L3 Maths Info
Algorithmes dans les graphes

Notes de cours 2024-2025

Virginie Gabrel
email : gabrel@lamsade.dauphine.fr

1 Définition, concepts élémentaires et vocabulaire

La théorie des graphes est un outil privilégié de modélisation et de résolution de problèmes dans un grand nombre de domaines allant de la science fondamentale aux applications technologiques concrètes. Par exemple, les graphes déterministes et aléatoires sont utilisés en chimie (modélisation de structure), en sciences sociales (pour représenter des relations entre groupes d'individus), en mathématiques combinatoires, en informatique (structures de données et algorithmique). Concernant les applications, elles sont très nombreuses : réseaux électriques et transport d'énergie, routage de véhicules et organisation de tournées, problèmes de localisation (localisation d'entrepôts dans les réseaux de distribution de marchandises, d'antennes ...), problèmes d'ordonnancements de tâches et d'affectation de ressources...

1.1 Graphe orienté

Exemple introductif. On a réalisé une étude marketing sur 8 nouveaux parfums, notés de P1 à P8. On a interrogé un panel représentatif de clients et on a obtenu les relations suivantes :
- P1 est préféré à P2,
- P2 est préféré à P4 et P5,
- P3 est préféré à P1 et P6,
- P4 est préféré à P1, P6 et P7,
- P5 est préféré à P7,
- P7 est préféré à P6 et P8,
- P8 est préféré à P5.

On souhaite déterminer quels sont les parfums préférés et quels sont ceux qui ont été les moins appréciés.

On peut représenter cette étude sous la forme d'un graphe : les sommets sont les parfums, et les arcs la relation 'est préféré à'. On obtient alors :

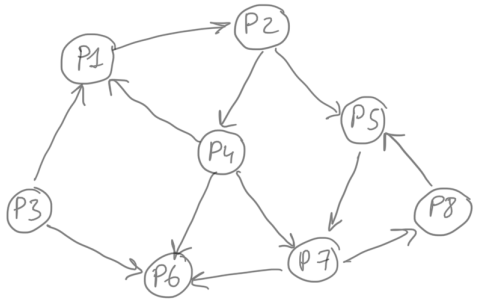


FIGURE 1 – Préférences sur les 8 parfums

Il est possible d'agréger et de réorganiser les relations et d'obtenir :

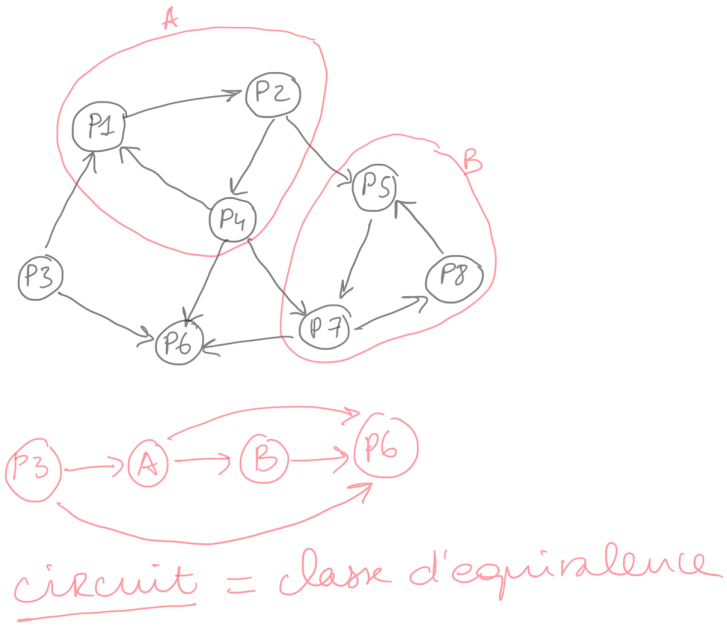


FIGURE 2 – Parfums préférés

Un **graphe orienté** $G = (X, U)$ est défini par :

— Un ensemble X dont les éléments sont appelés des **sommets**. L'ordre de G est le nombre n de sommets de G. Par la suite, les sommets seront numérotés de 1 à n.

— un ensemble U dont les éléments $u = (i, j)$ sont des couples ordonnés de sommets appelés des **arcs**. On dit que i est l'**extrémité initiale** et j l'**extrémité terminale**, j est **successeur** de i et i est **prédécesseur** de j. Par la suite, on aura $|U| = m$.

Les arcs représentent une relation binaire définie sur $X \times X$.

Un graphe $G = (X, U)$ est dit :
- symétrique si $u = (i, j) \in U$ alors $u = (j, i) \in U$
- antisymétrique si $u = (i, j) \in U$ alors $u = (j, i) \notin U$

Une **boucle** est un arc ayant le même sommet comme extrémité initiale et terminale.

Un **p-graphe** est un graphe sans boucle dans lequel il n'existe jamais plus de p arcs de la forme (i, j) entre i et j.

La densité d'un 1-graphe est donnée par le quotient m/n^2.

L'ensemble des successeurs de i se note $\Gamma^+_G(i)$ ou $\Gamma^+(i)$ quand il n'y a pas d'ambiguïté sur le graphe. L'ensemble de ses prédécesseurs se note $\Gamma^-_G(i)$ ou $\Gamma^-(i)$. On appelle l'ensemble de ses voisins l'ensemble : $\Gamma_G(i) = \Gamma^+_G(i) \cup \Gamma^-_G(i)$.

Le demi-degré extérieur (resp. intérieur) du sommet i, noté $d^+(i)$ (resp. $d^-(i)$) désigne le nombre d'arcs ayant i comme extrémité initiale (resp. terminale). On a :

$$d^+(i) = |\Gamma^+_G(i)| \text{ et } d^-(i) = |\Gamma^-_G(i)| \tag{1}$$

$$\sum_{i \in X} d^+(i) = \sum_{i \in X} d^-(i) = m \tag{2}$$

Le degré du sommet i, noté $d(i)$, est le nombre d'arcs ayant i comme extrémité. On a :

$$d(i) = d^+(i) + d^-(i) \tag{3}$$

$$\sum_{i \in X} d(i) = 2m \tag{4}$$

Etant donné un arc $u = (i, j)$, on dit que :
- les deux sommets i et j sont adjacents
- les deux sommets i et j sont adjacents à l'arc u
- l'arc u est incident aux deux sommets i et j

Deux arcs ayant une extrémité commune sont dits adjacents.

G^{-1} est le graphe inverse obtenu à partir de G en inversant le sens des arcs.

Graphe complet
Un graphe G est dit complet si pour toute paire (i, j) de sommets, il existe au moins un arc (i, j) ou (j, i).

Graphe plein
Un graphe G est dit plein si pour toute paire (i, j) de sommets, il existe l'arc (i, j) et l'arc (j, i).

1.2 Graphe non orienté

Exemple introductif. Étant donné un réseau social, l'objectif est d'identifier un groupe d'amis. Pour cela, on dispose de la description suivante du réseau :

— la liste des n personnes du réseau (indicées de 0 à $n - 1$) ;

— la liste des amis de chaque personne i, pour i allant de 0 à $n-1$.

Voici un exemple de réseau social contenant 8 personnes :

Les relations d'amitiés sont :
— les amis de 0 : 2, 3 ;
— les amis de 1 : 3, 5 ;
— les amis de 2 : 0, 3, 6, 7 ;
— les amis de 3 : 0, 1, 2, 4, 5, 6 ;
— les amis de 4 : 3, 7 ;
— les amis de 5 : 1, 3, 6, 7 ;
— les amis de 6 : 2, 3, 5 ;
— les amis de 7 : 2, 4, 5.

Une représentation graphique de ce réseau est proposée dans la figure 3.

On suppose que la relation d'amitié liant deux personnes est **symétrique** : si i est l'ami de j, alors j est l'ami de i.

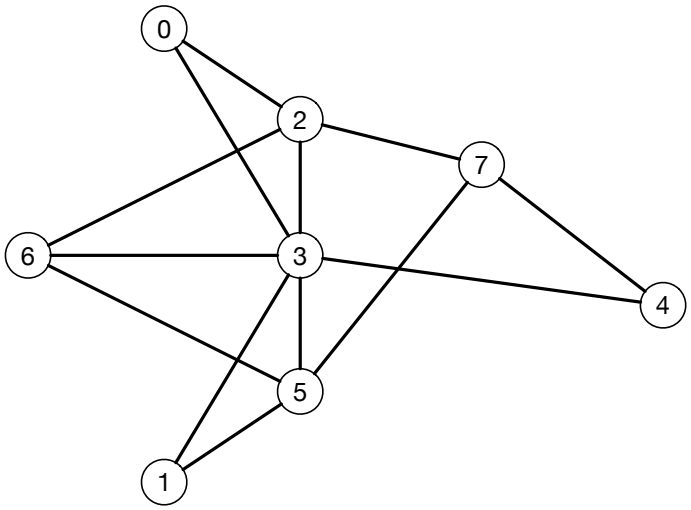


FIGURE 3 – Réseau à 8 personnes

Un **graphe non orienté** $G = (X, E)$ est défini par :
— L'ensemble X de sommets.
— L'ensemble E dont les éléments $e = ij$ (parfois noté $\{i, j\}$) sont des couples non ordonnés de sommets appelés des **arêtes**.

