$.

Définition 4.5. Un séparateur dans un graphe G est un ensemble de sommets S dont la suppression augmente le nombre de composantes connexes. Un séparateur est minimale si il ne contient pas strictement un autre séparateur.

Il est connu qu'un graphe G est triangulé si et seulement si tout séparateur minimal est une clique.

Définition 4.6. Un **arbre de cliques maximales** d'un graphe triangulé G est un arbre T_G dont les sommets sont les cliques maximales de G et les arêtes sont les séparateurs minimaux. De plus, pour chaque sommet de G, le sous graphe de T_G induit par les cliques maximales contenant celui ci est un sous arbre.

Il existe un autre algorithme de reconnaissance des graphes triangulé qui utilise un algorithme simple basé sur LexBFS pour construire un arbre de cliques et vérifier par la suite la proposition de Gavril [1] qui dit qu'un graphe est triangulé si et seulement si c'est le graphe d'intersection des sous-arbres d'un arbre. Cela signifie que les cliques maximales peuvent être arrangées sous forme d'un arbre tel que les cliques contenant un sommet donné induisent un sous-arbre et un tel arbre est appelé arbre de cliques.

Algorithme 4.2. *Lex-BFS et arbre de cliques*
Données : *Un graphe $G = (V, E)$*
Résultat : *Si G est triangulé : un ordre d'élimination simpliciale et un arbre de cliques $T = (I, F)$*

1. ***Début***
2. ***pour chaque*** *sommet $x \in V$ faire*
3. *Ètiquette $(x) = \emptyset$*
4. *Ètiquette précédente $= \emptyset$*
5. *$j = 0$*
6. ***Pour*** *$i = n$ jusqu'à 1 faire*
7. *Choisir un sommet non numéroté $x \in V$ d'étiquette maximum*
8. ***Si*** *étiquette précédente $\nsubseteq$ étiquette (x)* ***alors***
9. *$j = j + 1$*
10. *Créer la clique maximale $C_j =$étiquette $(x) \cup (x)$*
11. *$C(dernier(x))$ est le père de C_j dans T*
12. *L'arc de l'arbre C_jC (dernier(x)) est étiqueté par le séparateur minimal*
13. *$S_j = C_j \cap C(dernier(x)) =$ étiquette (x)*
14. ***Sinon*** *$C_j = C_j \cup \{x\}$*
15. ***Pour chaque*** *voisin non numéroté y de x* ***faire***
16. *Ajouter i à étiquette(y)*
17. *$dernier(y) = x$*
18. *étiquette précédente $=$étiquette(x)*
19. *$\lambda(i) \leftarrow x$*
20. *$C(x) = j$*

Pour la partie vérification :

1. Vérifier que dans une séquence croissante tous les sommets se sont marqué.
2. On s'assure que tous les sommets qui ont marqué une séquence croissante sont contenu dans la clique.

4.2 Arbre

Définition 4.7. Un **arbre** (tree) est un graphe connexe et sans cycle (acyclique).

Exemple 4.1. Dans la figure 4.1, les graphes G_1 et G_2 ne sont pas des arbres, car G_1 n'est pas connexe et G_2 n'est pas acyclique. Par contre, le graphe G_3 est connexe et acyclique, donc est un arbre.

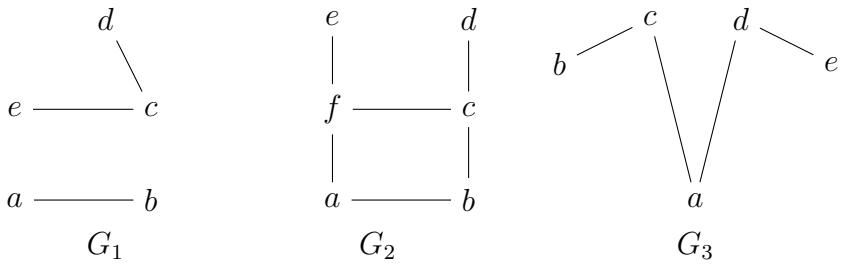

FIGURE 4.1: G1 grahe non connexe,G2 n'est pas acyclique et G3 graphe connexe et acyclique

Les détails sur les propositions de cette section peuvent être trouvés dans le livre [16].

