UNIVERSITE DE M'SILA

FACULTE DES MATHEMATIQUES ET DE L'INFORMATIQUE

Département de Mathématiques

MEMOIRE DE FIN D'ETUDE

Présenté pour l'obtention du diplôme de **Master**

Domaine : Mathématiques et Informatique

Filière : Mathématiques

Option : Mathématiques Appliquées et discrète

Par

BENABDELOUAHAB RACHIDA

Sujet

ETUDE SUR LES APPLICATIONS DE LA FERMETURE REFLEXIVE ET TRANSITIVE D'UNE RELATION BINAIRE R SUR UN ENSEMBLE E

Dirigé par :

Mr. **MIHOUBI DOUADI**

Promotion: 2011/2012

Remerciements

Je rends ma profonde gratitude à dieu qui m'a aidé à réaliser ce modeste travail.

Je tiens tout particulièrement à exprimer ma profonde gratitude à mes parents pour leur encouragement, leur soutien et pour les sacrifices qu'ils ont enduré

Je tiens à remercier vivement mon promoteur Dr : MIHOUBI. D, d' avoir accepté de diriger ce travail et de créer autour de moi un environnement de recherche par ses conseils et son soutien permanent et je remercie, ainsi qu'à toute l'équipe du département de mathématiques.

En fin Je tiens à remercier tous ceux qui m'ont participé de prés ou de loin à l'élaboration de ce travail.

Table des matières

Introduction

La théorie des graphes constitue un domaine des mathématiques qui historiquement, s'est aussi développé au sein de disciplines diverses telles que la chimie (modélisation de structures), la biologie (génome), les sciences sociales (modélisation des relations) ou en vue d'applications industrielles (problème du voyageur de commerce).

De manière générale, un graphe permet de représenter simplement la structure, les connexions, les cheminements possibles d'un ensemble complexe comprenant un grand nombre de situations, en exprimant les relations, les dépendances entre ses éléments (e.g., réseau de communication, réseaux ferroviaire ou routier, arbre généalogique, diagramme de succession de tâches en gestion de projet, ...).

En plus de son existence purement mathématique, le graphe est aussi une structure de données puissante pour l'informatique.

Alors dans ce mémoire, on s'intéresse aux applications sur les graphes notamment l'étude des problèmes : plus court chemin, flot maximal, et implémentation de quelques algorithmes en Matlab.

Chapitre 1

Etude sur les concepts des graphes

1.1 Un bref historique de la théorie des graphes

Tout le monde s'accorde à considérer que la théorie des graphes est née en 1736 avec la communication d'Euler $(1707-1783)$ dans laquelle il proposait une solution au célèbre problème des ponts de Königsberg. Le problème posé était le suivant : Deux îles A et D sur la rivière Pregel à Königsberg (alors capitale de la Prusse de l'Est, aujourd'hui rebaptisée Kaliningrad) étaient reliées entre elles ainsi qu'aux rivages B et C à l'aide de sept ponts (désignés par des lettres minuscules) comme le montre la figure 1.1.

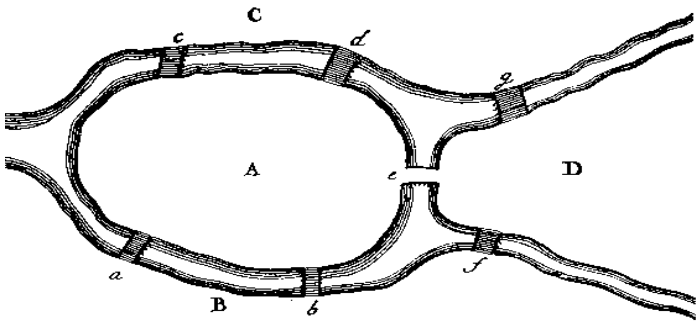


FIG. 1.1 : La rivière Pregel et l'île de Kneiphof

Le problème posé consistait, à partir d'une terre quelconque A, B, C, ou D, à traverser chacun des ponts une fois et une seule et à revenir à son point de départ (sans traverser la rivière à la nage !). Euler représenta cette situation à l'aide d'un "dessin" où les sommets représentent les terres et les arêtes, les ponts comme le montre la figure 1.2.

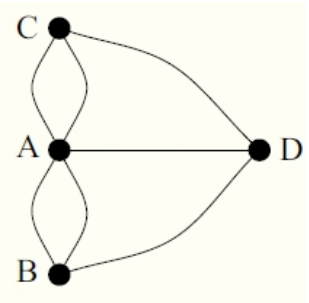


FIG. 1.2 : Graphe associé au problème des ponts de Königsberg

Comme nous le montrerons ultérieurement, Euler démontra que ce problème n'a pas de solution. Le problème des ponts de Königsberg est identique à celui consistant à tracer une figure géométrique sans lever le crayon et sans repasser plusieurs fois sur un même trait.

Pendant les cent années qui suivirent, rien ne fut fait dans ce domaine de recherche. En 1847, Kirchhoff $(1824-1887)$ développa la théorie des arbres pour l'appliquer à l'analyse de circuits électriques. Dix ans plus tard, Cayley $(1821-1895)$ découvrit la notion d'arbre alors qu'il essayait d'énumérer les isomères saturés des hydrocarbures de type C_nH_{2n+2}.

A cette époque, deux autres problèmes d'importance pour la théorie des graphes furent également proposés et partiellement résolus.

Le premier est *la conjecture des quatre couleurs* qui affirme que quatre couleurs suffisent pour colorier n'importe quelle carte plane telle que les "pays" ayant une frontière commune soient de couleurs différentes. C'est sans doute Möbius $(1790-1868)$ qui présenta le premier ce problème dans l'un de ses cours en 1840. Environ dix ans après, de Morgan $(1806-1871)$ essaya de résoudre ce problème. Les lettres de de Morgan à ses divers collègues mathématiciens constituent les premières références à la conjecture des

quatre couleurs. Le problème devint célèbre après sa publication, par Cayley en1879, dans le premier volume des *Proceedings of the Royal Geographic Society.* Ce problème est restétrès longtemps sans solution. Il fallut attendre jusqu'en 1976 pour que Appel et Haken prouvent ce théorème en réduisant le problème à un nombre fini de situations particulières et en trouvant une solution pour chacune d'entre elles à l'aide d'un ordinateur.

Le second problème est dû à Sir Hamilton $(1805-1865)$. En 1859, il inventa un casse-tête qu'il vendit pour 25 guinées à un fabricant de jouet de Dublin. Ce jeu consiste en un dodécaèdre régulier en bois (un polyèdre à 12 faces et 20 sommets), chaque face étant un pentagone régulier comme le montre la figure 1.3.

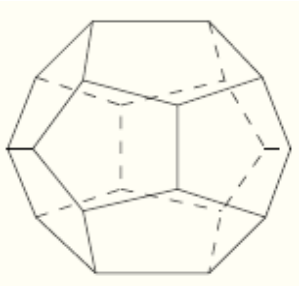


FIG. 1.3 : Un dodécaèdre régulier

Trois arêtes sont donc issues de chaque sommet. Un clou est fiché sur chaque sommet marqué du nom de vingt grandes villes mondiales. Le casse-tête consiste à enrouler une ficelle passant une fois et une seule fois par chacune des villes (sommets). Bien que la solution de ce problème soit aisée à obtenir, personne n'a encore trouvé de condition nécessaire et suffisante de l'existence d'un tel chemin (appelé chemin Hamiltonien) dans un graphe quelconque.

Cette période fertile fut suivie d'un demi-siècle de relative inactivité. Les années 1920 virent la résurgence de l'intérêt pour les graphes. L'un des pionniers de cette période fut König à qui l'on doit le premier ouvrage consacré entièrement à la théorie des graphes (König, 1936). Il est sans doute à l'origine de l'utilisation du terme "graphe" pour désigner ce qui était préalablement considéré comme un ensemble de "points et de flèches".

A partir de 1946, la théorie des graphes a connu un développement intense sous l'impulsion de chercheurs motivés par la résolution de problèmes concrets. Parmi ceux-ci, citons de manière privilégiée Kuhn (1955), Ford et Fulkerson (1956) et Roy (1959). Parallèlement, un important effort de synthèse a été opéré en particulier par Claude Berge. Son ouvrage *"Théorie des graphes et ses applications"* publié en 1958 (Berge, 1958) marque sans doute l'avènement de l'ère moderne de la théorie des graphes par l'introduction d'une *théorie des graphes* unifiée et abstraite rassemblant de nombreux résultats épars dans la littérature. Depuis, cette théorie a pris sa place, en subissant de très nombreux développement essentiellement dus à l'apparition des calculateurs, au sein d'un ensemble plus vaste d'outils et de méthodes généralement regroupées sous l'appellation "recherche opérationnelle" ou "mathématiques discrètes".

1.2 Définitions générales

1.2.1 Relation binaire

Définition 1.1.

une relation binaire entre deux ensembles A et B est définie comme un sous ensemble R du produit cartésien $A \times B$. Dans le cas particulière $A = B$ on dit simplement que R décrit une relation sur A.

1.2.2 Concepts orientés

Dans beaucoup d'applications, les relations entre éléments d'un ensemble sont orientées, i.e., un élément x peut être en relation avec un autre y sans que y soit nécessairement en relation avec x. On parle alors de graphe orienté (en Anglais directed graph ou plus simplement digraph).

Un graphe $G = (X, U)$ est déterminé par :

– un ensemble $X = \{x_1, x_2, ..., x_n\}$ dont les éléments sont appelés sommets ou nœuds

(ce dernier terme est plutôt dans le contexte des réseaux) ;

– un ensemble $U = \{u_1, u_2, ..., u_m\}$ du produit cartésien $X \times X$ dont les éléments sont appelés arcs.

Pour un arc $u = (x_i, x_j)$, x_i est l'extrémité initiale, x_j l'extrémité finale (ou bien origine et destination). L'arc u part de x_i et arrive à x_j. Graphiquement, l'arc u se représente de la manière suivante (diagramme sagittal) :

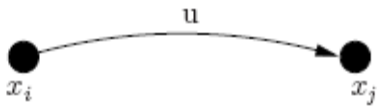


FIG. 1.4 : Arc $u = (x_i, x_j)$

Un arc (x_i, x_i) est appelé une boucle.

FIG. 1.5 : Boucle

Un p-graphe est un graphe dans lequel il n'existe jamais plus de p arcs de la forme (i, j) entre deux sommets quelconques.

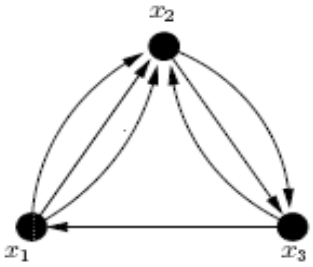


FIG. 1.6 : 3-graphe

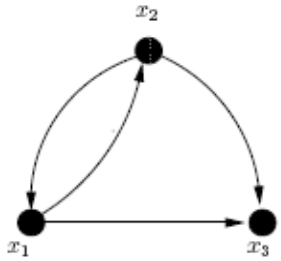


FIG. 1.7 : 1-graphe =graphe

Applications multivoques

x_j est successeur de x_i si $(x_i, x_j) \in U$; l'ensemble des successeurs de x_i est noté $\Gamma(x_i)$.

x_j est prédécesseur de x_i si $(x_j, x_i) \in U$; l'ensemble des prédécesseurs de x_i est noté $\Gamma^{-1}(x_i)$.