L'ensemble $\Gamma(i) = \{j \in X : ij \in E\}$ est appelé le voisinage de i et le degré de i est donné par $d(i) = |\Gamma(i)|$.

Les arêtes représentent une relation binaire symétrique définie sur $X \times X$.

Un **multigraphe** est un graphe non orienté pour lequel il peut exister plusieurs arêtes entre deux sommets i et j donnés.

Un graphe est dit **simple** s'il est sans boucle et, s'il n'existe pas plus d'une arête entre deux sommets quelconques.

Etant donné une arête $e = ij$, les deux sommets i et j sont dits adjacents et l'arête e est dite adjacente aux sommets i et j. Deux arêtes sont dites adjacentes si elles ont une extrémité commune.

Etant donné un **graphe non orienté** $G = (X, E)$, son graphe complémentaire, noté $\bar{G} = (X, \bar{E})$ est défini par : $ij \in \bar{E}$ ssi $ij \notin E$.

Pour passer d'un graphe non orienté $G = (X, E)$ à un graphe orienté $G' = (X, U)$, on transforme chaque arête en deux arcs en sens inverse :

$$\forall e = ij \in E, (i, j) \in U \ \texttt{et} \ (j, i) \in U$$

Pour passer d'un graphe orienté $G = (X, U)$ à un graphe non orienté $G' = (X, E)$, on relie par une arête tous les couples i et j de sommets reliés par un arc ((i, j) ou (j, i)) dans G :

$$\forall i, j \in X \times X, ij \in E \ \texttt{si} \ (i, j) \in U \ \texttt{et/ou} \ (j, i) \in U$$

1.3 Quelques graphes particuliers

Clique (non orienté)

Un sous-ensemble de sommets $C \subseteq X$ tel que deux sommets quelconques de C sont reliés par une arête est appelé une clique. Un sommet isolé constitue à lui seul une clique. Une clique d'ordre l est noté K_l.

Un problème classique consiste à déterminer une clique de cardinalité maximale. Un autre concerne la partition de X en un nombre minimal de cliques.

Stable (non orienté)

Un sous-ensemble de sommets $S \subseteq X$ tel que deux sommets quelconques de S ne sont pas reliés par une arête est appelé un stable. Un sommet isolé constitue à lui seul un stable.

Un problème classique consiste à déterminer un stable de cardinalité maximale. Un autre concerne le partition de X en un nombre minimal de stables, également connu comme le **problème de coloration**.

On remarque qu'un stable dans G est une clique dans $\bar{G}$. Ainsi, le problème de coloration dans G est exactement le problème de la partition en un nombre minimal de

cliques dans $\bar{G}$.

Graphe biparti
Un graphe $G = (X, U)$ est biparti si l'ensemble X des sommets peut être partitionné en deux sous-ensembles X_1 et X_2 ($X = X_1 \cup X_2$ et $X_1 \cap X_2 = \emptyset$) tel que :

$\forall e = (i, j)$, $i \in X_1$ et $j \in X_2$, ou bien $j \in X_1$ et $i \in X_2$.

Un graphe biparti complet est noté $K_{r,s}$ avec $r = |X_1|$ et $s = |X_2|$.

Graphe planaire
C'est un graphe que l'on peut représenter sur un plan sans que les arcs (ou arêtes) ne se coupent.

Sous-graphe engendré par un sous-ensemble de sommets
Soit $A \subset X$ un sous-ensemble de sommets, le sous-graphe engendré par A est le graphe G_A dont les sommets sont les éléments de A et les arcs les éléments de U ayant leur deux extrémités dans A.

Graphe partiel engendré par un sous-ensemble d'arcs
Soit $V \subset U$ un sous-ensemble d'arcs, le graphe partiel engendré par V est le graphe dont les sommets sont ceux de X et les arcs ceux de V.

Sous-graphe partiel
Soit $A \subset X$ et $V \subset U$, le sous-graphe partiel engendré par A et V est le graphe partiel de G_A engendré par V.

Un graphe (orienté ou non) est dit **valué** quand ses arcs et/ou ses sommets sont dotés d'un poids (ou longueur).

1.4 Connexite

1.4.1 Chaîne et cycle

Une **chaîne** est une **séquence d'arcs (ou d'arêtes) adjacent(e)s** $\{u_1, \ldots, u_q\}$ avec $u_{r-1} \neq u_r$ pour $r = 2, ..., q$. La longueur d'une chaîne est le nombre d'arcs (arêtes) lui appartenant (q dans notre exemple). Ainsi, dans une chaîne, chaque arc (ou arête) u_r ($2 \leq r \leq q - 1$) a une extrémité commune avec l'arc u_{r-1} et l'autre extrémité commune avec l'arc u_{r+1}. L'extrémité i de u_1 non adjacente à u_2 et l'extrémité j de u_q non adjacente à u_{q-1} sont appelées les extrémités de la chaîne. On note alors cette chaîne $\mu(i, j) = \{u_1, \ldots, u_q\}$.

Dans le cas d'un graphe simple, une chaîne de longueur q est parfaitement définie par la succession des $(q+1)$ sommets qu'elle rencontre : $\{i, i_1, \ldots, i_{q-1}, j\}$.

Un **cycle** est une chaîne $\{u_1, \ldots, u_q\}$ telle que :
- le même arc/arête ne figure pas deux fois dans la séquence
- les extrémités coïncident.

1.4.2 Chemin et circuit

Un **chemin** allant de i à j de longueur q est une séquence de q arcs $\mu[i,j] = \{u_1, \ldots, u_q\}$ avec : $u_1 = (i, i_1)$, $u_2 = (i_1, i_2)$,..., $u_q = (i_{q-1}, j)$. (uniquement pour les graphes orientés).

Un **circuit** est un chemin dont les extrémités coïncident.

Une chaîne, un cycle, un chemin ou un circuit est

1. **élémentaire** si les sommets qui le composent sont tous distincts (à l'exception des extrémités),
2. **hamiltonien** s'il passe une fois et une seule par chaque sommet du graphe,
3. **eulérien** s'il passe une fois et une seule par chaque arc du graphe,

1.4.3 Graphe connexe et composantes connexes

G est dit **connexe** si $\forall (i,j) \in X \times X$, $\exists \mu(i,j)$.

La relation R definie par $iRj \iff i = j$ ou $\exists \mu(i,j)$ est une relation d'équivalence car réflexive, symétrique et transitive.

Démonstration.
Réflexive : iRi
Symétrique : $iRj \Rightarrow jRi$ car il suffit d'inverser la chaîne
Transitive : iRj et $jRk \Rightarrow iRk$ car il suffit de concaténer les 2 chaînes.

Les classes d'équivalence induites sur X par R forment une partition de X en $X_1, X_2, \ldots, X_p$. Le nombre p de classes d'équivalence est appelé le **nombre de connexité** du graphe. Un graphe est dit connexe si et seulement si son nombre de connexité est égal à 1. Les sous-graphes $G_1, G_2, \ldots, G_p$ engendrés par les sous-ensembles $X_1, X_2, \ldots, X_p$ sont appelés les **composantes connexes** de G. Chaque composante connexe est un graphe connexe.

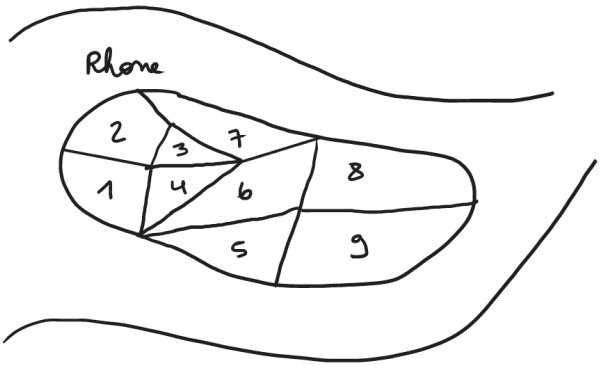
La vérification de la connexité d'un graphe est un des premiers problèmes de la théorie des graphes.

La connexité caractérise des classes de graphes très utilisés.

Qqs graphes particuliers :
- un graphe sans cycle (ou acyclique),
- un **arbre** est un graphe non orienté connexe sans cycle,
- une **forêt** est un graphe dont chaque composante connexe est un arbre.