Proposition 4.5. *Un arbre ayant au moins une arête (c'est à dire $n \geq 2$) possède au moins deux sommets pendants, c'est-à-dire de degré 1 (un sommet pendant est aussi appelé feuille).*

Démonstration. Soit $\mu = (x_1, x_2..., x_k)$ une chaine élémentaire maximale, c'est-à-dire non strictement contenue dans une chaîne élémentaire. Il s'agit de montrer que les extrémités de μ sont des feuilles.
Par l'absurde, supposons que le sommet x_1, par exemple, ne soit pas pendant. Donc x_1 est adjacent à un autre sommet $y \neq x_2$. On a deux cas :
1. Si le sommet y est un sommet de la chaine, alors il existe dans le graphe un cycle, ce qui contredit l'hypothèse que ce graphe est un arbre.
2. Si le sommet y n'est pas l'un des sommets de la chaîne, alors celle-ci n'est pas maximale, car strictement contenue dans la chaîne $(y, x_1, x_2..., x_k)$. Dans les deux cas, on aboutit à une contradiction. □

Proposition 4.6. *Dans un arbre, deux sommets quelconques sont reliés par une unique chaîne élémentaire.*

Définition 4.8. Un **isthme** d'un graphe G est une arête e dont la suppression augmente le nombre de composantes connexes d'une unité, c'est à dire $G - e$, le sous graphe obtenu par la suppression de l'arête e (sans supprimer ses extrémités) a une composante connexe de plus que G. On dit parfois aussi que l'arête $e = xy$ sépare les sommets x et y.

Remarque 4.1. Lorsque G est connexe, un isthme est une arête e telle que $G - e$ n'est pas connexe.

Lemme 4.7. *Une arête d'un graphe G est un isthme si et seulement si elle n'appartient pas à un cycle de G.*

Remarque 4.2. Dans un arbre toutes les arêtes sont des isthmes.

Théorème 4.8. *Les conditions suivantes pour un graphe G sont équivalentes :*

1. *G est un arbre.*
2. *G est connexe et $m = n - 1$.*
3. *G est connexe et minimal pour cette propriété.*
4. *G est acyclique et $m = n - 1$.*
5. *G est acyclique et maximal pour cette propriété.*
6. *G est connexe et toute arête est un isthme.*

Puisque un arbre est sans cycle, il est sans trou, d'où l'observation suivante.

Observation 1. *Tout arbre est un graphe triangulé.*

Rappelons qu'un graphe est triangulé si et seulement tout ordre LexBFS est un ordre d'élimination simplicial. Les sommets simpliciaux dans un arbre sont les sommets pendants, d'où l'observation suivante.

Observation 2. *Un graphe G est un arbre si et seulement si tout ordre LexBFS est un ordre d'élimination sommet pendant.*

Ainsi, le parcours LexBFS peut aussi être utilisé pour reconnaitre les arbres comme nous allons le voir ci-dessous

Algorithme 4.3. *Reconnaissance d'arbre*
***Entrée** : Un graphe G ordonné par lexBFS σ.*
***Sortie** : G est un arbre si et seulement σ est un ordre d'élimination sommet pendant.*

1. *Début*
2. *pour* $i = 1...n-1$ *faire*
3. *si* $|N^+(v_i)| \neq 1$, *alors retourner faux*
4. *fin si*
5. *retourner vrai*
6. *Fin*

Théorème 4.9. *L'algorithme 4.3, reconnait un arbre avec une complexité en* $O(n)$

Démonstration.

1. **La correction de l'algorithme**
   D'après l'observation 2, l'algorithme 4.3 renvoie vrai si et seulement si le graphe d'entrée est un arbre.
2. **Complexité**
   Le pire des cas correspond au cas où G est un arbre. Nous avons au plus $n-1$ itérations et dans chaque itération, l'algorithme teste si v_i est pendant, donc, fait une opération élémentaire. Par conséquent, à la fin on aura le nombre d'opérations est en $O(n-1) = O(n)$.