L'application Γ qui, à tout élément de X, fait correspondre une partie de X est appelée une application multivoque.

Pour un 1-graphe, G peut être parfaitement déterminé par (X, Γ),

Exemple 1.1.

$X = \{x_1, x_2, x_3, x_4, x_5\}$;

$\Gamma(x_1) = \{x_2\}$; $\Gamma(x_2) = \{x_1, x_3\}$; $\Gamma(x_3) = \{x_4, x_5\}$; $\Gamma(x_4) = \{x_5\}$; $\Gamma(x_5) = \{x_1, x_5\}$.

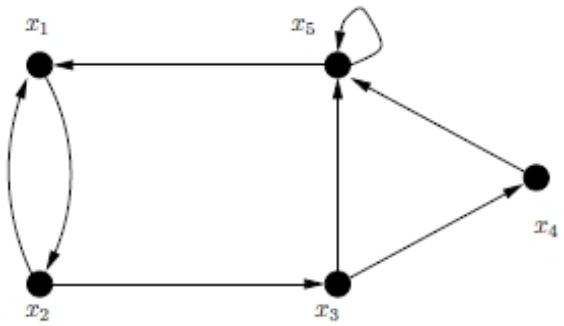


FIG. 1.8 : Graphe déterminé par (X, Γ)

Remarque 1.1.

on peut également définir une fonction ω telle que $\omega^+(x_i)$ représente l'ensemble des arcs sortant de x_i et réciproquement pour $\omega^-(x_i)$ et les arcs entrants.

1.2.3 Concepts non orientés

Lors de l'étude de certaines propriétés, il arrive que l'orientation des arcs ne joue aucun rôle. On s'intéresse simplement à l'existence d'arc(s) entre deux sommets (sans en préciser l'ordre). Un arc sans orientation est appelé arête. U est constitué non pas de couples, mais de paires de sommets non ordonnés. Pour une arête (x_i, x_j), on dit que u est incidente aux sommets x_i et x_j.

Remarque 1.2.

Dans le cas non-orienté, au lieu de noter $G = (X, U)$ et $u = (x_i, x_j)$, on préfère souvent $G = (X, E)$ et $e = [x_i, x_j]$.

Un multigraphe $G = (X, E)$ est un graphe pour lequel il peut exister plusieurs arête entre deux sommets.

Un graphe $G = (X, E)$ est simple :

1. s'il n'est pas un multigraphe.
2. s'il n'existe pas de boucles.

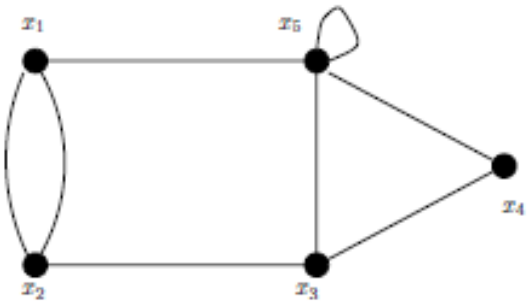


FIG. 1.9 : Graphe non orienté associé au graphe orienté de la fig 1.8

1.2.4 Principales définitions

– *Adjacence*

Deux sommets sont adjacents (ou voisins) s'ils sont joints par un arc.

Deux arcs sont adjacents s'ils ont au moins une extrémité commune.

– *Degrés*

- Le demi-degré extérieur de x_i, $d^+(x_i)$, est le nombre d'arcs ayant x_i comme extrémité initiale : $d^+(x_i) = |\omega^+(x_i)|$.

- Le demi-degré intérieur de x_i, $d^-(x_i)$, est le nombre d'arcs ayant x_i comme extrémité finale : $d^-(x_i) = |\omega^-(x_i)|$.

- Le degré de x_i est $d(x_i) = d^+(x_i) + d^-(x_i)$.

Remarque 1.3.

Dans le cas d'un 1-graphe, on peut tout aussi bien définir le degré d'un sommet à l'aide de l'application multivoque Γ puisque $|\omega^+(x_i)| = |\Gamma(i)|$ et $|\omega^-(x_i)| = |\Gamma^{-1}(i)|$.

Exemple 1.2.

$d^+(x_2) = 2$; $d^-(x_2) = 1$; $d(x_2) = 3$.

$d^+(x_5) = 2$; $d^-(x_5) = 3$; $d(x_5) = 5$.

$-$ *Graphe complémentaire*

$G = (X, U)$ et $G' = (X, U')$.

$(x_i, x_j) \in U \Rightarrow (x_i, x_j) \notin U'$ *et* $(x_i, x_j) \notin U \Rightarrow (x_i, x_j) \in U'$.

G' est le graphe complémentaire de G.

$-$ *Graphe partiel*

$G = (X, U)$ et $U_p \subset U$. $G_p = (X, U_p)$ est un graphe partiel de G. (On peut ainsi obtenir des sommets isolés...)

$-$ *Sous graphe*

$G = (X, U)$ et $X_s \subset X$. $G_s = (X_s, V)$ est un sous-graphe de G, où Vest la restriction de la fonction caractéristique de U à X_s.

$V = \{(x, y)/(x, y) \in U \cap X_s \times X_s\}. \forall x_i \in X_s$, $\Gamma_s(x_i) = \Gamma(x_i) \wedge X_s$.

$-$ *Sous graphe partiel*

Combine les deux définitions précédentes.

Exemple 1.3.

- Réseau routier que les autoroutes : graphe partie

- que la région Midi-Pyrénées : sous-graphe

- que les autoroutes de Midi-Pyrénées : sous-graphe partiel

$-$ *Graphe réflexif* : $\forall\ x_i \in X, (x_i, x_i) \in U$.

$-$ *Graphe symétrique* : $\forall\ x_i,\ x_j \in X, (x_i, x_j) \in U \Rightarrow (x_j, x_i) \in U$.

$-$ *Graphe antisymétrique* : $\forall\ x_i, x_j \in X, (x_i, x_j) \in U$ et $(x_j, x_i) \in U \Rightarrow x_i = x_j$.

$-$ *Graphe transitif* : $\forall\ x_i,\ x_j \in X, (x_i, x_j) \in U, (x_j, x_k) \in U \Rightarrow (x_i, x_k) \in U$.

$-$ *Graphe complet* : $\forall\ x_i,\ x_j \in X, (x_i, x_j) \notin U \Rightarrow (x_j, x_i) \in U$.

Exemple 1.4.

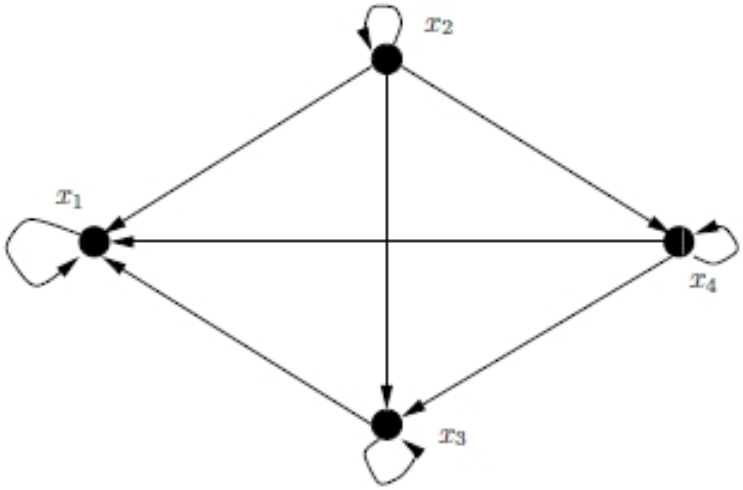


FIG. 1.10 : Graphe réflexif, antisymétrique, transitif et complet

1.2.5 Notion de complexité des algorithmes

Les problèmes de graphe se rattachent à la grande classe des problèmes d'optimisation combinatoire. Tous ces problèmes se répartissent en deux catégories : ceux qui sont résolus optimalement par des algorithmes efficaces (rapides), et ceux dont la résolution peut prendre un temps exponentiel sur les grands cas. On parle respectivement d'algorithmes polynomiaux et exponentiels.

Pour évaluer et classer les divers algorithmes disponibles pour un problème de graphe, il nous faut utiliser une mesure de performance indépendante du langage et de l'ordinateur utilisés. Ceci est obtenu par la notion de complexité d'un algorithme, qui consiste à mettre en évidence les possibilités et les limites théoriques du processus calculatoire, en évaluant le nombre d'opérations caractéristiques de l'algorithme dans le pire des cas. Elle est notée O(e.g., $O(n^2)$ pour une fonction qui augmente dans le carré de la taille des données). On rencontre aussi la notation Θ(e.g., $\Theta(n^2)$) qui donne une borne asymptotique par excès et par défaut (alors que O ne donne que la borne asymptotique par excès).

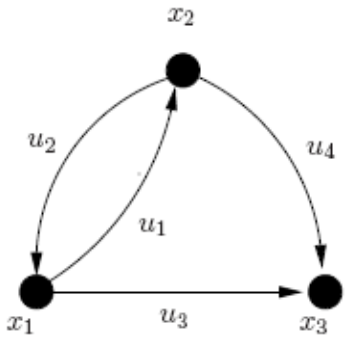
1.3 Représentations d'un graphe

Un certain nombre de représentations existent pour décrire un graphe. On distingue principalement la représentation par matrice d'adjacence, par matrice d'incidence sommets-arcs (ou sommets-arêtes dans le cas non orienté) et par listes d'adjacence.

1.3.1 Matrice d'adjacence

On considère un 1-graphe. La matrice d'adjacence fait correspondre les sommets origine des arcs (placés en ligne dans la matrice) aux sommets destination (placés en colonne). Dans le formalisme matrice booléenne, l'existence d'un arc (x_i, x_j) se traduit par la présence d'un 1 à l'intersection de x_i la ligne et de la colonne x_j, l'absence d'arc par la présence d'un 0 (dans un formalisme dit matrice aux arcs les éléments représentent le nom de l'arc).

Exemple 1.5.



origine ↑	x_1	x_2	x_3	← destination
x_1	0	1	1	
x_2	1	0	1	
x_3	0	0	0	

FIG. 1.11 : 1-graphe

Place mémoire utilisée : n^2 pour un graphe d'ordre $|X| = n$ (i.e., n sommets).

Remarque 1.4.

Pour des graphes numérisés (ou values ou pondérés), remplacer « 1 » par la valeur numérique de l'arc.

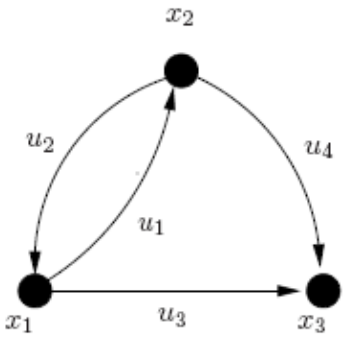
1.3.2 Matrice d'incidence sommets-arcs

Ligne $\longleftrightarrow$ sommet

colonne $\longleftrightarrow$ arc

Si $u = (i, j) \in U$, on trouve dans la colonne u : $a_{iu} = 1$: $a_{ju} = -1$; tous les autres termes sont nuls.

Exemple 1.6.



	u_1	u_2	u_3	u_4
x_1	1	-1	1	0
x_2	-1	1	0	1
x_3	0	0	-1	-1

FIG. 1.12 : 1-graphe

Place mémoire utilisée : $n \times m$.