Illustration : Problème de l'arbre couvrant

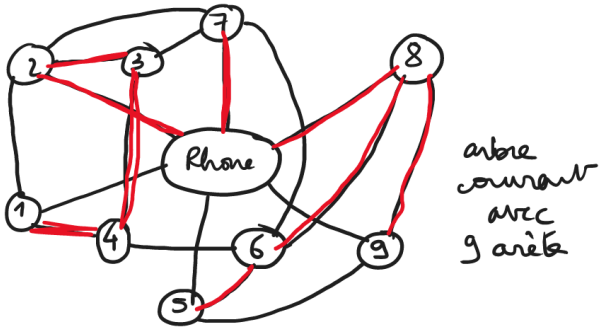
L'ile du Nivéou se consacre à la culture du riz. Sur cette ile se trouvent 9 champs entourés de murs et disposés de la façon suivante :



La culture du riz suppose que l'on puisse périodiquement inonder l'ensemble des champs. Cela est réalisé en ouvrant des vannes placées dans les murs séparant les champs et le Rhône ou les champs entre eux. Etant donné que l'installation d'une vanne est coûteuse, il s'agit de déterminer le nombre minimum de vannes et leur emplacement pour pouvoir, quand on le désire, inonder tous les champs.

Pour résoudre ce problème, on peut considérer le graphe non orienté comportant un sommet pour chaque champ plus un sommet représentant le Rhône. Ce graphe comporte une arête entre deux sommets si les champs correspondants, ou le Rhône, sont voisins. En considérant que lorsqu'on place une vanne sur un mur séparant deux champs (ou une vanne séparant un champ du Rhône) on conserve l'arête joignant les deux sommets correspondant aux champs (ou au champ et au Rhône), le problème revient à chercher un graphe partiel connexe (autrement dit, tous les champs doivent être reliés directement ou pas au Rhône). Comme on souhaite poser le moins de vannes possible, il s'agit de garder le moins d'arêtes possible. On cherche donc un graphe partiel connexe tel que si l'on supprime une arête de plus il ne soit plus connexe : il s'agit d'un arbre. Ici, étant donné que le graphe possède 10 sommets, l'arbre devra

comporter 10-1=9 arêtes. On devra installer 9 vannes.



1.4.4 Graphe fortement connexe et composantes fortement connexes

Un graphe est dit **fortement connexe** si, étant donné deux sommets quelconques i et j, il existe un chemin d'extrémité initiale i et d'extrémité terminale j et, il existe un chemin d'extrémité initiale j et d'extrémité terminale i.

Considérons la relation R' définie comme suit : $iR'j \iff i = j$ ou ($\exists \mu[i, j]$ et $\exists \mu[j, i]$). Cette relation est une relation d'équivalence car elle est réflexive, symétrique et transitive. Les classes d'équivalence induites sur X par R' forment une partition de X en $X_1, \ldots, X_q$. q est appelé le **nombre de connexité forte** du graphe. Les sous-graphes $G_1, \ldots, G_q$ de G engendrés par $X_1, \ldots, X_q$ sont fortement connexes et sont appelés les **composantes fortement connexes** de G. Un graphe est fortement connexe si et seulement s'il n'a qu'une seule composante fortement connexe.

On appelle le **graphe réduit**, noté G_r, le graphe défini comme suit : les sommets de G_r représentent les composantes fortement connexes et il existe un arc entre deux sommets i et j si et seulement si il existe au moins un arc entre un sommet de X_i et un sommet de X_j dans le graphe G. G_r est nécessairement sans circuit.

2 Représentation des graphes

Pour représenter un graphe en machine, il faut être capable de représenter pour chaque sommet s de notre graphe G, l'ensemble de ses successeurs/prédécesseurs ou voisins.

Il faut également que les opérations suivantes puissent être réalisées le plus efficacement possibles :

— tester si un arc/arête existe
— accèder à l'ensemble des successeurs/prédécesseurs/voisins d'un sommet
— insérer/supprimer un sommet et/ou un arc/arête

2.1 Par matrices

Matrices d'adjacence. Soit $G = (V, U)$ un 1-graphe, la **matrice d'adjacence** est une matrice carrée

$$A = (a_{ij})_{n \times n}$$

à coefficient 0 ou 1 est définie comme suit :

$a_{ij} = 1 \iff (i, j) \in U$;
$a_{ij} = 0$ sinon.

Exemple graphe orienté :

0	1	0	0	0
1	0	1	0	0
0	0	0	1	1
0	0	1	0	1
1	0	0	0	0

Ainsi, le nombre de 1 dans la ligne i vaut $d^+(i)$, et le nombre de 1 dans la colonne i vaut $d^-(i)$.

De même, pour un graphe $G = (V, E)$ non orienté, a_{ij} vaut 1 si le graphe contient l'arête ij (et 0 sinon).

Exemple graphe non orienté :

0	1	1	1
1	0	1	0
1	1	0	1
1	0	1	0

Ainsi, la matrice d'adjacence d'un graphe non orienté est symétrique. Le nombre de 1 dans la ligne i égale $d(i)$, de même que le nombre de 1 dans la colonne i.

Dans le cas de graphe pondéré sur les arêtes, on code le poids plutôt que 1 (attention au poids nul).

Matrices d'incidence. Soit $G = (V, U)$ un graphe orienté sans boucle, la **matrice d'incidence sommets-arcs** est une matrice

$$A = (a_{ij})_{n \times m}$$

à coefficients entiers 0, +1, -1 telle que chaque colonne correspond à un arc et chaque ligne à un sommet. Si $e = (u, v)$ est un arc du graphe alors la colonne e a tous ses termes nuls sauf :

$$a_{ue} = +1\,;$$
$$a_{ve} = -1.$$

Le nombre de +1 (resp. de -1) dans la ligne v égale $d^+(v)$ (resp. $d^-(v)$).

Exemple :

+1	-1	0	0	0	0	0	-1
-1	+1	+1	0	0	0	0	0
0	0	-1	+1	-1	+1	0	0
0	0	0	0	+1	-1	+1	0
0	0	0	-1	0	0	-1	+1

On stocke $n \times m$ => matrice creuse.

Soit $G = (V, E)$ un graphe non orienté sans boucle, la **matrice d'incidence sommets-arêtes** est une matrice $A = (a_{ij})$ ($n \times m$) à coefficients entiers 0 ou 1 telle que chaque colonne correspond à une arête et chaque ligne à un sommet. Si $e = ij$ est une arête du graphe alors la colonne e a tous ses termes nuls sauf :

$$a_{ie} = 1\,;$$
$$a_{je} = 1.$$

Dans le cas de graphe pondéré sur les arêtes, on code le poids plutôt que 1.

2.2 Représentation par listes d'adjacences

L'occupation mémoire d'une matrice d'adjacence est $O(n^2)$. Dans le cas de graphes peu denses ($m \ll n^2$) pour les graphes orientés et $m \ll n(n+1)/2$ pour les graphes

non orientés), il est plus avantageux de ne décrire uniquement que les termes non nuls de la matrice d'adjacence.

On mémorise pour chaque sommet la liste de ses prédécesseurs ou de ses successeurs en utilisant des listes chaînées par des pointeurs ou des curseurs. C'est donc un tableau de listes.

Exemple :

succ = $[[2], [1, 3], [4, 5], [3, 5], [1]]$

Pour les graphes non orienté, l'arête uv est stockée deux fois.

Dans le cas de graphe pondéré sur les arêtes, on ajoute le poids dans la structure d'arête.

2.3 Comparaison des représentations par matrices et par listes

Si l'on s'intéresse au test d'existence d'un arc entre i et j, la représentation par matrice est meilleure que la représentation par listes de successeurs. En effet, avec une matrice, il suffit d'accéder au coefficient ligne i colonne j pour tester s'il est égal à 0 ou 1. Cette opération se fait en $O(1)$. Par contre, avec une liste, il faut parcourir la liste avant de conclure. La taille maximale de cette liste est $n - 1$. En conséquence, cette opération se fait alors en $O(n)$. Le tableau 2.3 synthétise quelques résultats de complexité théorique sur des opérations élémentaires.