□

4.3 Graphes d'intervalles

Définition 4.9. Un graphe $G = (V, E)$ est d'**intervalles** s'il existe une famille de n intervalles $\mathcal{I} = \{I_1, I_2, ..., I_n\}$ dans $\mathbb{R}$ et une bijection de V dans $\mathcal{I}$ qui à tout $v \in V$, fait associer un intervalle $I_v \in \mathcal{I}$ tels que

$$uv \in E \text{ si et seulment si } I_u \cap I_v \neq \varnothing$$

Autrement dit, G est le graphe d'intersection d'un ensemble d'intervalles de la droite réelle.

Les graphes d'intervalles apparaissent naturellement dans le processus de modélisation de situations réelles, en particulier celles impliquant des dépendances de temps ou d'autres restrictions qui sont de nature linéaire. On l'utilise dans divers domaines tel que l'archéologie, la biologie, la psychologie, la sociologie, la gestion, la génétique ....
On peut vérifier facilement qu'un graphe d'intervalles est un graphe sans trou, d'où

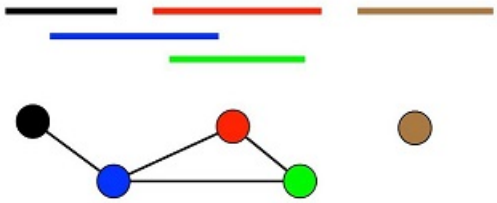

FIGURE 4.2: Graphe d'intervalles

Propriétés. Tout graphe d'intervalles est triangulé.

Théorème 4.10. *[17] Un graphe est un graphe d'intervalles si et seulement si ses cliques maximales peuvent être ordonnées linéairement de telle sorte que pour chaque sommet du graphe le maximum des cliques auxquelles il appartient se produisent consécutivement dans l'ordre linéaire.*

Habib, Paul et Viennot [4] ont développé un algorithme en temps linéaire qui est relativement facile à mettre en œuvre. Il utilise la recherche lexicographique en largeur d'abord pour déterminer que le graphe donné est triangulé et produire un arbre de cliques. Ils manipulent ensuite cette arbre de cliques dans un chemin de clique et après il voit si le théorème 4.10 est vérifié, cet algorithme se fait en trois étapes :

1. Utiliser LexBFS pour faire un ordre des cliques maximales est formé un arbre de cliques (théorème 4.2).
2. Partitionné l'ensemble des cliques en s'appuyant sur un arbre de cliques (La partition du début est constituée de la dernière clique visitée par LexBFS et des autres cliques).
3. Vérifié que chaque sommet doit se trouver dans des cliques consécutives au sein de la chaîne de cliques. Sinon, le graphe n'est pas un graphe d'intervalles.

Algorithme 4.4. *Partitionnement des cliques maximales*
Données : *un graphe* $G = (V, E)$
Résultat : *Si* G *est un graphe d'intervalles : une chaine de cliques* L

1. ***Début***
2. *Calculer les cliques maximales et un arbre de cliques* $T = (I, F)$ *en utilisant l'algorithme 4.2*

3. *Si G n'est pas triangulé* ***alors*** *retourner "G n'est pas un graphe d'intervalle"*
4. *Soit I l'ensemble des cliques maximales $I = \{(C_1, ..., C_k)\}$*
5. *Soit L la liste ordonnée (I)*
6. *Pivots $= \emptyset$ est un pile vide*
7. ***tant que*** *il existe une classe I_c qui n'est pas un singleton dans $L = (I_1, .., I_l)$*
8. ***faire***
9. ***si*** *pivots$= \emptyset$* ***alors***
10. *Soit C_l la dernière clique de I_c découverte par LexBFS*
11. *Remplacer I_c par $I_c \setminus \{C_l, C_l\}$ dans L*
12. *$C = C_l$*
13. ***Sinon*** *choisir un sommet x non utilisé de pivots (supprimés ceux qui ont été utilisés)*
14. *Soit C l'ensemble de toutes les cliques contenant x*
15. ***Si*** *toutes les cliques de C apparaissent dans des classes consécutives* ***alors***
16. *Soit I_a la première classe contenant une telle clique*
17. *Soit I_b la dernière classe contenant une telle clique*
18. ***Sinon*** *retourner "G n'est pas un graphe d'intervalle"*
19. ***si*** *une classe strictement entre I_a et I_b contient une clique n'appartenant pas à C*
20. ***alors*** *retourner "G n'est pas un graphe d'intervalle"*
21. *Remplacer I_a par $I_a \setminus C$, $I_a \cap C$ et I_b par $I_b \setminus C$, $I_a \cap C$*
22. ***Pour chaque*** *arc C_iC_j de l'arbre de cliques connectant une clique $C_i \in C$ a une clique C_j n'appartient pas à C faire*
23. *pivots=pivots $C_i \cap C_j$*
24. *Supprimer C_iC_j de l'arbre de cliques*
25. ***fin***