Remarque 1.5.

La somme de chaque colonne est égale à 0 (un arc a une origine et une destination) ; la matrice est totalement unimodulaire, i.e., toutes les sous-matrices carrées extraites de la matrice ont pour déterminant $+1$, -1 ou 0.

1.3.3 Listes d'adjacence

Pour un 1-graphe, l'avantage de la représentation par listes d'adjacence (grâce à l'application multivoque Γ), par rapport à celle par matrice d'adjacence, est le gain obtenu en place mémoire ; ce type de représentation est donc mieux adapté pour une implémentation. Le but est de représenter chaque arc par son extrémité finale, son extrémité initiale étant définie implicitement. Tous les arcs émanant d'un même sommet sont liés entre eux dans une liste. A chaque arc sont donc associés le nœud destination et le pointeur au

prochain sommet dans la liste.

On crée deux tableaux LP (tête de listes) de dimension $(n+1)$ et LS (liste de successeurs) de dimension m (cas orienté) ou $2m$ (cas non orienté). Pour tout i, la liste des successeurs de i est dans le tableau LS à partir de la case numéro $LP(i)$.

1. On construit LS par $\Gamma(1), \Gamma(2), \ldots, \Gamma(n)$.
2. On construit LP qui donne pour tout sommet l'indice dans LS où commencent ses successeurs.
3. Pour un sommet $i \rightarrow 1er$ suivant : $LS(LP(i))$; $2ème$ suivant : $LS(LP(i)+1)$. L'ensemble des informations relatives au sommet i est contenue entre les cases $LP(i)$ et $LP(i+1)-1$ du tableau LS.
4. Si un sommet i n'a pas de successeur, on pose $LP(i) = LP(i+1)$ (liste vide coincée entre les successeurs de $i-1$ et ceux de $i+1$).

Pour éviter des tests pour le cas particulier $i = n$ (le sommet $i+1$ n'existant pas), on « ferme » par convention la dernière liste en posant $LP(n+1) = m+1$.

Exemple 1.7.

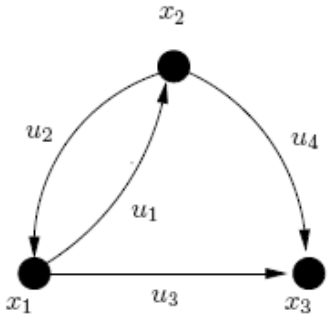


FIG. 1.13 : 1-graphe

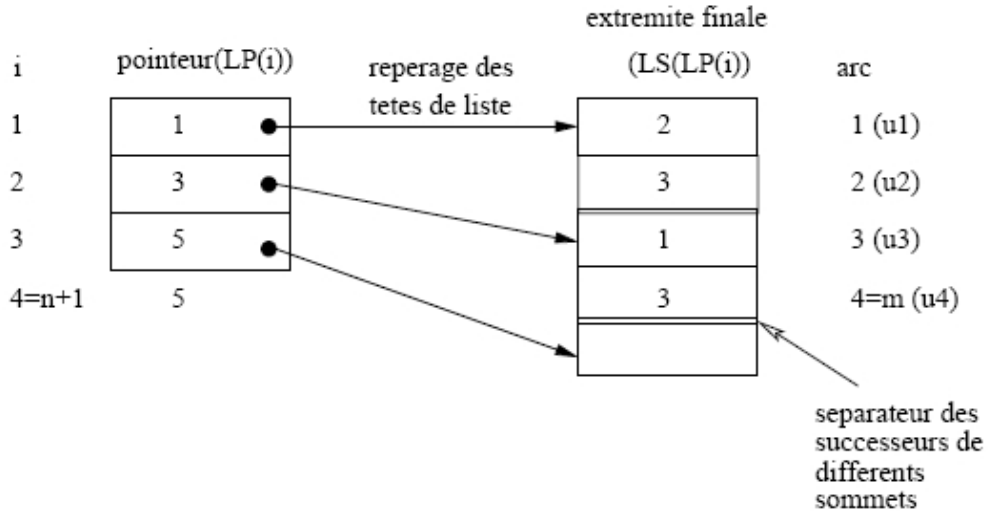


FIG. 1.14 : Liste d'adjacence

Place mémoire utilisée : $(n+1)+m$.

1.4 Connexité dans les graphes

1.4.1 Chaîne – Cycle

Une chaîne est une séquence d'arcs telle que chaque arc ait une extrémité commune avec le suivant.

Un cycle est une chaîne qui contient au moins une arête, telle que toutes les arêtes de la séquence sont différentes et dont les extrémités coïncident.

Exemple 1.8.

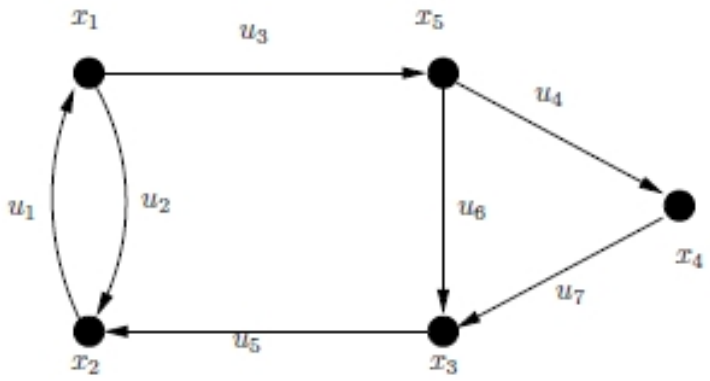


FIG. 1.15 : $< u_2, u_5, u_6, u_4 >$ est une chaîne de x_1à x_4. $< u_4, u_7, u_6 >$ est un cycle.

1.4.2 Chemin – Circuit

Ce sont les mêmes définitions que les précédentes mais en considérant des concepts orientés.

Racine : Soit $G = (E, \vec{\Gamma})$, $x \in E$. On dit que x est une racine de G si $\forall y \in E \backslash \{x\}$, il existe un chemin de x à y.

Exemple 1.9. [Fig. 1.15]

$< u_1, u_3, u_4, u_7 >$ est un chemin de x_2 à x_3.

$< u_1, u_3, u_6, u_5 >$ est un circuit.

Le sous-ensemble de sommets atteignables à partir d'un sommet donné, grâce à des chemins, est appelé fermeture transitive de ce sommet.

Le terme de parcours regroupe les chemins, les chaînes, les circuits et les cycles.

Un parcours est :

Elémentaire : si tous les sommets qui le composent sont tous distincts.

Simple : si tous les arcs qui le composent sont tous distincts.

Hamiltonien : passe une fois et une seule par chaque sommet du graphe.

Eulérien : passe une fois et une seule par chaque arc du graphe.

Préhamiltonien : passe au moins une fois par chaque sommet du graphe.

Préeulérien ou chinois : passe au moins une fois par chaque arc du graphe.

Remarque 1.6.

Le problème du voyageur de commerce est voisin du problème hamiltonien. Il consiste à trouver un circuit hamiltonien de coût minimal dans un graphe value.

1.4.3 Connexité

Un graphe $G = (X, U)$ est connexe si $\forall\ i,\ j \in X$, il existe une chaîne entre i et j.

On appelle composante connexe le sous-ensemble de sommets tels qu'il existe une chaîne entre deux sommets quelconques.

Un graphe est connexe s'il comporte une composante connexe maximale et une seule.

Chaque composante connexe est un graphe connexe.

Exemple 1.10.

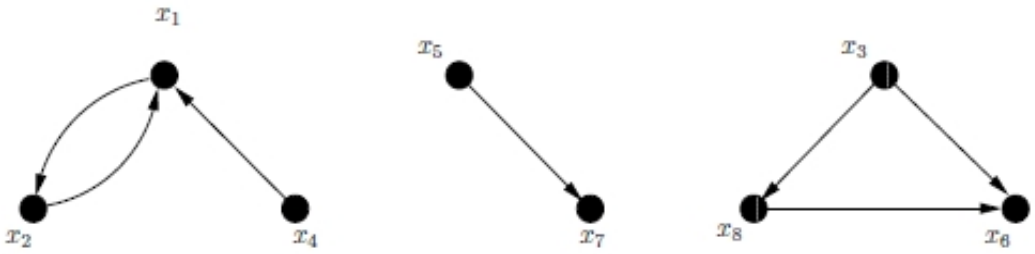


FIG. 1.16 : Graphe ayant trois composantes connexes.

Application : Vérification de la connexité aux réseaux téléphoniques ou électriques.

1.4.4 Forte connexité

Un graphe $G = (X, U)$ est fortement connexe si $\forall\ i,\ j \in X$, il existe un chemin entre i et j.

Une composante fortement connexe (cfc) est un sous-ensemble de sommets tel qu'il existe un chemin entre deux sommets quelconques.

Une cfc maximale (cfcm) est un ensemble maximal de cfc. Les différentes cfcm définissent une partition de X.

Un graphe est fortement connexe s'il comporte une seule cfcm.

Chapitre 2

Etude algorithmique de quelques problèmes

2.1 Le problème du plus court chemin

Les problèmes de cheminement dans les graphes (en particulier la recherche d'un plus court chemin) comptent parmi les problèmes les plus anciens de la théorie des graphes et les plus importants par leurs applications. On peut citer entre autres :

– les problèmes de tournées,

– certains problèmes d'investissement et de gestion de stocks,

– les problèmes de programmation dynamique à états discrets et temps discret,

– les problèmes d'optimisation de réseaux (routiers, télécommunications),

– certaines méthodes de traitement numérique du signal, de codage et de décodage de l'information,

– les problèmes de labyrinthe et de récréations mathématiques,

On considère dans ce chapitre des graphes orientés.

2.1.1 Définitions

Définition 2.1.

On désigne par $\vec{\Gamma}$ l'ensemble des arcs du graphe (E, Γ). L'ensemble des arcs est un sous-ensemble du produit cartésien $E \times E = \{(x, y)/x \in E, y \in E\}$, autrement dit, $\vec{\Gamma} \in P(E \times E)$. On peut définir formellement $\vec{\Gamma}$ par :

$$\vec{\Gamma} = \{(x, y) \in E \times E / y \in \Gamma(x)\}$$

On peut représenter G par (E, Γ) ou par $(E, \vec{\Gamma})$, les données de ces deux informations étant équivalentes. On appellera également graphe le couple $(E, \vec{\Gamma})$.

Définition 2.2.

Nous allons travailler sur un graphe sans boucle $G = (E, \vec{\Gamma})$ et une application l de $\vec{\Gamma}$ dans R. L'application l associe à chaque arc u un réel $l(u)$ appelé longueur de l'arc u.

Le triplet $R = (E, \vec{\Gamma}, l)$ est appelé un réseau.

Par simplicité, si $u = (x, y)$, on notera $l(x, y)$ la longueur de u (plutôt que $l(x, y)$, correct mais lourd).

Soit $\sigma = (x_0, x_1, ..., x_n)$ un chemin dans G. On définit la longueur de σ (dans R) comme la somme des longueurs des arcs de σ, autrement dit :

$$l(\sigma) = \sum_{0 \leq i \leq n-1} l(x_i, x_{i+1})$$

2.1.2 Plus court chemin

Définition 2.3.