	Matrice d'adj	$\Gamma^+(i)$	$\Gamma^-(i)$
tester si un arc (u, v) existe	$O(1)$	$O(n)$	$O(n)$
accèder à l'ensemble des successeurs	$O(n)$	$O(1)$	$O(m)$
accèder à l'ensemble des prédecesseurs	$O(n)$	$O(m)$	$O(1)$
supprimer un arc	$O(1)$	$O(n)$	$O(n)$

FIGURE 4 – Complexité théorique des opérations élémentaires sur un graphe orienté

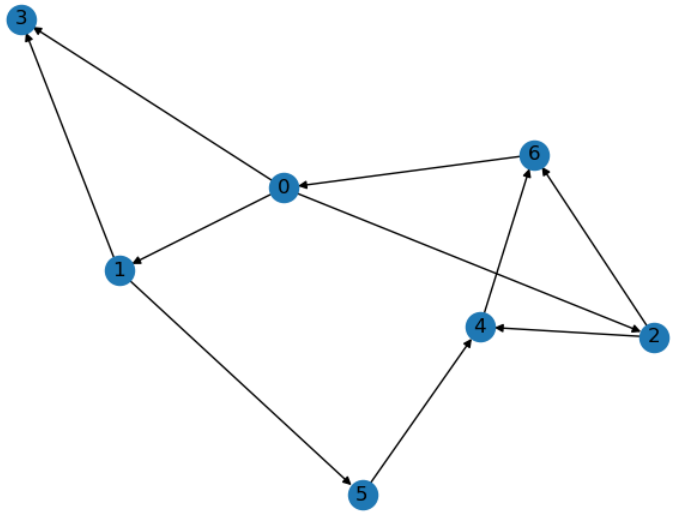
D'autres structures que les listes peuvent être utilisées pour représenter les listes de successeurs et de prédécesseurs. Par exemple, les tables de hachages qui vont permettre d'effectuer ces 4 opérations élémentaires en $O(1)$.

2.4 Représentation avec le module networkx de Python.

On utilise les dictionnaires (dict). Dans le module networkx, un graphe est un dictionnaire de dictionnaire de dictionnaire. En exécutant le programme ci-dessous :

```
import networkx as nx
import matplotlib.pyplot as plt
G=nx.DiGraph()
G.add_edges_from([(0,1),(0,2),(0,3),(1,3),(1,5),(2,4),(2,6),
(4,6),(5,4),(6,0)])
nx.draw(G, with_labels =True)
plt.show()
```

On obtient :



Et le graphe est un dictionnaire de dictionnaires :

```
>>> G.adj
AdjacencyView({0: {1: {}, 2: {}, 3: {}}, 1: {3: {}, 5: {}}, 2: {4: {},
6: {}}, 3: {}, 5: {4: {}}, 4: {6: {}}, 6: {0: {}}})
```

3 Parcours de graphes

De nombreux problèmes fondamentaux en théorie des graphes concernent la connexité. Par exemple :

— Un sommet j est-il accessible par un chemin à partir d'un autre sommet i ?
— Quel est l'ensemble de tous les sommets accessibles par un chemin à partir d'un sommet i ?
— Le graphe est-il connexe, c'est-à-dire tous les sommets du graphe sont-ils accessibles par une chaîne à partir d'un sommet donné i ?
— Le graphe est-il fortement connexe, c'est-à-dire existe-il un chemin joignant toute paire ordonnée de sommet (i, j) dans le graphe ?

3.1 Parcours en profondeur d'un graphe orienté (Depth First Search)

3.1.1 Algorithme

On explore G à partir d'un sommet i_0 quelconque. Au cours de l'exploration, chaque sommet peut se trouver dans l'un des deux états suivants :

— non exploré
— exploré

Un sommet i exploré peut être dans l'un des deux états suivants :

— fermé. C'est le cas quand tous les successeurs j de i ont été explorés,
— ouvert. C'est le cas lorsque certains successeurs j de i n'ont pas encore été explorés.

A l'itération courante, on part du sommet exploré i (exploré à partir de s). Si i est ouvert, on explore un successeur non encore exploré j de i ; j passe alors à l'état d'exploré et on commence une nouvelle itération à partir de j. On dit alors que l'arc (i,j) est un **arc d'arbre** ; il a permis d'explorer un nouveau sommet. Si i est fermé, il faut remonter sur le sommet s pour commencer une nouvelle itération à partir de s. L'exploration à partir de i_0 se termine lorsqu'on est remonté en ce sommet de départ i_0 et que le sommet i_0 est fermé. On a donc pu atteindre tous les sommets pouvant être atteints par un chemin à partir de i_0 dans le graphe. (Notons qu'un sommet i relié par un chemin à tous les autres sommets d'un graphe G est appelé **racine**). Voici une présentation en pseudo-langage de la fonction appelée `explore` :

```
FONCTION explore(graphe, sommet)
    etat[sommet] <- 'exploré'
    pour tout j successeur de sommet dans graphe
        si etat[j] == 'inexploré' alors
            explore(graphe, j)
```

Lorsqu'on appelle la fonction `explore` comme suit `explore(G,i_0)`, on explore les sommets de G qu'il est possible d'atteindre à partir de i_0 via un chemin.

Une **arborescence** de racine r, notée $\mathcal{A}_r$ est un graphe orienté tel que, pour tout sommet j de l'arborescence (autre que r), il existe un chemin unique allant de r à j. Un noeud i quelconque sur le chemin allant de r à j est appelé ascendant de j. j est un descendant de i. Si i est le prédécesseur de j dans le chemin allant de r à j alors i est également appelé le père de j et j le fils de i. Un sommet j sans fils est appelé une feuille. La longueur du chemin entre r et j est la profondeur de j dans l'arborescence. La plus grande profondeur dans l'arborescence est appelée la hauteur de l'arborescence.

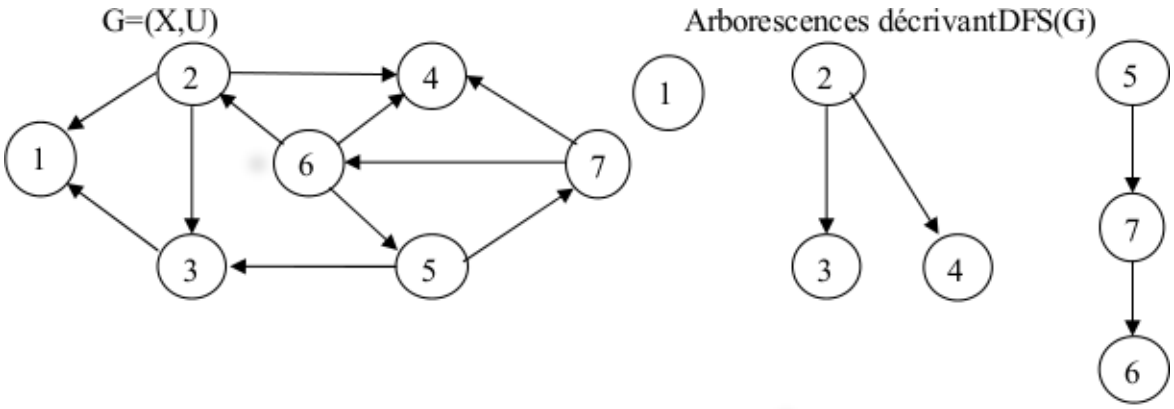
Après l'exécution de `explore(G,i_0)`, l'ensemble des arcs d'arbre constituent une arborescence, notée $\mathcal{A}_{i_0}$, décrivant l'exploration par DFS de G à partir de i_0.

Si après l'appel `explore(G,i_0)`, il reste des sommets non explorés (il s'agit des sommets j tels que il n'existe pas de chemin allant de i_0 à j), on doit choisir un autre sommet parmi les sommets non encore explorés, et appeler `explore` à partir de ce sommet pour continuer l'exploration du graphe. Et ainsi de suite tant qu'il reste des sommets non explorés. L'algorithme prend fin lorsque tous les sommets ont été explorés.

```
ALGORITHME DFS
    lire G, n
    pour i allant de 1 à n
        etat[i] <- 'inexploré'
    pour i allant de 1 à n
        si etat[i] == 'inexploré' alors
            explore(G, i)
```

On obtient alors une ou plusieurs arborescence(s) décrivant l'exploration de G par DFS.

Exemple :



La complexité de cet algorithme est en $O(m) + O(n)$. Pour un sommet i, la procédure explore contient la boucle "pour tout sommet j successeur du sommet i" qui consiste à parcourir la liste des successeurs de i afin de savoir si on peut explorer un sommet non encore exploré. Comme chaque sommet est exploré au plus une fois, le procédure explore appliquée à chaque sommet engendre $O(m)$ opérations au total. Dans DFS on a $O(n)$ opérations ce qui nous donne $O(m) + O(n)$. Or, si G connexe, $m \geq n - 1$ et le terme $O(n)$ est absorbé.