Cet algorithme calcule une chaine de cliques si et seulement si le graphe est un graphe d'intervalle et sa complexité est en $O(n + m)$.

CHAPITRE 5

LEXBFS ET PROBLÈME DE COLORATION

5.1 Coloration d'un graphe triangulé

5.1.1 Coloration d'un graphe

La Coloration d'un graphe G consiste à attribuer des couleur à ses sommets ou à ses arêtes en respectant une certaine condition. L'une des coloration les plus utilisée est la coloration propre.

Définition 5.1. Une **coloration propre** des sommets d'un graphe $G = (V, E)$ est une application $c : X \to C$ vérifiant la condition suivante, quel que soient $x, y \in V$

$$c(x) = c(y) \Leftrightarrow uv \notin E.$$

où C est l'ensemble des couleurs. Le nombre minimum de couleur possibles dans une coloration propre d'un graphe G est appelé **nombre chromatique** de G, il est noté $\chi(G)$.

Remarque 5.1. Notons que l'ensemble des couleurs peut être représenté par un sous-ensemble des entiers.

Définition 5.2. Soit G un graphe ordonné par un ordre σ. La coloration des sommets de G selon l'ordre σ en attribuant à chaque itération, à chaque sommet, la plus petite couleur non attribuée à ses voisins inférieurs.

Algorithme 5.1. COLORATION GLOUTONNE
Données : *Un graphe $G = (V, E)$ ordonné par $\sigma = (v_1, v_2...v_n)$et un ensemble de couleurs $C = \{1, 2, 3, ...\}$.*
Résultat : *Une coloration propre par un nombre minimum de couleurs.*

1. *Début*
2. *Pour i de 1 à n faire*
3. *$c(v_i)$ est la plus petite couleur non utilisée par les voisins inférieurs de v_i.*
4. *Fin Pour*
5. *Retourner c*
6. *Fin*

Remarque 5.2. La coloration gloutonne ne donne pas nécessairement une coloration optimale. En effet, le nombre de couleurs données par l'algorithme sur le graphe de la figure 5.1 est 3, alors que ce même graphe peut être colorer par deux couleurs. Cependant, il existe toujours un ordre des sommets pour lequel une coloration gloutonne donnera une coloration optimale. Le problème qui se pose dans ce cas est que, pour un graphe d'ordre n, il faut calculer $n!$ ordres possibles, ce qui n'est pas possible à réaliser en pratique quand n dépasse une certaine constante, par exemple quand $n = 30$.

FIGURE 5.1: Coloration gloutonne

5.1.2 Coloration d'un graphe triangulé

Pour obtenir une coloration optimale, pour la classe des graphes triangulés, on utilise l'algorithme de LexBFS COLOR qui se base sur deux étapes qui sont les suivantes :

- Appliquer LexBFS pour ordonner les sommets.
- Appliquer la coloration gloutonne sur l'ordre inverse obtenu par LexBFS.

5.2 Problème de l'emploi du temps

Une école doit organiser les examens des matières optionnels de ses élèves de troisième année. Les différentes options sont : Français(F) ; Anglais(A) ;

Mécanique(M) ; Sport(S) ; Informatique(Ie) ; Dessin(D) ; Espagnol(E) ; Théâtre(T) ; Musique(m) et Italien(i).