Soient x et y deux sommets de E. Un chemin σ de x à y dans R est un plus court chemin de x à y si pour tout chemin σ' de x à y, on a $l(\sigma) \leq l(\sigma')$

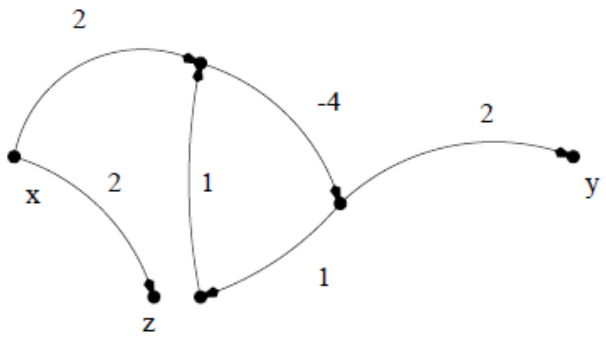
Dans la recherche d'un plus court chemin de x à y, trois cas peuvent se présenter :

– Il n'existe aucun chemin de x à y (par exemple, si x et y appartiennent à deux composantes connexes différentes de G).

– Il existe des chemins de x à y mais pas de plus court chemin.

– Il existe un plus court chemin de x à y.

Exemple 2.1.



Il existe des chemins (combien ?) de x à y, mais il n'existe pas de plus court chemin de x à y. Il existe un plus court chemin de x à z mais pas de chemin de z à x.

Remarque 2.2.

Un plus court chemin dans $R = (E, \overrightarrow{\Gamma}, l)$ est un plus long chemin dans $R' = (E, \overrightarrow{\Gamma}, -l)$, et réciproquement.

Circuit absorbant : Un circuit de longueur négative est appelé circuit absorbant.

Dans l'exemple précédent, il existe un circuit absorbant.

Remarque 2.3.

Si une composante fortement connexe présente un circuit absorbant, alors il n'existe pas de plus court chemin entre deux sommets quelconques de cette composante.

Proposition 2.1.[9]

Soit $R = (E, \overrightarrow{\Gamma}, l)$ un réseau. une condition nécessaire et suffisante pour qu'il existe un plus court chemin entre s et tout sommet de $E \setminus \{s\}$ est :

1. s est une racine de $(E, \overrightarrow{\Gamma})$, et

2. il n'y a pas de circuit absorbant dans R

2.1.3 Problématique du plus court chemin

Proposition 2.2.[9]

Soit R un réseau, soit $c = (x_0, ..., x_k)$ un plus court chemin de x_0 à x_k et soit $c' = (x_i, ..., x_j)$, $0 \leq i \leq j \leq k$ un sous-chemin extrait de c, alors c' est un plus court chemin de x_i à x_j.

Remarque 2.4.

Il s'agit ici de la propriété très importante d'optimalité des sous-chemins.

Graphe des plus courts chemins

Soit $R = (E, \vec{\Gamma}, l)$ un réseau sans circuit absorbant, soit $i \in E$ On définit l'application $\pi_i : E \to R \cup \{\infty\}$ par :

$$\pi_i(x) = \begin{cases} \text{la longueur d'un plus court chemin de } i \text{ à } x \text{ s'il en existe} \\ \infty \text{ si non} \end{cases}$$

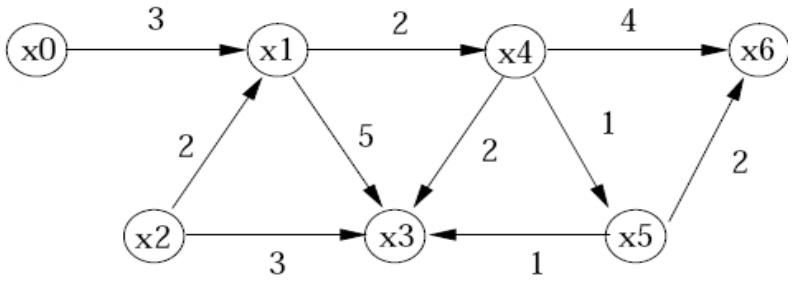
Soit $(E, \vec{\Gamma})$ le graphe défini par : $\vec{\Gamma'} = \left\{(x, y) \in \vec{\Gamma} \backslash l(x, y) = \pi_i(y) - \pi_i(x)\right\}$

Le graphe $(E, \vec{\Gamma})$ est appelé graphe des plus courts chemins relativement au sommet i. L'ensemble $\vec{\Gamma'}$ est constitué des arcs faisant partie d'un plus court chemin de i à un sommet de E.

Proposition 2.3.[9]

Un plus court chemin de i à x dans R est un chemin de i à x dans $(E, \vec{\Gamma'})$.

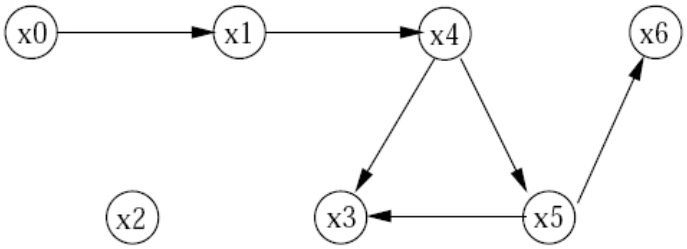
Exemple 2.2.



On peut construire l'application π_{x_0}

x	$\pi_{x_0}(x)$
x_0	0
x_1	3
x_2	∞
x_3	7
x_4	5
x_5	6
x_6	8

Le graphe des plus courts chemins est alors :



2.1.4 D'un sommet à tous les autres

Réseaux à longueurs quelconques (Bellman)

Nous allons étudier l'algorithme de Bellman pour calculer les longueurs de plus courts chemins dans un réseau à longueurs quelconques. Cet algorithme calcule la longueur des plus courts chemins d'un sommet initial i à tout autre sommet x du graphe.

But : soit $R = (E, \overrightarrow{\Gamma}, l)$, $i \in E$, calculer π_i.

L'idée est qu'un chemin de i à x dans R passe nécessairement par un sommet $y^* \in \Gamma^{-1}(x)$.

La longueur d'un plus court chemin de i à x passant par y^* est :

$$\pi_i(x) = \pi_i(y^*) + l(y^*, x).$$

Et donc, on a la propriété suivante, dite propriété de Bellman.

Proposition 2.4.[9]

$$\pi_i(x) = \min_{y \in \Gamma^{-1}(x)} \{\pi_i(y) + l(y,x)\}$$

Algorithme de Bellman (Données : E, Γ^{-1}, l, $i \in E$, Résultats : π, CIRCABS)

Pour alléger les notations, on notera dans la suite $\pi = \pi_i$, sauf en cas de nécessité.

1. CIRCABS = FAUX ; $\pi^0(i) = 0$; $\pi^1(i) = 0$; $k = 1$
2. for all $x \in E \setminus \{i\}$ do
3. $\quad \pi^0(x) = \infty$
4. $\quad \pi^1(x) = \infty$
5. end for
6. for all $x \in E$ tel que $i \in \Gamma^{-1}(x)$ do
7. $\quad \pi^1(x) = l(i,x)$
8. end for
9. while $k < n+1$ et $\exists\ x \in E$ tel que $\pi^k(x) \neq \pi^{k-1}(x)$ do
10. $\quad k = k+1$
11. $\quad$ for all $x \in E$ do
12. $\quad\quad \pi^k(x) = \min\left[\pi^{k-1}(x),\ \pi^{k-1}(y) + l(y,x);\ y \in \Gamma^{-1}(x)\right]$
13. $\quad$ end for
14. end while
15. if $k = n+1$ then
16. $\quad$ CIRCABS = VRAI
17. end if
18. $\pi = \pi_k$

Cet algorithme renvoie également dans la variable CIRCABS un booléen indiquant la présence d'un circuit absorbant.

Complexité : $O(nm)$.

Justification

L'idée est que la longueur d'un plus court chemin du sommet i à n'importe quel sommet x est imposée par :

$$\pi(x) = \min\{\pi(y) + l(y, x)\}$$

Cette formule locale est appliquée jusqu'à convergence (ou jusqu'à n itérations, dans ce cas il y a présence d'un circuit absorbant).

On appelle $k-$chemin un chemin possédant au plus k arcs. Dans cet algorithme, $\pi^k(x)$ représente la longueur d'un plus court k-chemin de i à x (s'il en existe ∞ sinon). Pour prouver cette proposition, nous allons procéder par récurrence.

La proposition est trivialement vérifiée pour $k = 0$ et $k = 1$. Supposons qu'elle est vérifiée jusqu'à l'étape $k-1$ et plaçons-nous à l'étape k. Soit C_k l'ensemble des sommets y tels qu'il existe un $k-$chemin de i à y.

- Si $x \notin C_k, \forall\ y \in \Gamma^{-1}(x)$, $y \in C_{k-1}$. Donc d'après l'hypothèse de récurrence, $\forall\ y \in \Gamma^{-1}(x)$, $\pi^{k-1}(x) = \infty$, d'où $\pi^k(x) = \infty$.

- Si $x \in C_k$ et $x \neq i$ alors $\exists\ y \in \Gamma^{-1}(x)$ tel que $y \in C_{k-1}$ donc $\pi^{k-1}(y) < \infty$. Tout k-chemin de i à x passe par un $y \in \Gamma^{-1}(x)$

Un plus court k-chemin de i à x est composé d'un plus court (k - 1)-chemin de i à un sommet $y \in \Gamma^{-1}(x)$ et de l'arc (y, x). Il a pour longueur $\pi^{k-1}(y) + l(y, x)$. La longueur d'un plus court k-chemin de i à x est donc :

$$\min_{y \in \Gamma^{-1}(x)} \pi_i(y) + l(y, x)$$

De plus, la longueur d'un plus court k-chemin de i à x ne peut être supérieure à celle d'un plus court (k-1)-chemin de i à x (puisqu'un (k-1)-chemin est aussi un k-chemin), d'où l'instruction de la ligne 12.

On peut facilement voir que s'il existe un plus court chemin σ de i à x alors il existe

un plus court chemin de i à x qui est un chemin élémentaire (en effet si σ comporte un circuit de longueur positive ou nulle, alors en ôtant ce circuit de σ on obtient un chemin de longueur inférieure ou égale à $l(\sigma)$). Or un tel chemin possède au plus $n-1$ arcs (avec $n = |E|$) Donc s'il n'existe pas de circuit absorbant, on $\forall\ k \geq n\ \ \pi = \pi_i$,

Par conséquent, si $\pi^n \neq \pi^{n-1}$, alors le graphe présente un circuit absorbant.

Réseaux à longueurs positives (Dijkstra)

L'algorithme de Dijkstra permet de trouver les longueurs des plus courts chemins d'un sommet donné i à tous les autres sommets dans un réseau (E, Γ, l) à longueurs positives.

Dans un tel réseau on a l'équivalence suivante :

Il existe un chemin de x à $y \Longleftrightarrow$ il existe un plus court chemin de x à y

pour tout couple de sommets (x, y). Cette équivalence est due au fait qu'il n'existe pas de circuit absorbant.

Le principe de l'algorithme est le suivant :

Soit $R = (E, \vec{\Gamma}, l)$ avec pour tout $u \in \vec{\Gamma}$, $l(u) \geq 0$.

Soit $i \in E$ sommet initial. On désigne par S l'ensemble des sommets x de E pour lesquels la longueur d'un plus court chemin de i à x a été calculée.

Initalement $S = \{i\}$, $\pi(i) = 0$.