A l'issue de DFS(G), on obtient une caractérisation des arcs du graphe. On distingue 4 types d'arc :

- **Arc d'arbre** : arc ayant permis de visiter de nouveaux sommets lors de l'exploration par DFS.
- **Arc avant** : (i, j) n'est pas un arc d'arbre et il existe un chemin allant de i à j dans une arborescence (en d'autres termes i est un ancêtre de j dans l'une des arborescences décrivant l'exploration par DFS).
- **Arc arrière** : (i, j) n'est pas un arc d'arbre et il existe un chemin allant de j à i dans l'une des arborescences décrivant l'exploration par DFS (en d'autres termes i est un descendant de j dans une des arborescences décrivant l'exploration par DFS).
- **Arc croisé** : (i, j) n'est pas un arc d'arbre et il n'existe pas de chemin allant de i à j et de j vers i dans les arborescences décrivant l'exploration par DFS (en d'autres termes il n'existe aucun lien de parenté entre ces deux sommets).

Remarque 3.1 *Le parcours en profondeur d'un graphe s'applique également à un graphe non orienté en explorant, non pas les successeurs des sommets, mais ses voisins.*

3.1.2 Numérotations préfixe et suffixe

On peut ajouter des instruction à l'algorithme DFS(G) pour mémoriser l'ordre d'exploration des sommets du graphe, décrit par la numérotation préfixe, et l'ordre de fermeture des sommets (fin d'un appel récursif), décrit par la numérotation suffixe, comme suit :

```
FONCTION exploreNum(graphe, i, A, pref, suff, p, s)
    pref[i] <- p
    p <- p+1
   pour tout j successeur de i dans graphe
        si pref[j]==0 ALORS
            A <- A + (i,j)
            exploreNum(graphe, j, A, pref, suff, p, s)
    suff[i] <- s
    s <- s+1

ALGORITHME DFS_RECURSIF_PREF_SUFF
    lire G,n
    pour i allant de 1 à n
       pref[i] <- 0
       suff[i] <- 0
    A <- ensemble vide
    p <- 1
    s <- 1
    POUR i allant de 1 à n
        SI pref[i]==0 ALORS
            exploreNum(G, i, A, pref, suff, p, s)
```

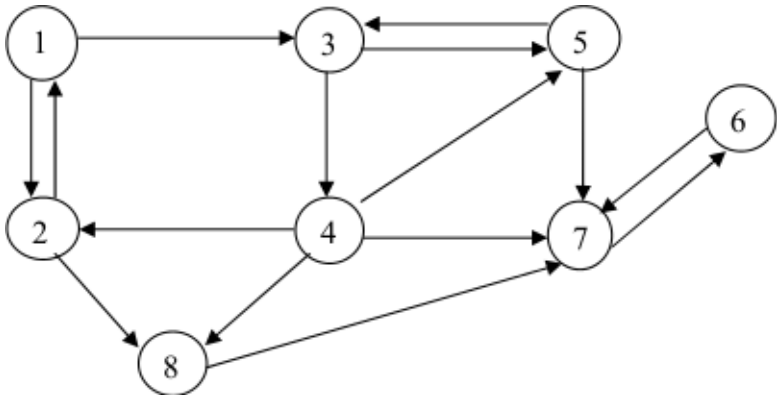
Grâce à ces numérotations, on peut facilement caractériser les types d'arcs : avant, arrière ou croisé.

(i, j) est avant	préfixe(i) < préfixe(j)	suffixe(i) > suffixe(j)
(i, j) est croisé	préfixe(i) > préfixe(j)	suffixe(i) > suffixe(j)
(i, j) est arrière	préfixe(i) > préfixe(j)	suffixe(i) < suffixe(j)

Exemple :

Les numérotations préfixes et suffixes sont :

i	1	2	3	4	5	6	7	8
$pref(i)$	1	2	6	7	8	5	4	3
$suff(i)$	8	4	7	6	5	1	2	3



Arcs arrières : $(2,1), (6,7), (5,3)$
Arcs avant : $(3,5)$
Arcs croisés : $(4,2), (4,8), (4,7), (5,7)$

3.1.3 Version itérative (avec une Pile)

Une pile (stack en anglais) est un type abstrait de données qui correspond à des valeurs enregistrées selon le principe du dernier entrée / premier sortie (comme une pile d'assiettes) - LIFO en anglais : last in / first out.

En pseudo code, on définit :
- sommetPile(P) : retourne la valeur au sommet de P (dernière entrée) sans la supprimer
- PileVide() : retourne une pile ne contenant aucune valeur
- empiler(v,P) : ajoute la valeur v au sommet de P (dernière entrée)
- empiler(v,P) : ajoute la valeur v au sommet de P (dernière entrée)
- depiler(P) : retourne la valeur v au sommet de P et supprime cette valeur de la pile

```
ALGORITHME DFS_itératif
lire G, n
pour i allant de 1 à n
   pref[i] <- 0
   suff[i] <- 0
P <- PileVide()
A <- ensemble vide
p <- 1
s <- 1
Pour i allant de 1 à n
    Si pref[i]==0 Alors
        empiler(P,i)
```

```
pref[i] <- p
p <- p+1
Tant que P non vide faire
    v <- sommetPile(P)
    Si existe un successeur j de v t.q pref[j]==0 alors
        empiler(P,j)
        pref[j] <- p
        p <- p+1
        A <- A + (v,j)
    Sinon
        i <- depiler(P)
        suff[i] <- s
        s <- s+1
```

En python, on implémente une pile P à l'aide du type list :
- on ajoute/empile une nouvelle valeur v dans la liste P avec la méthode append : P.append(v)
- si P est non vide, on retire/dépile la valeur en fin de liste avec la méthode pop : P.pop()

3.2 Détermination des composantes connexes

3.2.1 Cas non orienté

Pour déterminer les composantes connexes d'un graphe $G = (X, U)$ non orienté, il suffit d'appliquer l'algorithme de parcours du graphe en profondeur d'abord. Chaque arborescence obtenue couvre des sommets appartenant à une même composante connexe et, le nombre d'arborescence est donc le nombre de connexité de G.

3.2.2 Cas orienté

Pour déterminer les composantes connexes d'un graphe $G = (X, U)$ orienté, il faut transformer G en un graphe G' non orienté et appliquer DFS(G'). Chaque arborescence obtenue couvre des sommets appartenant à une même composante connexe et, le nombre d'arborescence est donc le nombre de connexité de G.

3.3 Détermination des composantes fortement connexes

L'algorithme de Kosaraju-Sharir utilise la numérotation suffixe pour déterminer les composantes fortement connexes d'un graphe $G = (X, U)$

1. Appliquer DFS(G) en numérotant les sommets dans l'ordre suffixe
2. Construire le graphe $G^{-1} = (X, U^{-1})$ obtenu en inversant le sens des arcs de G

3. Appliquer DFS(G^{-1}) en démarrant par le sommet de plus grand numéro suffixe et itérer le processus à partir du sommet non marqué de plus grand numéro suffixe.

Théorème

Chaque arborescence $\mathcal{A}_i$ décrivant l'exploration par DFS(G^{-1}) couvre des sommets constituant une composante fortement connexe de G.

Preuve

1. Il faut commencer par montrer que si s et t, deux sommets de G, appartiennent à une même composante fortement connexe, alors ils appartiennent à une même arborescence décrivant l'exploration par DFS de G^{-1}.
   Si s et t appartiennent à la même composante, il existe un chemin allant de s à t et un autre allant de t à s dans G. En conséquence, il existe dans G^{-1} un chemin allant de t à s et un autre allant de s à t. Ainsi, si on effectue un parcours en profondeur d'abord dans G^{-1} à partir d'un sommet x et que l'on atteint s alors on atteindra nécessairement t car s et t sont mutuellement accessibles et inversement.

2. Il faut également montrer que si s et t appartiennent à une même arborescence décrivant l'exploration par DFS de G^{-1}, alors s et t appartiennent à la même composante fortement connexe de G.

   Soit i la racine de l'arborescence contenant s et t. Il existe nécessairement un chemin allant de i à s dans G^{-1} et donc de s à i dans G. Par ailleurs, on a par construction : suffixe(i)>suffixe(s) ce qui signifie que, dans DFS(G), l'appel récursif en s s'est terminé avant celui en i. Comme il existe un chemin allant de s à i, cela signifie que i a été exploré avant s. on a donc prefixe(i)<prefixe(s) et suffixe(i)>suffixe(s) => Il existe donc un chemin allant de i à s dans G ce qui implique que i et s appartiennent à la même composante fortement connexe.

   Un raisonnement analogue permet de montrer que i et t sont aussi dans la même composante fortement connexe. Donc s et t appartiennent à la même composante fortement connexe.

Les q CFC d'un graphe G constituent une partition de G en sous-graphes fortement connexes : $G_1 = (X_1, U_1), \ldots, G_q = (X_q, U_q)$. On peut alors définir le **graphe réduit** de G, noté G^r : les q sommets $\{v_1, \ldots, v_q\}$ de G^r représentent les q CFC, respectivement $\{X_1, \ldots, X_q\}$ et, il existe un arc allant de v_i à v_j si et seulement si il existe au moins un arc entre un sommet de X_i et un sommet de X_j dans G.