L'administration trouve un problème dans la planification de l'emploi du temps de ses examens, car certains étudiants ont choisi plusieurs options (cela signifie que ses options ne doivent pas être programmées en même temps) Le tableau ci-dessous présente les matières dont chaque étudiants est concerné.

étudiants	Matiéres
E1	F,A,M
E2	M,I
E3	I,A,S
E4	I,E
E5	E,S,D
E6	T,E,m
E7	E,m,i
E8	T,I

La question qu'on se pose est : combien de demi-journée seront-elles nécessaires à cette organisation sachant que la durée de chaque épreuve est d'une demi-journée ?
Tel qu'il est posé, il existe généralement plusieurs solutions à ce problème, et nous vous proposons ici de trouver le meilleur horaire, c'est-à-dire celui qui minimise le nombre de créneaux nécessaires. La résolution de ce problème va se faire sur trois étapes :

1. Modéliser le problème sous forme de graphe : quels sont ses sommets ? quelles sont ses arêtes ?
2. On va passer à la partie reconnaissance pour pouvoir déterminer à quelle classe de graphe il appartient.
3. Définir la méthode de résolution.

5.2.1 La modélisation sous forme de graphe

Ce problème peut être ramené à la recherche d'une coloration minimum dans un graphe qu'on va définir comme suit : chaque matière sera représentée par un sommet donc l'ensemble des sommets $V = \{F, A, M, S, I, D, E, T, m, i\}$. Deux sommets sont adjacents si et seulement s'il existe un étudiant concerné par les matières qu'ils représentent, c'est-à-dire, les matières que les deux sommets représentent ne peuvent pas

avoir lieu en même temps.

Ainsi, nous trouvons le graphe donné par la figure 5.2

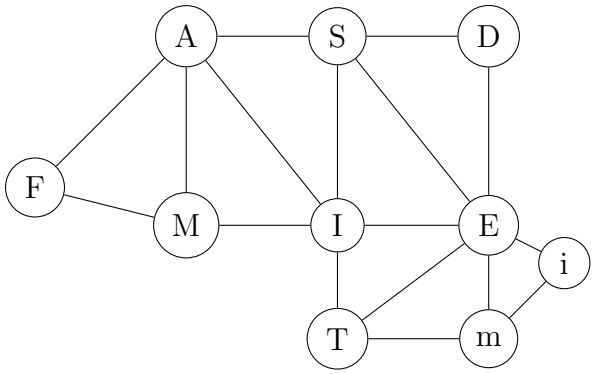

FIGURE 5.2: Modélisation graphique du problème

5.2.2 Reconnaissance du graphe

On voit bien que le graphe est triangulé mais pour en être sur on va appliquer LexBFS et s'il admet un ordre d'élimination parfait alors on dira qu'il triangule.
Après l'application de LexBFS on aura ce graphe :

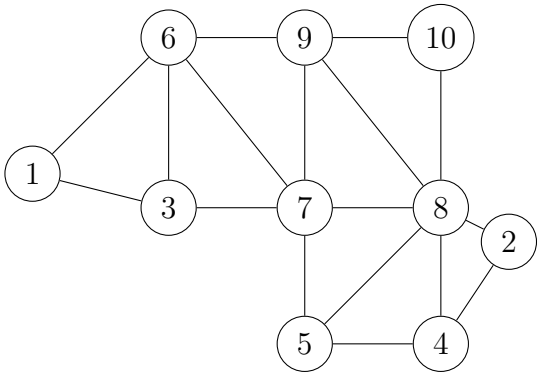

FIGURE 5.3: Ordre LexBFS

le graphe admet bien un ordre d'élimination parfait, le sommet 1 et bien un sommet simplicial on aura ce graphe après suppression des sommet :