On appelle S-chemin un chemin partant de i et ayant tous ses sommets dans S sauf le dernier.

Soit $Y = E \backslash S$. Pour tout sommet y dans Y, on désigne par $\pi\prime(y)$

la longueur d'un plus court S-chemin de i à y. On sélectionne y^* dans Y tel que :

$\pi\prime(y*) = min\{\pi\prime(y),\ y \in Y\}$. On est alors assuré que $\pi\prime(y^*) = \pi(y^*)$.

Algorithme de Dijkstra (Données : E, Γ, l, i Résultats : π, S)

1. $S = \{i\}$, $\pi(i) = 0$, $k = 1$, $x_1 = i$
2. for all $x \in E \setminus \{i\}$ do
3. $\quad \pi(x) = \infty$

4. end for

5. while $k < n$ et $\pi(x_k) < \infty$ do

6. for all $y \in \Gamma(x_k)$ tel que $y \notin S$ do

7. $\pi(y) = \min[\pi(y),\ \pi(x_k) + l(x_k, y)]$

8. end for

9. Extraire un sommet $x \notin S$ tel que $\pi(x) = \min\{\pi(y),\ y \notin S\}$

10. $k = k + 1$, $x_k = x$, $S = S \cup \{x_k\}$

11. end while.

Complexité : La complexité de l'algorithme de Dijsktra (sous cette forme) est en $O(n^2)$.

2.1.5 Entre tous les couples de sommets (algorithme matriciel)

On va ainsi calculer un distancier $n \times n$. Si tous les arcs sont tous de longueur positive ou nulle ($l(u) \geq 0$), on peut appliquer n fois l'algorithme de Dijkstra-Moore pour chaque sommet i. Si le graphe comporte des arcs de longueur strictement négative, on peut appliquer n fois l'algorithme de Bellman-Ford.

L'algorithme de Floyd constitue une autre approche qui peut être avantageuse principalement par rapport à la seconde solution, qui nécessite un temps d'exécution en $O(n^4)$ pour des graphes denses i.e., des graphes tels que $m = n^2$.

Contrairement aux algorithmes à origine unique qui supposent que le graphe est représenté par une liste d'adjacence, l'algorithme de Floyd-Warshall (algorithme de programmation dynamique) utilise une représentation par matrice d'adjacence.

Soient les matrices : $L = [l_{ij}]$; $P = [p_{ij}]$

$l_{ij} = l(i, j)$ si $(i, j) \in U$

$= \infty$ si non

A l'initialisation : $l_{ij} = 0$ $p_{ij} = 0$ si $l_{ij} = \infty$

$= i$ sinon

En fin d'algorithme :

l_{ij} = longueur du plus court chemin entre i et j

p_{ij} = prédécesseur de j sur le plus court chemin de i à j.

L'algorithme consiste en trois boucles imbriquées sur i, j, k de 1 à n et on calcule :

$$l_{ij} = \min(l_{ij}, l_{ik} + l_{kj})$$

Floyd-Warshall(L,P)

1. pour $k = 1$ à n
2. pour $i = 1$ à n sauf k
3. si $l_{ik} + l_{ki} < 0$ fin % présence d'un circuit négatif
4. si $l_{ik} = \infty$ alors
5. pour $j = 1$ à n sauf i
6. $\gamma = l_{ik}$ % pour éviter le calcul d'adresse de la variable indicée à chaque itération de j
7. si $\gamma + l_{kj} < l_{ij}$
8. $l_{ij} \leftarrow \gamma + l_{kj}$
9. $p_{ij} \leftarrow p_{kj}$

Complexité : $O(n^3)$.

Exemple 2.3.

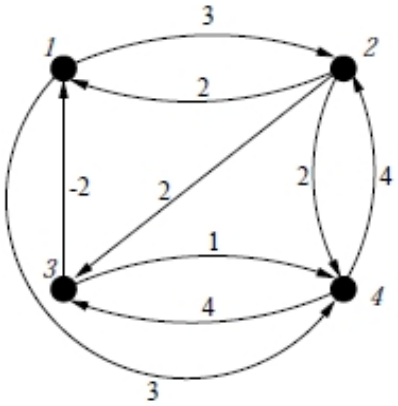


Fig. 2.3 : Procédure de Floyd-Warshall

$$L^{(0)} = \begin{pmatrix} 0 & 3 & \infty & 3 \\ 2 & 0 & 2 & 2 \\ -2 & \infty & 0 & 1 \\ \infty & 4 & 4 & 0 \end{pmatrix}, \qquad P^{(0)} = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 0 & 3 & 3 \\ 0 & 4 & 4 & 4 \end{pmatrix}$$

$$L^{(1)} = \begin{pmatrix} 0 & 3 & \infty & 3 \\ 2 & 0 & 2 & 2 \\ -2 & 1 & 0 & 1 \\ \infty & 4 & 4 & 0 \end{pmatrix}, \qquad P^{(1)} = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 1 & 3 & 3 \\ 0 & 4 & 4 & 4 \end{pmatrix}$$

$$L^{(2)} = \begin{pmatrix} 0 & 3 & 5 & 3 \\ 2 & 0 & 2 & 2 \\ -2 & 1 & 0 & 1 \\ 6 & 4 & 4 & 0 \end{pmatrix}, \qquad P^{(2)} = \begin{pmatrix} 1 & 1 & 2 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 1 & 3 & 3 \\ 2 & 4 & 4 & 4 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & 5 & 3 \\ 0 & 0 & 2 & 2 \\ -2 & 1 & 0 & 1 \\ 2 & 4 & 4 & 0 \end{pmatrix}, \qquad P^{(3)} = \begin{pmatrix} 1 & 1 & 2 & 1 \\ 3 & 2 & 2 & 2 \\ 3 & 1 & 3 & 3 \\ 3 & 4 & 4 & 4 \end{pmatrix}$$

$$L^{(4)} = \begin{pmatrix} 0 & 3 & 5 & 3 \\ 0 & 0 & 2 & 2 \\ -2 & 1 & 0 & 1 \\ 2 & 4 & 4 & 0 \end{pmatrix}, \qquad P^{(4)} = \begin{pmatrix} 1 & 1 & 2 & 1 \\ 3 & 2 & 2 & 2 \\ 3 & 1 & 3 & 3 \\ 3 & 4 & 4 & 4 \end{pmatrix}$$

2.2 Le problème du l'arbre couvrant minimal

2.2.1 Définitions

Définition 2.4.

Un arbre est un graphe connexe sans cycles.

Un graphe sans cycle qui n'est pas connexe est appelé une forêt (chaque composante connexe est un arbre).

Par définition même, un arbre est donc un graphe simple. On constate également que $T = (X, T)$ est un arbre si et seulement s'il existe une chaîne et une seule entre deux sommets quelconques.

Etant donné un graphe quelconque $G = (X, A)$ un arbre de G est un graphe partiel connexe et sans cycles. Si ce graphe partiel inclut tous les sommets du graphe G, l'arbre est appelé arbre maximum ou arbre couvrant.

-Une forêt d est un graphe partiel sans cycle de G (non nécessairement connexe).

-Une forêt maximale de G est une forêt de G maximale pour l'inclusion (l'ajout d'une seule arête supplémentaire du graphe à cette forêt crée un cycle).

Définition 2.5.

Un graphe G est une *arborescence* s'il existe un sommet R appelé racine de G tel que, pour tout sommet S de G, il existe un chemin et un seul de R vers S.

2.2.2 Arbres couvrants de poids minimum

Considérons le problème qui consiste à relier n villes par un réseau câblé de la manière la plus économique possible. On suppose connue la longueur $l_{ij} = l(a_{ij})$ la longueur de câble nécessaire pour relier les villes i et j. Le réseau doit évidemment être connexe et il ne doit pas admettre de cycles pour être de coût minimal ; c'est donc un arbre et ce doit être l'arbre maximum le plus économique.

Le problème à résoudre se pose donc dans les termes suivants :

Définition 2.6.

Soit un graphe non orienté G, connexe, pondéré par une fonction positive l attachée aux arêtes. Soit un arbre couvrant $T = (X, B)$ défini comme graphe partiel de G avec un ensemble d'arêtes B. Son poids (ou coût) total est :

$$l(T) = \sum_{a \in B} l(a)$$

On dit que T est un arbre couvrant de poids minimal de G si $l(T)$ est minimal parmi les poids de tous les arbres couvrants possibles de G.

On peut montrer que si toutes les arêtes sont de "poids" différents, l'arbre couvrant de poids minimal est unique. Plusieurs algorithmes ont été proposés pour résoudre ce problème. Les plus simples sont les algorithmes de Prim et de Kruskal.

Algorithme de Prim

L'algorithme de Prim consiste à bâtir progressivement un arbre à partir d'un sommet quelconque (arbitrairement le sommet numéro 1) et en y greffant, à chaque étape, l'arête de poids minimal parmi celles qui permettent de maintenir un graphe partiel qui soit un arbre. Si le graphe est connexe, le processus s'arrête avec un arbre couvrant. Sinon, il aboutit à un arbre couvrant pour une composante connexe ; on poursuit avec les autres composantes connexes pour obtenir une forêt couvrante.

Plus précisément, on construit progressivement, à partir du sommet numéro 1, un sous ensemble de sommet $S \subset X$ contenant $\{1\}$ et un sous-ensemble $T \subset A$ tel que le graphe partiel (S, T) soit un arbre de poids minimal du sous-graphe engendré par S. Pour cela, à chaque étape, on sélectionne dans le cocycle :

$$\omega(s) = \{(k, l) | (k, l) \in A,\ k \in S,\ l \in X \backslash S\}$$

L'arête de poids minimal, soit $a = (i, j)$. Les sous-ensembles S et T sont alors augmentés en leur ajoutant le sommet j et l'arête a respectivement :

$$S \leftarrow S \cup \{j\},\ T \leftarrow T \cup \{a\}.$$

La mise en œuvre de cet algorithme peut se faire aisément à l'aide d'une procédure de marquage de la façon suivante. On associe, à chaque sommet i, un nombre réel $\pi(i)$ appelé

marque du sommet i. A une étape quelconque, la marque $\pi(i)$ d'un sommet $i \in X\backslash S$ représente le poids de l'arêtes de poids minimal dans l'ensemble des arêtes joignant i à S.

D'autre part, on conserve dans un tableau α l'indice $\alpha(i)$ de l'arête ayant permis d'attribuer au sommet i, la marque $\pi(i)$. Ces informations permettent aisément d'obtenir, à l'étape courante, l'arête de poids minimal dans le cocycle $\omega(s)$: il suffit en effet de déterminer le sommet $i \in X\backslash S$ ayant une marque minimale et l'arêtes cherchée est $\alpha(i)$. Par ailleurs, lorsque le sous-ensemble S est augmenté du sommet i, les marques sont mises à jour en examinant toutes les arêtes issues de i et dont l'autre extrémité j ne fait pas encore partie de l'arbre ($j \in X\backslash S$) et en effectuant la substitution $\pi(j) \leftarrow min(\pi(j), lij)$.

Chaque fois qu'une marque est améliorée, le tableau α est mis à jour. Si le graphe est connexe, l'algorithme s'arrête lorsque $S = X$ (tous les sommets ont été intégrés à l'arbre). Les éléments du tableau α représentent alors les indices des arêtes constituant l'arbre couvrant de poids minimal.