3.4 Tri topologique dans un graphe orienté sans circuit

Les graphes sans circuit sont fréquents dans les applications. Par exemple, dans un problème d'ordonnancement, le graphe potentiels-tâches $G = (X, U)$, défini de la façon suivante : les sommets représentent les tâches à éxecuter et, (i, j) est un arc si et seulement si l'éxecution de la tâche i doit précéder celle de la tâche j (la longueur de l'arc (i, j) étant la durée de la tâche i), doit par construction être un graphe sans circuit.

Propriété
Un graphe G est sans circuit si et seulement si il n'existe pas d'arc arrière dans le parcours en profondeur d'abord de G.

Démonstration
Il faut montrer que s'il existe un circuit, alors il existe un arc arrière dans le parcours en profondeur d'abord de G. Soit le circuit $\mu = (i, i_0, ..., i_k, i)$. Si i est le premier sommet du circuit exploré avec DFS, i sera la racine d'un arbre comprenant au moins tous les autres sommets du circuit. Si un arc du circuit n'est pas un arc d'arbre, alors il s'agit d'un arc arrière. Et il existe au moins un arc du circuit qui ne sera pas un arc d'arbre !

Dans un graphe sans circuit, on peut définir les notions d'**ordre topologique** et de **fonction rang**.

Définition : Etant donné un graphe $G = (X, U)$ d'ordre n, on appelle ordre topologique une numérotation v des sommets de 1 à n telle que les numéros d'ordre $v(j)$ associés aux sommets j vérifient :

$$\forall (i, j) \in U : v(i) < v(j).$$

En d'autres termes, si l'on parcourt la liste des sommets dans l'ordre défini par une telle numérotation, un sommet j ne peut être rencontré que si l'on a, au préalable, rencontré tous ses prédécesseurs.

Remarque : il n'y a pas unicité (plusieurs ordres topologiques peuvent être définis sur un même graphe).

Propriété
L'ordre suffixe inversé est un ordre topologique.

Démonstration.

Comme il n'existe par d'arc arrière lorsque le graphe est sans circuit, l'ordre suffixe inversé est bien un ordre topologique.

Définition
Si l'on note R l'ensemble des sommets d'un graphe sans circuit G de demi-degré intérieur nul, on appelle fonction rang associée à G l'application rang qui a tout sommet j associe un nombre entier défini de la façon suivante :

— $rang(j) = 0 \; \forall j \in R$
— $rang(j)$ = nombre d'arcs dans un chemin de cardinal maximal joignant un sommet quelconque de l'ensemble R et le sommet j.

La fonction rang est unique et elle définit une partition de l'ensemble des sommets en niveaux, chaque niveau k étant formé par le sous-ensemble de sommets :

$$X_k = \{i \in X : rang(i) = k\}$$

Algorithme de détermination de la fonction rang
Debut
$R \leftarrow \{j \in X \texttt{ tel que } d^-(j) = 0\}$
rang(j) $\leftarrow 0 \;\; \forall j \in R$
$k \leftarrow 1$
Tant que $X \neq \emptyset$ faire
Debut Tant que
$X \leftarrow X \setminus R$
$R \leftarrow \{j \in X \texttt{ tel que } |\Gamma^-(j) \cap X| = 0\}$
rang(j) $\leftarrow k \;\; \forall j \in R$
$k \leftarrow k + 1$
Fin Tant que
Fin

Exercices :

1. Que peut-on dire d'un sommet i qui a une valeur rang(i)$= 1$?
2. Que peut-on dire d'un sommet i qui a une valeur rang(i)$= 2$?
3. Que peut-on dire d'un sommet i qui a une valeur rang(i)$= k$ avec $k > 2$?
4. Soit $kmax$ la plus grande valeur de rang, à quoi cette valeur correspond-t-elle dans le graphe G ?
5. En raisonnant sur la base de la fonction rang, dans quel cas sait-on qu'un graphe admet un unique ordre topologique ?

3.5 Fermeture transitive d'un graphe

Considérons un 1-graphe orienté connexe $G = (X, U)$. La **fermeture transitive** d'un graphe G est un graphe $F = (X, U_F)$ défini sur le même ensemble de sommets X et dont l'ensemble des arcs est défini par :

$$U_F = \{(i, j) \in X \times X : \exists \mu[i, j]\}$$

Le problème de la fermeture transitive d'un graphe a été étudié par de très nombreux auteurs. Une des premières approches suggérées pour le résoudre est la technique dite des multiplications matricielles, consistant à élever à la puissance $n-1$ la matrice d'adjacence du graphe. Il existe une autre famille de méthodes où l'on se ramène à la fermeture transitive d'un graphe sans circuit en décomposant le problème en trois étapes : dans une première étape, on détermine les composantes fortement connexe de G; dans une deuxième étape, on détermine la fermeture transitive du graphe réduit G^r induit par les composantes fortement connexes, lequel est sans circuit; et enfin dans une troisième étape, on déduit la fermeture transitive de G de la fermeture transitive de G^r.

Remarque : Si $G = (X, U)$ est fortement connexe, sa fermeture transitive est un 1-graphe plein sans boucle, c'est-à-dire $\forall (i,j) \in X \times X,\ (i,j) \in U_F$ et $(j,i) \in U_F$.

L'algorithme de Kosaraju-Sharir nous permet de mener à bien l'étape 1 de l'algorithme. Les étapes 2 et 3 peuvent se dérouler comme suit :

Etape 2. Recherche de la fermeture transitive du graphe $G^r = (X^r, U^r)$ (sans circuit) d'ordre q.
Pour cela, il faut en premier lieu définir un ordre topologique sur les sommets de G^r. Les sommets sont alors numérotés de 1 à q selon cet ordre topologique. Pour tout sommet v de X_r, L^v désigne la liste des sommets i tq il existe un chemin allant de i à v (liste des ascendants de v). Pour chaque sommet v, la liste L^v est initialisée par la liste des prédécesseurs de v. Puis, pour chaque sommet v allant de 1 à q, on établit sa liste des ascendants de la façon suivante :

$$L^v \leftarrow L^v \bigcup_{i \in \Gamma^-(v)} L^i$$

Comme les sommets sont examinés dans l'ordre topologique, lorsque le sommet v est examiné, les listes définitives L^i de tous ses prédecesseurs directs i ont déjà été constituées.

La fermeture transitive $F^r = (X^r, U^{F_r})$ de G^r est constituée des arcs suivants : $\forall v \in X^r\ \forall i \in L^v, (i,v) \in U^{F_r}$.

Etape 3. Déduire la fermeture transitive de G de la fermeture transitive de G^r.

(a) Pour chaque composante fortement connexe X_i de cardinal n_i, établir la liste des $n_i(n_i-1)$ arcs du sous-graphe plein sans boucle défini sur X_i.

(b) Si k et l sont deux sommets de G^r (correspondant aux composantes fortement connexes X_k et X_l de G) tels que l'arc (k,l) appartienne à la fermeture transitive de G^r, alors établir la liste des $n_k \times n_l$ arcs de la forme (i,j) avec $i \in X_k$ et $j \in X_l$.

4 Cheminement dans les graphes

Les problèmes de cheminement dans les graphes (en particulier la recherche d'un plus court chemin) comptent parmi les plus classiques de la théorie des graphes et les plus importants dans leurs applications. Le problème du **plus court chemin** (pcch) peut être posé de la façon suivante : étant donné un graphe $G = (X, U)$, on associe à chaque arc $u = (i, j)$ un nombre réel, noté $l(u)$ ou l_{ij}, appelé la longueur de l'arc. Le problème du pcch entre deux sommets i_0 et j_0 du graphe consiste à déterminer, parmi tous les chemins allant de i_0 à j_0 celui, noté μ^* dont la longueur totale :

$$v(\mu^*[i_0, j_0]) = \sum_{u \in \mu^*[i_0, j_0]} l(u)$$

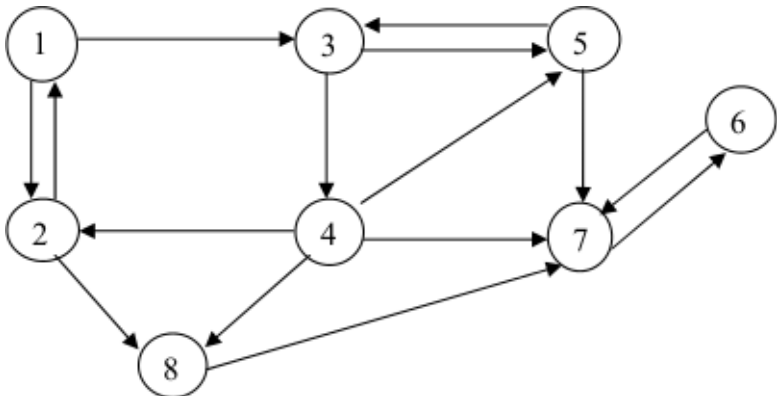
soit minimale.