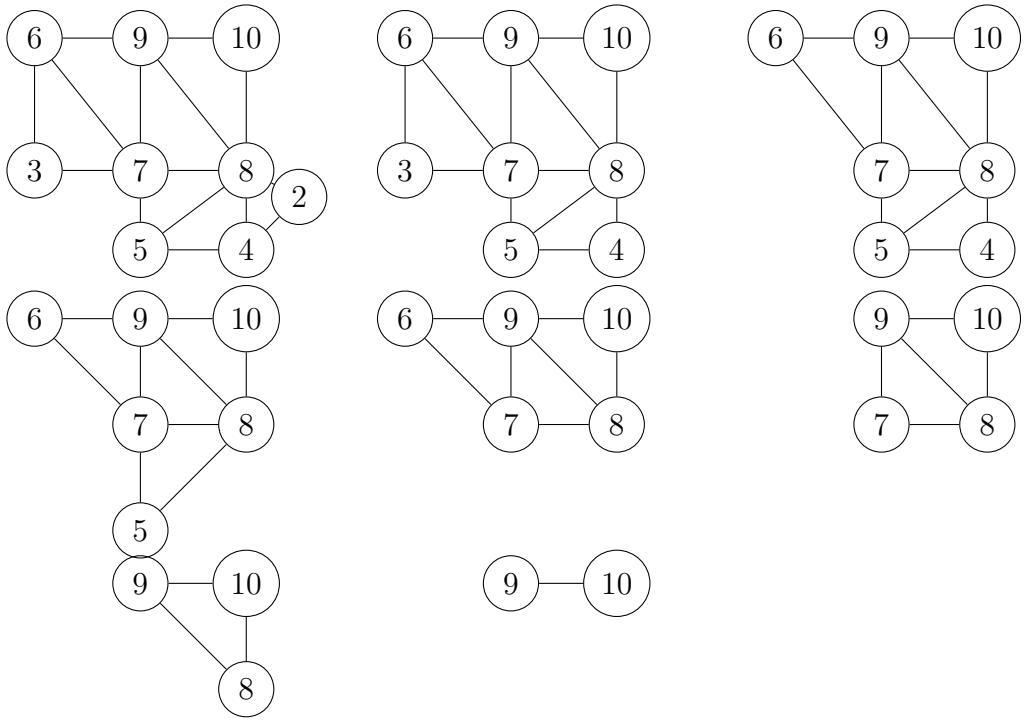

FIGURE 5.4: Ordre d'élimination parfait

Conclusion : le graphe admet bien un ordre d'élimination parfait donc il est triangulé.

5.2.3 Définire la méthode de résolution :

Pour trouver la solution on va appliquer un algorithme de coloration sur le graphe ou le nombre de couleurs va représenter le nombre de demi-journée et les sommets qui ont la même couleur c'est les matières qui peuvent être programmé en même temps.
on va utiliser LexBFS color qu'ont à présenter précédemment.

Sommets	F	i	M	m	T	A	I	E	S	D
Oredre LexBFS	1	2	3	4	5	6	7	8	9	10

Après l'application de l'algorithme de coloration figure 5.5 on a trouvé que le graphe et 3-colorables cela signifie qu'on a besoin de 3 demi-jours pour

organiser les examens de l'école et tous les sommets qui ont la même couleur représentent les matières qui peuvent être organisé en même temps.
les solution sont montrés dans le tableau suivant :

demi-journée 1	demi-journée 2	demi-journée 3
musique(m)	Anglais(A)	italien(i)
Informatique(I)	Espagnol(E)	Mecanique(M)
Dessin(D)		Sport(S)
Français(F)		Théatre(T)