Algorithme de Prim

Recherche d'un arbre couvrant de poids minimal.

(a) - Initialisations

$\pi(1) = 0$

$\pi(i) = \infty, \qquad \forall\ i \in \{2, 3, ..., n\}$

$\alpha(i) = \infty, \qquad \forall\ i \in \{1, 2, ..., n\}$

$S = \emptyset$

(b) - Sélectionner i tel que $\pi(i) = \underset{j \in X\backslash S}{min}(\pi(j))$

Si $\pi(i) = \infty$ ou S= X alors FIN

$S \leftarrow S \cup \{i\}$

(c) - Pour toutes les arêtes $a = (i, j)$ telles que $j \in X\backslash S$ faire

Si $l_{ij} < \pi(j)$ alors $\pi(j) = l_{ij}$, $\alpha(j) = a$

Retourner en (b).

2.3 Réseaux, réseaux de transport et problèmes de flots

2.3.1 Définitions

Définition 2.7.

Un graphe fortement connexe, sans boucle et ayant plus d'un sommet, est appelé un *réseau*.

On appelle *nœud* d'un réseau un sommet qui a plus de deux arcs incidents. Les autres sommets sont appelés *antinœuds*.

On appelle *branche* tout chemin pour lequel seuls les premiers et derniers sommets sont des nœuds.

Dans un graphe orienté G, un *flot* est l'affectation d'une valeur réelle à chaque arc de G, représentant une quantité transportée sur cet arc, de telle sorte que, en chaque sommet, la somme des flots entrants soit égale à la somme des flots sortants (loi de Kirchhoff : conservation des flux en chaque sommet).

Parmi les problèmes les plus classiques, on peut citer celui de la recherche d'un flot maximal. On se donne une capacité maximale sur chaque arc qui sera une borne supérieure du flot autorisé sur cet arc. Le problème du flot maximal consiste à déterminer un flot dont la valeur en un certain lieu est maximale. On peut, de plus, se donner un coût de transport d'une unité de flot sur chaque arc et chercher le flot maximal de coût minimal.

Définition 2.8.

On appelle *réseau de transport* un graphe orienté antisymétrique value $G = (X, A, C)$, sans boucle et dans lequel il existe :

– un sommet x_1 sans prédécesseur (c'est-à-dire $\Gamma^{-1}(x_1) = \emptyset$) nommé *entrée* ou *source* du réseau,

– un sommet x_n sans successeur (c'est-à-dire $\Gamma^{-1}(x_n) = \emptyset$) nommé *sortie* ou *puits* du réseau, et tel qu'au moins un chemin unisse x_1 à x_n dans G.

La fonction de pondération C est supposée positive et l'on nomme capacité de l'arc a le nombre $C(a)$.

Définition 2.9.

Si l'on désigne par A_x^- l'ensemble des arcs sortants du sommet x et A_x^+ l'ensemble des arcs entrants de ce même sommet x, on dit qu'une fonction $\varphi(a)$ définie sur A et à valeurs réelles est un flot pour le réseau de transport si :

– il est positif : $\varphi(a) \geq 0, \forall a \in A$,

– il vérifie la loi des nœuds de Kirchhoff :

$$\sum_{a \in A_x^-} \varphi(a) - \sum_{a \in A_x^+} \varphi(a) = 0, \qquad \forall x \neq x_1 \text{ et } x \neq x_n$$

– il ne dépasse pas la capacité des arcs : $\varphi(a) \leq c(a), \forall a \in A$

Si x n'est ni x_1 ni x_n, la quantité entrante en x doit être égale à la quantité sortante que nous désignons par $\varphi(x)$:

$$\varphi_x = \sum_{a \in A_x^-} \varphi(a) = \sum_{a \in A_x^+} \varphi(a)$$

Si φ est un flot sur un réseau de transport G, alors on a $\varphi_{x_1} = \varphi_{x_n}$; cette quantité s'appelle la *valeur du flot*.

La principale question qui se pose pour un réseau de transport donné est de déterminer un flot de valeur maximale ainsi que les flots le long de chaque arc. Il arrive fréquemment également que l'on doive considérer des réseaux avec des capacités localisées non seulement sur les arêtes mais également sur les sommets. C'est notamment le cas pour les réseaux téléphoniques pour lesquels la limite de capacité est autant due aux lignes qu'aux centraux. On peut ramener aisément ce problème au précédent ; il suffit de dédoubler chaque sommet en une entrée et une sortie liées par un arc ayant pour capacité celle qu'on attribuait précédemment au sommet.

Dans ce qui suit, les valeurs des flots et des capacités sont considérées comme entières.

Si l'on ne peut pas se ramener directement à cette situation, on approche les valeurs réelles par des rationnels, on réduit ces nombres au même dénominateur commun d et l'on choisit comme unité de référence $1/d$.

2.3.2 Recherche d'un flot complet

Définition 2.10.

Pour un flot φ dans un réseau de transport $G = (X, A, C)$, on dit qu'un arc est *saturé* si on a $\varphi(a) = C(a)$.

Le flot est dit *complet* si tout chemin allant de x_1 à x_n contient au moins un arc saturé.

Si l'on considère le graphe partiel engendré par les arcs non saturés par le flot et si le flot n'est pas complet, il existe nécessairement un chemin μ allant de l'entrée à la sortie. On peut alors définir un nouveau flot pour le réseau en augmentant de 1 le flot de chacun des arcs constituant le chemin μ ; la valeur du flot est alors également augmentée de 1. On peut donc progressivement augmenter la valeur d'un flot incomplet jusqu'à ce qu'il soit complet. En tenant compte des différences entre les capacités et la valeur du flot sur les arcs de μ, on peut connaître d'avance l'augmentation possible du flot. Cependant, le flot complet ainsi obtenu n'est pas, en général, le flot maximal.

Définition 2.11.

Soit un réseau de transport $G = (X, A, C)$ possédant un flot complet φ. On appelle graphe d'écart ou réseau résiduel, le réseau $G'(\varphi) = (X, A', C')$ tel que :

– si $a \in A$ et $\varphi(a) < C(a)$ alors $a \in A$ et $C'(a) = C(a) - \varphi(a)$

– si $a = (x, y) \in A$ et $\varphi(a) > 0$ alors $a^{-1} = (y, x) \in A'$ *et* $C'(a^{-1}) = \varphi(a)$

Le réseau résiduel indique le long de quels arcs on peut augmenter ou diminuer le flot.

L'intérêt du graphe d'écart apparaît dans le théorème suivant.

Théorème 3.1.[5]

Soit φ un flot de G (de x_1 à x_n) et $G'(\varphi)$ le réseau résiduel associé à φ. Une condition nécessaire et suffisante pour que le flot φ soit maximal est qu'il n'existe pas de chemin

de x_1 à x_n dans $G'(\varphi)$.

2.3.3 Recherche d'un flot maximal

A partir des résultats précédents, on peut maintenant proposer une méthode constructive de recherche d'un flot maximal.

Algorithme (Ford et Fulkerson)

Recherche d'un flot maximum.

(a) - Itération $k = 0$

Partir d'un flot initial φ^0 (compatible avec les contraintes de capacité),

par exemple $\varphi^0 = (0, 0, ..., 0)$.

(b) - A l'itération k, soit φ^k le flot courant.

Rechercher un chemin μ^k de x_1 à x_n dans le graphe d'écart $G'(\varphi^k)$.

S'il n'en existe pas, FIN : le flot φ^k est maximal.

(c) - Soit ε^k la capacité résiduelle du chemin μ^k

(minimum des capacités résiduelles des arcs du chemin).

Définir le flot φ^{k+1} par :

$$\begin{cases} \varphi_a^{k+1} = \varphi_a^k + \varepsilon^k & \text{si } a \in \mu \text{ et si } a \text{ est orientée dans le sens de } \mu \\ \varphi_a^{k+1} = \varphi_a^k - \varepsilon^k & \text{si } a \in \mu \text{ et si } a \text{ est orientée dans le sens contraire de } \mu \end{cases}$$

Faire $k \leftarrow k + 1$ et retourner en (b).

Par définition du graphe d'écart, les flots φ^k sont tous compatibles avec les capacités. Par ailleurs, ε^k est positif à chaque itération et par suite l'algorithme produit une séquence de flots compatibles de valeurs strictement croissantes.

A chaque étape, la méthode consiste donc à rechercher une chaîne joignant x_1 à x_n. Il est évident que le choix d'une telle chaîne n'est pas unique. Plusieurs travaux ont tenté de systématiser ce choix afin d'améliorer les performances moyennes de l'algorithme.

2.3.4 Exemple traité manuellement

On considère le graphe de la figure 2.4 (les nombres associés aux arcs représentent les capacités), pour lequel on cherche à déterminer un flot maximal entre le sommet 1 et le sommet 7.

Toutes les composantes du flot initial sont considérées nulles $\varphi^0 = (0, 0, ..., 0)$ (figure 2.5.a), le graphe d'écart $G'(\varphi^0)$ est représenté au figure 2.5.b. Le chemin choisi est tracé en pointillés, sa capacité résiduelle vaut $\varepsilon^1 = min(5, 4, 7) = 4$.

On augmente donc de quatre le flot sur les arcs composant le chemin (ils sont tous orientés positivement).

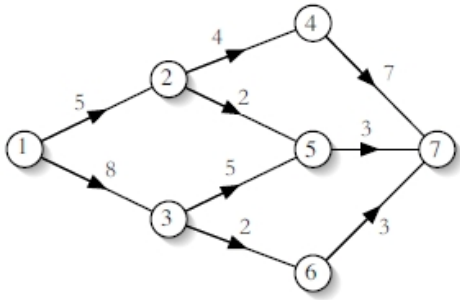


Fig. 2.4 : Graphe orienté élémentaire

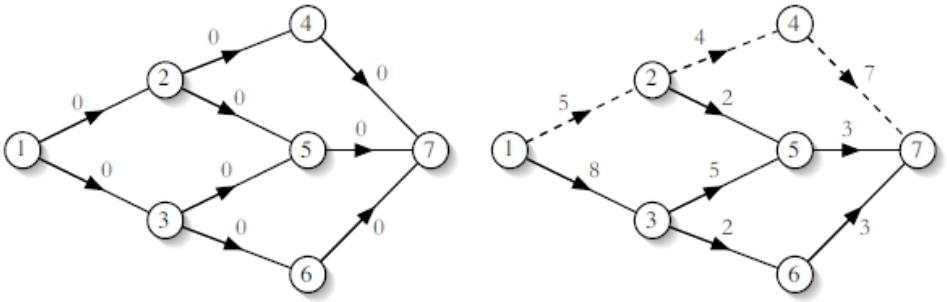


Fig. 2.5 : Flot initial et graphe d'écart

On obtient un nouveau flot φ_1, ainsi que son graphe d'écart $G(\varphi^1)$, tous deux représentés aux figures 2.6.a et 2.6.b.