On peut utiliser le parcours en largeur d'abord pour déterminer les pcch au sens du nombre d'arcs, avec $\forall u \in U, l(u) = 1$.

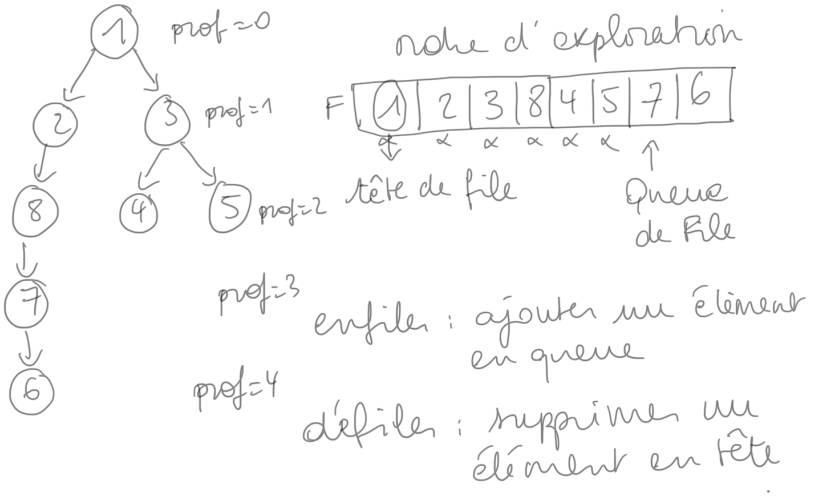
4.1 Parcours en Largeur d'un graphe

Le principe est le même : on explore une et une seule fois chacun des sommets du graphe. Il s'agit alors, partant d'un sommet i_0, d'explorer tous ses successeurs non explorés avant d'explorer les successeurs de ses successeurs.

Exemple : Reprenons le graphe ci-dessous :



Le parcours en largeur d'abord est décrit par l'arborescence suivante :



On sait que le plus court chemin en nbre d'arcs pour aller de 1 à 8 est de 2.

En appelant `exploreLargeur(G, 1, etat ,FileVide(), ∅)`, on obtient une arborescence $\mathcal{A}_1$ qui contient l'ensemble des sommets j tq $\exists \mu[1, j]$ et $prof(j)$ est la longueur du plus court chemin allant de 1 à j.

Une file (queue en anglais) est un type abstrait de données qui correspond à des valeurs enregistrées selon le principe du premier entrée / premier sortie (comme une file d'attente) - FIFO en anglais : first in / first out.

En pseudo code, on définit :
- la tête de la file : premiere valeur
- la queue de la file : dernière valeur
- FileVide() : retourne une file ne contenant aucune valeur
- F.enfiler(v) : ajoute la valeur v en queue de F (dernière entrée)
- F.défiler() : retourne la valeur v en tête de F et supprime cette valeur

```
FONCTION exploreLargeur(graphe, sommet, etat, F, A)
    etat[sommet] <- 'exploré'
    F.enfiler(sommet)
    tant que F non vide faire
        v <- F.defiler()
        pour tout j successeur de v
            si etat[j]=='inexploré' alors
                etat[j] <- 'exploré'
                A <- A + (v,j)
                F.enfiler(j)
```

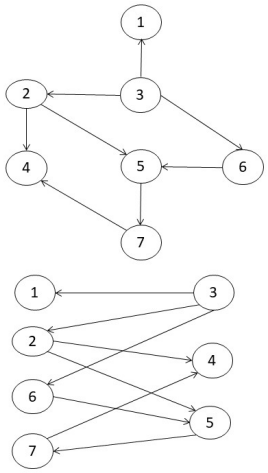
```
Algorithme BFS
    lire G, n
    A <- ensemble vide
    F <- FileVide()
    pour i allant de 1 à n
        etat[i] <- 'inexploré'
    pour i allant de 1 à n
        si etat[i] == 'inexploré' alors
            exploreLargeur(G, i, etat, F, A)
```

La complexité de l'algorithme est également en $O(n) + O(m)$.

BFS permet également de déterminer le nombre de connexité. Il suffit d'appliquer BFS sur la version non orienté de G. On ajoute une variable `nbConnexite` initialisée a 0 en début d'algo et dans la boucle while, avant d'appeler `exploreLargeur()`, on incrémente `nbConnexite` d'une unité.

En python l'enregistrement d'une file avec une variable de type list n'est pas efficace car, lorsqu'on défile on enlève la valeur en position 0 et il faut décaler toutes les autres valeurs vers la gauche. Il existe en Python un module appelé collections qui contient le type deque : conteneur se comportant comme une liste avec des ajouts et retraits rapides à chaque extrémité.

Autre application du parcours en profondeur d'abord Un graphe $G = (X, E)$ est dit biparti s'il est possible de partitionner son ensemble de sommets X en deux sous-ensembles X_1 et X_2 disjoints tq $\forall\{i, j\} \in E$, si $i \in X_1$ alors $j \in X_2$, et si $i \in X_2$ alors $j \in X_1$.

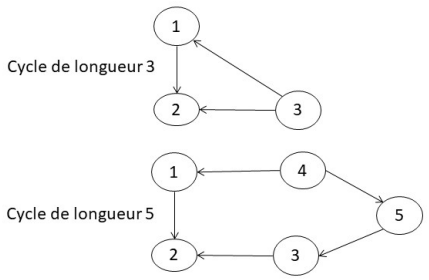


Ce graphe est biparti.
Appliquer l'algo de parcours en largeur sur la version non orienté

1
|
3
| \
2 6
| \
4 5
|
7

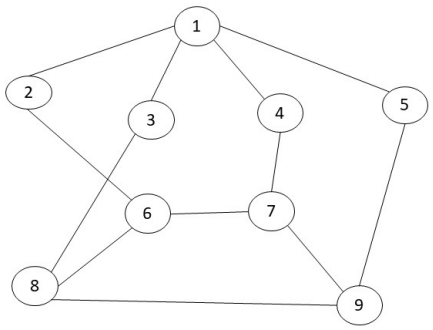
X1 = {1, 2, 6, 7} X2 = {3, 4, 5}

Remarque : Si G contient un cycle de longueur impair, alors il n'est pas biparti !



Dans le cycle de longueur 5, si on met 1 dans X1 alors on met 2 et 4 dans X2. 5 doit aller dans X1. 3 doit également aller dans X1 => arc reliant deux sommets d'un même sous-ensemble !

G ci-dessous n'est pas biparti car cycle 1-2-6-7-4



Réalisons un parcours en largeur d'abord sur la graphe G représenté ci-dessus :

```
   1
/  |  \  \
2  3  4  5
|  |  |  |
6  8  7  9
```

A la profondeur 2, deux sommets sont reliés : cycle impair.

=> une arête retour dont les deux extrémités ont la même profondeur dans l'arborescence décrivant BFS(G) = cycle impair !

Propriété : Si G ne contient pas de cycle de longueur impaire alors il est biparti.

Lorsqu'on cherche un pcch entre deux sommets, on doit déterminer d'autres pcch entre i_0 et d'autres sommets du graphe. Aussi, les algorithmes existant se divisent en deux catégories : ceux dans lesquels on recherche un pcch d'un sommet spécifié i_0 à tous les autres sommets du graphe ; ceux qui procèdent directement de la recherche de tous les pcch entre i et j pour tous les couples (i, j) du graphe.

Il existe un grand nombre d'algorihmes permettant de déterminer un pcch d'un sommet particulier i_0 à tous les autres sommets du graphe. Les plus efficaces d'entre eux réalisent un marquage des sommets c'est-à-dire qu'à chaque sommet i est associé une marque $\lambda(i)$ représentant à la fin de l'algorithme la longueur du pcch allant de i_0 à i.

4.2 Plus courts chemins d'origine fixée dans un graphe avec longueurs non négatives : algorithme de Dijkstra

Soit $G = (X, U)$ un graphe dont les arcs sont munis de longueur réelles positives ou nulles. On cherche les pcch de i_0 à tous les autres sommets du graphe.