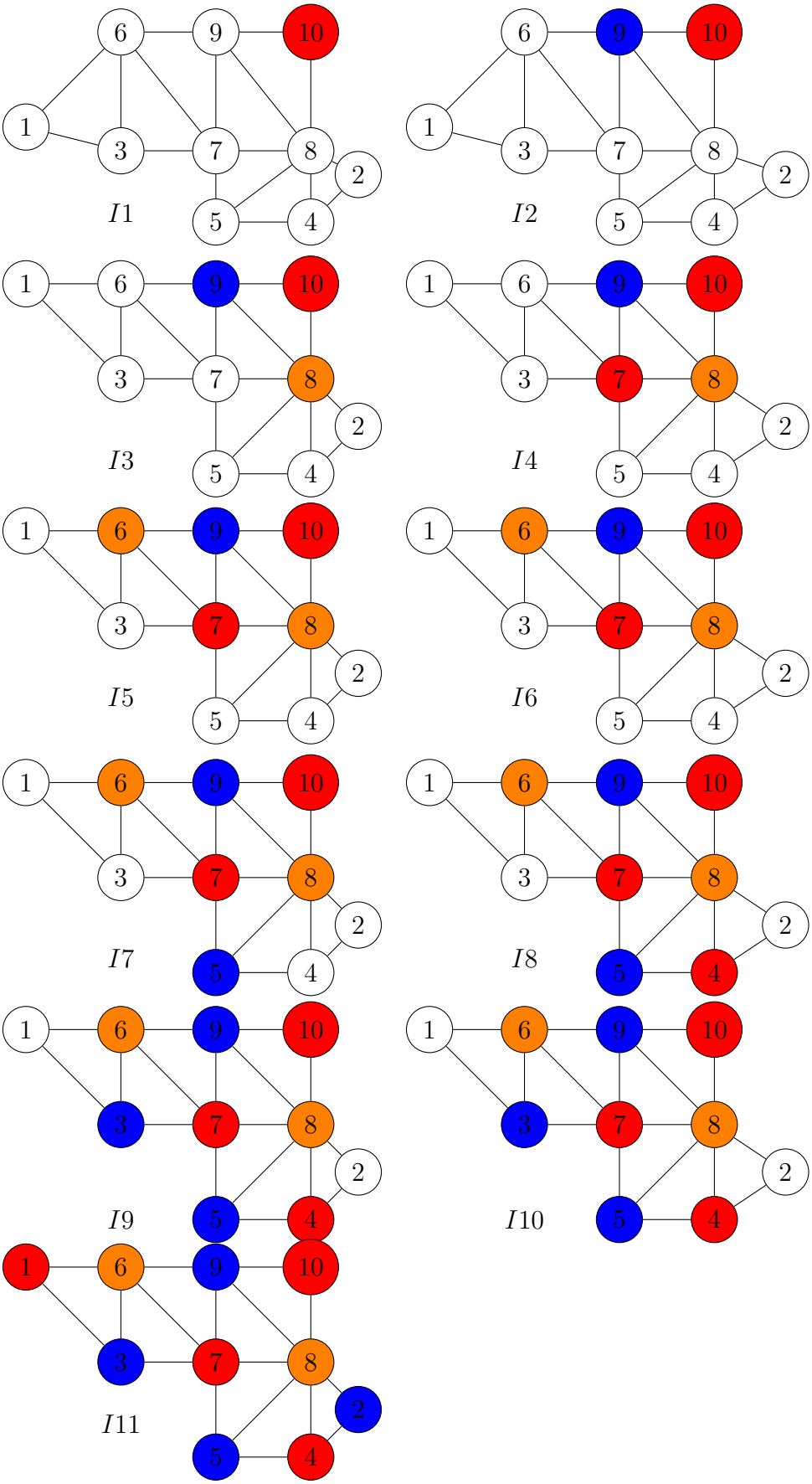

FIGURE 5.5: Application de la coloration Gloutonne

CONCLUSION

Pour résoudre un problème modélisé par l'outil de graphe, on est obligé de parcourir les sommets de ce dernier pour l'explorer. Les parcours les plus utilisés sont le parcours en largeur BFS et le parcours en longueur DFS. Notre étude porte sur un parcours très particulier, le parcours en largeur lexicographique LexBFS. le parcours LexBFS, qui est une variante de BFS, à été introduit par Rose et al en 1976 pour la reconnaissance d'une classe de graphes, les graphe triangulé. Au départ, ce parcours n'a pas été popularisé, mais, à partir des années 90 ses applications n'ont pas cessé de se multiplier.

Dans ce mémoire, nous avons présenté une petite synthèse sur quelques applications du parcours LexBFS qui sont la reconnaissance de classes de graphes, la résolution d'une manière efficace de quelques problèmes, qui sont NP-complet en général, sur ses classes de graphes....
Nous avons aussi présenté un exemple concret que nous avons modélisé par un problème de coloration dans un graphe triangulé.