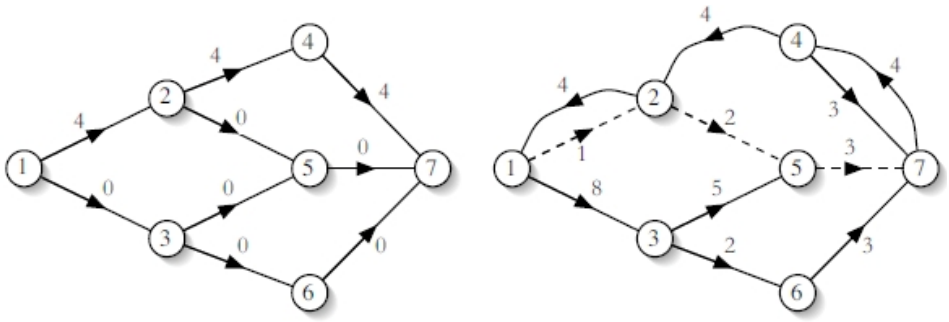


Fig. 2.6 : Flot à l'itération numéro 1 et graphe d'écart

Le nouveau chemin est toujours représenté en traits pointillés. Cette fois, sa capacité résiduelle vaut $\varepsilon^2 = min(1, 2, 3) = 1$.

Le nouveau flot φ^2 et le graphe d'écart correspondant sont donnés à la figure 2.7.

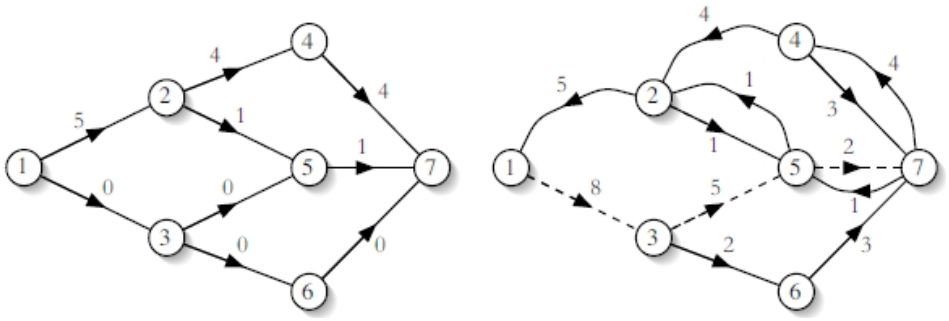


Fig. 2.7 : Flot à l'itération numéro 2 et graphe d'écart

En poursuivant ainsi l'algorithme, on obtient les situations décrites aux figures 2.8 et 2.9.

Après l'itération numéro 4, on constate qu'il n'existe plus de chemin joignant le sommet 1 au sommet 7 dans le graphe d'écart.

L'algorithme s'achève donc et le flot maximal est celui obtenu lors de cette dernière étape $\varphi_1 = \varphi_7 = 9$.

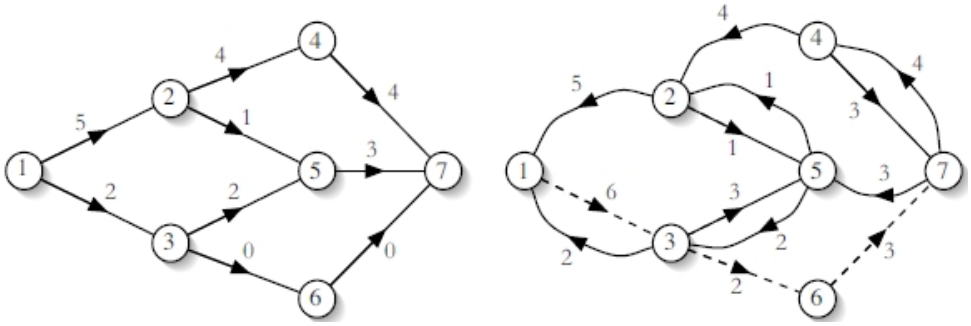


Fig. 2.8 : Flot à l'itération numéro 3 et graphe d'écart

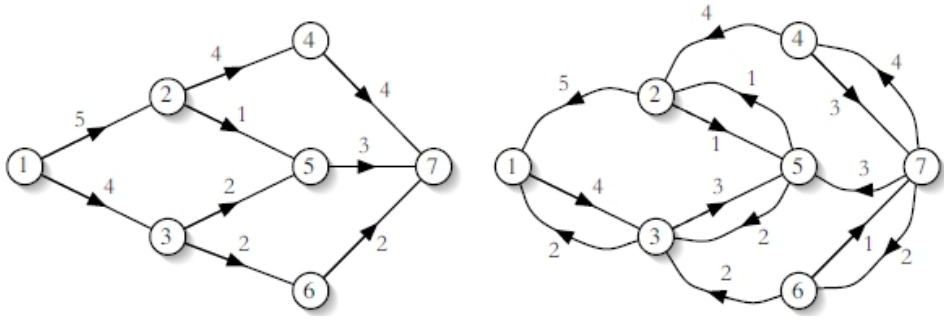


Fig. 2.9 : Flot à l'itération numéro 4 et graphe d'écart

Notons également que l'algorithme peut être adapté au cas où les capacités des arcs ont des bornes inférieures non nulles (positives, voire même négatives). En pratique, on associe à la plupart des réseaux de transport, une notion de coût unitaire du transport qui se traduit par l'association, à chaque arc a du réseau, d'un nombre réel $w(a)$ représentant ce coût unitaire.

Le coût total d'un flot φ est défini comme la somme des coûts associés aux flots élémentaires :

$$W(\varphi) = \sum_{a \in A} w(a)\varphi(a).$$

On peut alors se poser le problème consistant à rechercher, parmi les flots à valeur maximale, celui qui est à coût minimal. L'algorithme de Ford et Fulkerson peut également être adapté à cette situation.

Chapitre 3

Implémentation des algorithmes en Matlab

3.1 Implémentation de l'algorithme de Moore-Dijkstra

Le programme Matlab ci-dessous est une implémentation de l'algorithme de Moore-Dijkstra permettant l'obtention du plus court chemin entre deux sommets.

Les trois paramètres d'appel de la fonction moore sont les suivants :

L : matrice des longueurs (s'il n'y a pas d'arc entre le sommet i et le sommet j, alors $l_{ij} = \infty$ et conventionnellement $l_{ii} = 0$).

dep : numéro du sommet de départ du chemin recherché.

arv : numéro du sommet d'arrivée du chemin recherché.

En sortie, on récupère :

s : la longueur du plus court chemin ($s = \infty$ s'il n'existe pas de chemin).

m : le vecteur des longueurs des plus courts chemins entre dep et tous les autres sommets.

```
function [s,mark]=moore(L,dep,arv)
%MOORE Recherche du plus court chemin entre deux sommets d'un graphe
```

```
%   S=MOORE(L,DEP,ARV) calcule la longueur du plus court chemin entre
%   les sommets de numéro DEP et ARV.
%   [S,M]=MOORE(L,DEP,ARV) renvoie également le vecteur des longueurs
%   des plus courts chemins entre le sommet DEP et tous les autres sommets.
%   Voir aussi MOORE1
n=size(L,1);
% Initialisation
Sb =1 :n;
Sb(dep)=[];
mark=L(dep, :);
while 1
    [v,indj]=min(mark(Sb));
    j=Sb(indj);                          % j sommet de marque provisoire minimale
    Sb(indj)=[];                         % on retire j de Sb
    if isempty(Sb),break,end             % si Sb est vide on arête
    succ=find(L(j, :)~=inf & L(j, :)~=0); % liste des successeurs de j
    A=intersect(succ,Sb);                % intersection entre cette liste et Sb
    if ~isempty(A)                       % si cette liste n'est pas vide
    mark(A)=min(mark(A),mark(j)+L(j,A));% on remet à jour les marques
    end
end
s=mark(arv);
```

Considérons le graphe de la figure 3.1 :

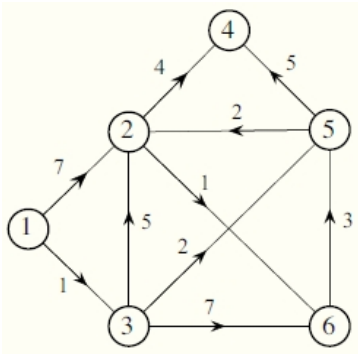


Fig. 3.1 : Graphe orienté

Voilà ci-dessous un exemple de programme d'appel pour calculer les longueurs des plus courts chemins joignant le sommet 1 à tous les autres :

```
clear
% Matrice de distance
D=[ 0 7 1 inf inf inf
    inf 0 inf 4 inf 1
    inf 5 0 inf 2 7
    inf inf inf 0 inf inf
    inf 2 inf 5 0 inf
    inf inf inf inf 3 0 ];
%
dep=input('Sommet de départ : ');
arv=input('Sommet d"arrivée : ');
[s,m]=moore(D,dep,arv)
```

Pour dep = 1 et arv = 6, on obtient :

```
Sommet de départ : 1
Sommet d'arrivée : 6
```

$s = 6$

$m = 0\ 5\ 1\ 8\ 3\ 6$

La fonction précédente peut être modifiée de manière à mémoriser, en plus de la longueur du chemin, la succession des sommets par lesquels il passe. Il suffit pour cela de maintenir à jour une table de prédécesseurs. Celle-ci peut ensuite être exploitée pour reconstituer le chemin suivi.

Le programme Matlab ci-dessous réalise cette fonction. Les trois paramètres d'appel de la fonction moore1 sont identiques à ceux de la fonction précédente. En sortie, on obtient :

s : la longueur du plus court chemin ($s = \infty$ s'il n'existe pas de chemin).

chemin : le vecteur des numéros de sommets décrivant le chemin.

```
function [s,chemin]=moore1(L,dep,arv)
%MOORE1 Recherche du plus court chemin entre deux sommets d'un graphe
%   S=MOORE1(L,DEP,ARV) calcule la longueur du plus court chemin entre
%   les sommets de numéro DEP et ARV.
%   [S,CHEMIN]=MOORE1(L,DEP,ARV) renvoie également le chemin emprunté
%   (séquence de numéro de sommets).
%   Voir aussi MOORE
n=size(L,1);
% Initialisation
Sb=1:n;
Sb(dep)=[];
mark=L(dep,:);
p=ones(1,n)*inf;
succ=find(L(dep,:)~=inf & L(dep,:)~=0);
p(succ)=dep;
while 1
  [v,indj]=min(mark(Sb));
  j=Sb(indj);                          % j sommet de marque provisoire minimale.
```

```
    Sb(indj)=[] ;                                  % on retire j de Sb.
    if isempty(Sb),break,end                       % si Sb est vide on arête.
    succ=find(L(j, :)~=inf & L(j, :)~=0) ;         % on cherche la liste des successeurs de j.
    A=intersect(succ,Sb) ;                         % intersection entre cette liste et Sb.
    if ~isempty(A)                                 % si cette liste n'est pas vide
      oldmark=mark(A) ;                            % on mémorise l'ancienne marque
      mark(A)=min(mark(A),mark(j)+L(j,A)) ;        % et on remet à jour les marques.
      id=find(oldmark~=mark(A)) ;                  % pour les sommets dont les marques ont
      p(A(id))=j ;                                 % changé on mémorise le prédécesseur.
    end
  end
  s=mark(arv) ;
  chemin=[] ;
  if isfinite(p(arv))
    chemin=[arv] ;                                 % on construit le chemin
    while chemin(end)~=dep                         % à l'aide de la table des prédécesseurs
      k=chemin(end) ;                              % à partir de la fin.
      chemin=[p(k) chemin] ;
    end
  end
```

Ci-dessous un exemple de programme d'appel correspondant au graphe de la figure 3.2 :

```
  clear
  % Matrice de distance
  D=[ 0 7 1 inf inf inf
      inf 0 inf 4 inf 1
      inf 5 0 inf 2 7
      inf inf inf 0 inf inf
```

```
      inf 2 inf 5 0 inf
      inf inf inf inf 3 0 ] ;
%
dep=input('Sommet de départ : ') ;
arv=input('Sommet d'arrivée : ') ;
[s,chemin]=moore1(D,dep,arv) ;
if isfinite(s)
   fprintf('Le chemin ')
   fprintf('%i ',chemin)
   fprintf('de longueur %f ',s)
   fprintf('est le chemin de longueur minimale\n')
else
fprintf('Il n'existe pas de chemin !\n')
end
```

Pour dep = 1 et arv = 6, on obtient :

```
Sommet de départ : 1
Sommet d'arrivée : 6
Le chemin 1 3 5 2 6 de longueur 6.000000 est le chemin de longueur minimale
```

Pour dep = 4 et arv = 3, on obtient :

```
Sommet de départ : 4
Sommet d'arrivée : 3
Il n'existe pas de chemin !
```

3.2 Implémentation de l'algorithme de Floyd

Le programme Matlab ci-après est une implémentation de l'algorithme de Floyd permettant l'obtention de la matrice des plus courts chemins dans un graphe dont les arcs sont values positivement. Le seul paramètre d'appel de la fonction floyd est le suivant :

L : matrice des longueurs (s'il n'y a pas d'arc entre le sommet i et le sommet j, alors $l_{ij} = \infty$ et conventionnellement $l_{ii} = 0$).