BIBLIOGRAPHIE

[1] R. G. Michael and D. S. Johnson, Computers and intractability : a guide to the theory of NP-completeness, Freeman, 1979.

[2] Mémoire MERAKECHI. s, TISSELI.f ,université saad Dahlab Blida1 faculté des science ,département mathématiques spécialité Recherche Opérationnelle thème : nombre chromatique et nombre chromatique équitable d'un graphe 2016-2017.

[3] F.Gavril. The intersection graphs of a path in a tree are exactly the chordal graphs.journ. Comb.Theory,16 :47-56,1974

[4] P. Galinier, M. Habib, and C. Paul. Chordal graphs and their clique graph. In Graph-Theoretic Concepts in Computer Science, WG'95,volume 1017 of LNCS, pages 358-371, 1995.

[5] K.S. Booth and G.S. Leuker. Testing for the consecutive ones property, interval graphs and graph planarity using pq-tree algorithm. J.Comput. Syst. Sci., 13 :335 -379, 1976.

[6] Gabriel A. Dirac. On rigid circuit graphs. Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg 25, Universität Hamburg, 196.

[7] Delbert R. Fulkerson and O. A. Gross. Incidence matrices, interval graphs, and seriation in archaeology. Pacific Journal of Mathematics, 28 :565-570, 1969.

[8] F nic Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. Journal of Combinatorial Theory Series B, 16 :47-56, 1974.

[9] N. Kortee and R. Mohring. An incremental linear-time algorithm for recognizing interval graphs. SIAM J. of Comput., 18 :68-81, 1989.

[10] Hsu and Ma. Substitution decomposition on chordal graphs and applications. In Proceedings of the 2nd ACM-SIGSAM Internationnal Symposium on Symbolic and Algebraic Computation, number 557 in LNCS, 1991.

[11] Cook,S.,the complexity of theorem proving Procedures,Proceedings of the thrid annual ACM Symposium on theory of computing,ACM New York, 2570,P. 151-158.

[12] M.Habib ,Ross McConnell,Christophe Paul,L.Viennot.Lex-BFS and Partition refinement with application to transitive orientation interval graph recognition and consecutiv Ones testing,Theoretical Computer Science 234 (2000) 59-84.

[13] Christophe Paul,Laurent viennot.Quelques algorithme linéaire de reconnaissance autour de Lex-BFS,[Rapport de recherche]1997.

[14] Martin Charles GOLUBIC ,Algorithme graph theory and perfect graphs ,second Edition.

[15] D.J. Rose, R.E. Tarjan, and G.S. Lueker : Algorithmic aspects of vertex elimination on graphs, SIAM J. Comput. 5 (1976) 266-283.

[16] J.C.Fournier, Théorie des graphes et applications avec exercices et problèmes, 2^e édition

[17] P.C Gilmore and A.J. hoffman,A characterization of comparability graphs and interval graphs, Canad. J.Math (1964) ,539-548.

ملخص

في هذا العمل ، قدمنا عرضًا واسعًا حول خوارزميات استكشافات الرسوم البيانية ولكن بشكل أكثر دقة على اجتياز العرض المعجمي .
وهكذا قمنا بتعريف الخوارزميات بناءا على المسار المعجمي للتعرف على الرسم البياني (المثلث ، الفاصل الزمني) وأهميتها في حل بعض المشكلات اليومية (على سبيل المثال مشكلة الجدول الزمني)

Résumé

Dans ce travail on a fait une vaste présentation sur les algorithmes d'explorations des graphes mais plus précisément le parcours en largeurs lexicographic (Lex BFS).
Ainsi on a défini des algorithmes basés sur LexBFS pour la reconnaissance deS graphes triangulés, intervalles et leurs importances dans la résolution de certains problèmes de tous les jours (ex : Problème de l'emploi du temps).

Abstract

In this work we made a vast presentation on the algorithms of explorations of the graphs but more precisely the traversal in lexicographic widths (Lex BFS).
Thus we have defined algorithms based on LexBFS for graph recognition (triangulated, interval) and their importance in solving certain everyday problems (eg: Problem of the timetable).