En sortie, on récupère :

A : la matrice des plus courts chemins entre tous les sommets.

P : la matrice des prédécesseurs (pij représente le numéro du sommet prédécesseur immédiat de j sur le plus court chemin entre i et j).

```
function [A,P]=floyd(L)
%FLOYD Recherche de la matrice des plus courts chemins entre sommets
%   [A]=FLOYD(L) calcule a matrice des plus courts chemins entre sommets
%   [A,P]=FLOYD(L) renvoie également la matrice des prédécesseurs
[n,m]=size(L);
if n ~= m error('La matrice des distance n'est pas carrée');end
A=L;
for i=1 :n
  P(i, :)=i*ones(1,m);
end
for k=1 :n
  for i=1 :n
    for j=1 :n
      if A(i,k)+A(k,j)<A(i,j)
        A(i,j)=A(i,k)+A(k,j);
        P(i,j)=P(k,j);
      end
    end
  end
end
```

Considérons de nouveau le graphe de la figure 3.1. Le programme ci-après permet de calculer la matrice des plus courts chemins et d'éditer, en exploitant la matrice des

prédécesseurs, le chemin reliant un sommet initial à un sommet terminal quelconques sous la forme d'une liste de sommets.

```
clear ;
% Matrice de distance
C=[ 0 7 1 inf inf inf
      inf 0 inf 4 inf 1
      inf 5 0 inf 2 7
      inf inf inf 0 inf inf
      inf 2 inf 5 0 inf
      inf inf inf inf 3 0 ] ;
%
[A,P]=floyd(C)
initial=input('Sommet initial : ') ;
terminal=input('Sommet terminal : ') ;
%
if ~isfinite(A(initial,terminal))
  fprintf('Il n'existe pas de chemin entre les sommets %d et %d\n\n',initial,terminal) ;
else
   j=terminal ;
   chemin=[j] ;
   while j ~= initial
       j=P(initial,j) ;
       chemin=[j chemin] ;
   end
   fprintf('Plus court chemin entre les sommets %d et %d\n\n',initial,terminal)
   fprintf('%d ',chemin) ;
   fprintf('\n\n')
end
```

Pour initial = 1 et terminal = 6, on obtient :

Plus court chemin entre 1 et 6

1 3 5 2 6

Alors que pour initial = 6 et terminal = 1, le résultat est le suivant :

Il n'existe pas de chemin entre 6 et 1

3.3 Implémentation de l'algorithme de Prim

Le programme Matlab ci-après est une implémentation de l'algorithme de Prim permettant l'obtention d'un arbre couvrant de poids minimal. Les deux paramètres d'appel de la fonction prim sont les suivants :

L : matrice des longueurs (s'il n'y a pas d'arc entre le sommet i et le sommet j, alors $l_{ij} = \infty$ et conventionnellement $l_{ii} = 0$).

dep : numéro du sommet de départ pour la construction de l'arbre recherché.

En sortie, on récupère :

alpha : un tableau de chaînes de caractères désignant les arêtes constituant l'arbre.

```
function [alpha]=prim(L)
%PRIM Recherche d'un arbre couvrant de poids minimal
%   ALPHA=PRIM(L) détermine un arbre couvrant de poids minimal
%   ALPHA est un tableau de chaînes de caractères désignant les
%   arêtes de l'arbre.
n=size(L,1);
mark=ones(1,n)*inf;
alpha=ones(n,3)*abs('-');        % liste des arêtes de l'arbre
dep=1;                           % sommet de départ (fixé à 1)
mark(dep)=0;
A=[];
S=1 :n;
```

```
while 1
   S=setdiff(S,A) ;                  % ensemble X\A
   [val,l]=min(mark(S)) ;            % recherche du sommet ayant la plus petite marque
   i=S(l) ;                          % numéro du sommet
   if isempty(setxor(A,1 :n)),break, end   % si A=X alors FIN
   A=[A i] ;                                % ajout du sommet à l'arbre
   succ=find(L(i, :)~=inf & L(i, :)~=0) ;  % successeurs du sommet i
   E=setdiff(succ,A) ;               % liste des successeurs moins les sommets déjà dans S
   for k=1 :length(E)                % mise à jour des marques
      if L(i,E(k))<mark(E(k))
         mark(E(k))=L(i,E(k)) ;
         alpha(E(k), :)=[num2str(i) '-' num2str(E(k))] ;% mémorisation arêtes
      end
   end
end
```

Considérons le graphe de la figure 3.2. Le programme ci-après permet de déterminer un arbre couvrant de ce graphe.

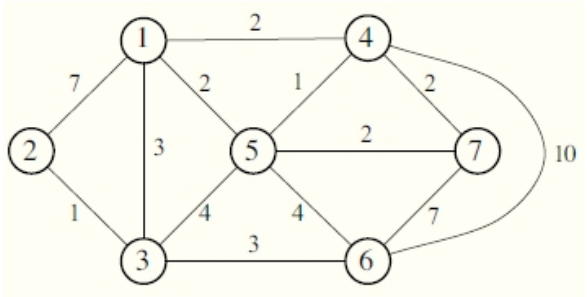


Fig. 3.2 : Graphe simple

```
clear
% 1 2 3 4 5 6 7
```

D=[0 7 3 2 2 inf inf

7 0 1 inf inf inf inf

3 1 0 inf 4 3 inf

2 inf inf 0 1 10 2

2 inf 4 1 0 4 2

inf inf 3 10 4 0 7

inf inf inf 2 2 7 0]

%

alpha=prim(D) ;

setstr(alpha)

Le résultat obtenu est :

ans =

—

3-2

1-3

1-4

4-5

3-6

4-7

Cela correspond à l'arbre suivant :

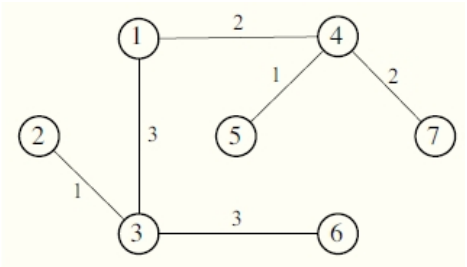


Fig. 3.3 : Arbre couvrant de poids minimal.

Conclusion

La théorie des graphes constitue aujourd'hui un corpus de connaissances très important. Les derniers travaux en théorie des graphes sont souvent effectués par des informaticiens, du fait de l'importance qu'y revêt l'aspect algorithmique.

Effectivement, il s'agit essentiellement de modéliser des problèmes. Nous exprimons le problème en termes de graphes et ensuite il devient un problème de la théorie des graphes que nous savons le plus souvent résoudre car il rentre dans une catégorie de problèmes connus.

Les solutions de problèmes de graphes peuvent être faciles et efficaces (car le temps nécessaires pour les traiter informatiquement est raisonnable car il dépend polynomialement du nombre de sommets du graphe) ou difficiles (car le temps de traitement est exponentiel) dans quel cas nous utilisons une heuristique, c'est-à-dire un processus de recherche d'une solution (pas forcément la meilleure). Dans ce travail nous utilisons les algorithmes suivants : l'algorithme de Bellman, Dijkastra et Floyd Warshall pour calculer les plus courts chemins, l'algorithme de Prim pour rechercher d'un arbre couvrant de poids minimal, on a utilisé l'algorithme de Ford et Fulkerson pour rechercher un flot maximal .

Bibliographie

[1] B. Roy. *Algèbre Moderne et Théorie des Graphes, tomes 1 & 2.* Dunod, 1970.

[2] B. Carre. *Graphs and Networks,* Clarendon Press, Oxford, 1979.

[3] C. Berge. *Théorie des graphes et ses applications,* Dunod, Paris, 1958.

[4] C. Berge. *Graphes,* Gauthiers-Villars, Paris, 1983

[5] D. Maquin. *Eléments de Théorie des Graphes et Programmation Linéaire Cours de l'Ecole Nationale d'Electricité et de Mécanique,* INPL, 2008.

[6] D. West. *Introduction to Graph Theory,* Prentice Hall, 1996.

[7] F. Droesbeke, M. Hallin, C. Lefevre. *Les graphes par l'exemple,* Ellipses,1987.

[8] G. Desbazeille. *Exercices et problèmes de recherche opérationnelle,* Dunod, Paris, 1976.

[9] M. Couprie. *Graphes et algorithmes Notes de cours et exercices,* 2010.

[10] M. Gondran, M. Minoux. *Graphes et algorithmes,* Eyrolles, Paris, 1984.

[11] N. Biggs, E. Lloyd, R. Wilson. *Graph Theory,* Clarendon Press Oxford, 1976.

[12] N. Deo. *Graph theory with applications to engineering and computer science,* Prentice-Hall, Englewood cliffs (N.J.) ,1975.

[13] P. Lopez. *Cours de Graphes,* LAAS–CNRS, 2005

[14] R. Faure. *Précis de recherche opérationnelle,* Dunod, Paris, 1979.

[15] Roseaux. *Exercices et problèmes résolus de recherche opérationnelle. Tome 1. Graphes : leurs usages, leurs algorithmes,* Dunod, Paris, 1998.

Résumé

Ce travail constitue une introduction à la théorie des graphes.

Le contenu est composé de trois chapitres. Le premier « Etude sur les concepts des graphes » présente les concepts généraux. Le deuxième « Etude algorithmique de quelques problèmes » comme le problème de plus court chemin qui aborde certainement l'un des plus fameux sujets de la théorie des graphes, en présentant les principaux algorithmes de recherche de chemins de longueur minimale dans un graphe.

Et le problème du l'arbre à coût minimum. Et le problème du flots dans les réseaux qui parle plus particulièrement des réseaux de transport et la recherche d'un flot maximum.

Le troisième chapitre « Implémentation des algorithmes en Matlab » plus clairs pour la compréhension, décrite dans ce travail quelques implémentations à l'aide du langage Matlab et on fait des exemples.