L'algorithme de Dijkstra procède en $n-1$ itérations. A l'initialisation, $\lambda(i_0) \leftarrow 0$ et $\lambda(i) \leftarrow \infty$ pour tout $i \neq i_0$. A une itération quelconque de l'algorithme, l'ensemble des sommets est partagé en deux sous-ensembles S et $X \setminus S$. Le sous-ensemble S contient l'ensemble des sommets définitivement marqués c'est-à-dire les sommets i pour lesquels la marque $\lambda(i)$ représente effectivement la longueur d'un pcch allant de i_0 à i (à l'initialisation $S \leftarrow \{i_0\}$). $X \setminus S$ contient les sommets i ayant une marque provisoire vérifiant :

$$\lambda(i) = \min_{k \in S \cap \Gamma^-(i)} \{\lambda(k) + l_{ki}\}$$

A chaque itération, on sélectionne le sommet i de plus petite marque provisoire, on l'inclut dans l'ensemble S des sommets définitivement marqués et on remet à jour les marques de ces successeurs non définitivement marqués de la façon suivante :

$$\forall j \in \Gamma^+(i) \setminus S,\ \lambda(j) = \min\{\lambda(j); \lambda(i) + l_{ij}\}$$

Lorsque tous les sommets du graphe sont dans l'ensemble S, les marques représentent les longueurs des pcch allant de i_0 à tous les autres sommets du graphe.

Comme à chaque itération, on attribue une marque définitive à un nouveau sommet, on détermine $n-1$ pcch entre i_0 et les autres sommets du graphe en au plus $n-1$ itérations .

$\lambda(i_0) \leftarrow 0$
$\lambda(i) \leftarrow +\infty\ \forall i \in X \setminus \{i_0\}$
$p(i) \leftarrow i\ \forall i \in X$
$S \leftarrow \{i_0\}$
$i \leftarrow i_0$
TantQue $S \neq X$ Faire
  Pour tout $j \in \Gamma^+(i) \setminus S$ Faire
    Si $\lambda(j) > \lambda(i) + l_{ij}$ alors
      $\lambda(j) \leftarrow \lambda(i) + l_{ij}$
      $p(j) \leftarrow i$
    FinSi
  FinPour
  Sélectionner $i \in X \setminus S$ tel que $\lambda(i) = \min_{j \in X \setminus S} \lambda(j)$
  $S \leftarrow S \cup \{i\}$
FinTantQue

L'algorithme est basé sur le théorème suivant.

Théorème : Si i est un sommet de $X \setminus S$ de marque provisoire $\lambda(i)$ minimale :

$$\lambda(i) = \min_{j \in X \setminus S} \lambda(j)$$

alors $\lambda(i)$ est la longueur d'un pcch chemin allant de i_0 à i, notée $v(\mu^*[i_0, i])$.

Preuve par récurrence du théorème :

Au rang 0 : $\lambda(i_0) = 0 = v(\mu^*[i_0, i_0])$ et i_0 est le seul sommet dans S.
Au rang 1 : Soit $i \in \Gamma^+(i_0)$ tel que $\lambda(i) = \min_{j \in X \setminus \{i_0\}}\{\lambda(j)\}$.
$\lambda(i) = l_{i_0 i} = v(\mu^*[i_0, i])$ car tout autre chemin allant de i_0 à i comportera au moins deux arcs et sera donc de valeur minimale : $v = \min_{j \in \Gamma^+(i_0) \setminus \{i\}}\{\lambda(j) + q\}$. Et, comme $q \geq 0$, et $\min_{j \in \Gamma^+(i_0) \setminus \{i\}}\{\lambda(j)\} > \lambda(i)$, on a $v \geq \lambda(i)$. En csq, i peut être ajouté à S
On suppose que la propriété est vraie au rang $k-1$. Ainsi, $\forall s \in S, \lambda(s) = v(\mu^*[i_0, s])$.
Au rang k : Soit i tel que $\lambda(i) = \min_{j \in X \setminus S}\{\lambda(j)\}$
Raisonnons par l'absurde et supposons qu'il existe un chemin de

$$\mu'[i_0, i] = \{i_0, \ldots, s, r, \ldots, i\}$$

de longueur $v(\mu'[i_0, i])$ tel que $v(\mu'[i_0, i]) < \lambda(i)$. Soit s le premier sommet appartenant à S rencontré en remontant le chemin μ' de i vers i_0. Comme la propriété est supposée vraie au rang $k-1$, on a $\lambda(s) = v(\mu^*[i_0, s])$ et :
- la marque de $r \notin S$ est telle que : $\lambda(r) \leq \lambda(s) + l_{sr} = v(\mu^*[i_0, s]) + l_{sr}$
- comme μ^* est le chemin optimal, $\lambda(r) \leq v(\mu'[i_0, s]) + l_{sr}$
- comme toute les longueurs sont positives ou nulles, $v(\mu'[i_0, s]) + l_{sr} \leq v(\mu'[i_0, s]) + l_{sr} + v(\mu'[r, i])$
Or, $v(\mu'[i_0, s]) + l_{sr} + v(\mu'[r, i]) = v(\mu'[i_0, i])$ qui par hypothèse est $< \lambda(i)$
En csq, $\lambda(r) < \lambda(i)$, c'est donc le sommet non marqué r qui aurait dû être sélectionné au rang k et non le sommet i. On aboutit donc à une contradiction.

Complexité : A chaque itération, on sélectionne le sommet j de plus petite marque en $O(n)$ opérations dans le pire cas, et on remet à jour les marques des successeurs de j en $O(d^+(j))$ opérations. En tout il y a n itérations pour marquer tous les sommets du graphe. La complexité totale est donc en $O(n^2) + O(m) \approx O(n^2)$.

Remarquons que lorsque le graphe est peu dense, le terme $O(n^2)$ l'emporte sur $O(m)$ et, l'opération la plus coûteuse consiste à rechercher le sommet de plus petite marque. Pour diminuer la complexité de cette recherche, on peut utiliser une structure de données appelée tas.

4.3 Mise en oeuvre de l'algorithme de Dijkstra pour les graphes peu denses

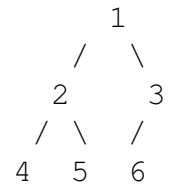
4.3.1 Présentation de la structure de données de tas (heap)

Un **arbre binaire** est une arborescence telle que :
- chaque noued a un père et un seul
- un seul noeud, appelé racine, n'a pas de père
- chaque noeud a 0, 1 ou 2 fils (fils gauche et fils droit)
- un noeud sans fils est appelé une feuille.

Dans un **arbre binaire équilibré**, les seules feuilles manquantes doivent se trouver à droite des feuilles situées au plus bas niveau de l'arbre).

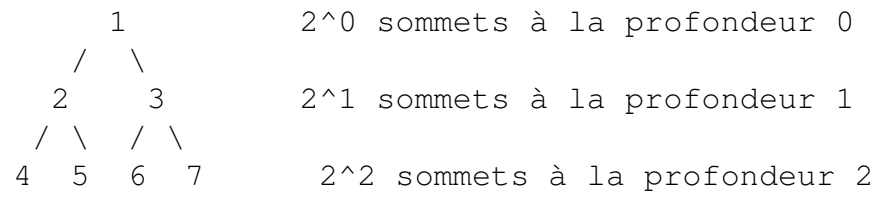
La profondeur d'un noeud i est la longueur (en nombre d'arcs) du chemin allant de la racine à i. La hauteur d'un arbre est la profondeur maximum.

Exemple : Arbre binaire équilibré de profondeur 2 contenant 6 noeuds.



Un arbre binaire est complet si tous les sommets ont deux fils sauf les feuilles.

Exemple : Arbre binaire complet de profondeur 2.



Un arbre binaire complet de hauteur h comporte : $n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$ sommets. A la profondeur p, on a 2^p sommets.

Rappel : suite géométrique de raison q et de 1er terme u_0 :

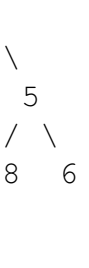
$$\sum_{i=0}^{n} u_0 q^i = \frac{u_0(q^{n+1} - 1)}{q - 1}$$

Donc quand $u_0 = 1$ et $q = 2$ on a :

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

Dans un **arbre binaire valué partiellement ordonné** ou minimier, les noeuds sont valués et la valuation d'un noeud est inférieure ou égale à celle de chacun de ses fils.

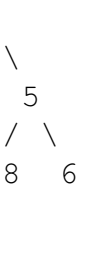
Exemple : Minimier de hauteur 3 comportant 10 noeuds valués



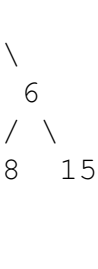
Le sommet de plus petite valuation est le sommet racine. On peut donc le trouver en $O(1)$ opération.

Si on le supprime, on détruit la structure d'arbre. Pour la recomposer, il suffit de prendre la feuille la plus à droite du niveau le plus bas et on la place temporairement à la racine. Puis, il faut pousser cet élément aussi bas que possible en l'échangeant avec celui de ses fils ayant la plus petite valuation inférieure. On s'arrête lorsque l'élément est, soit, devenu une feuille, soit, a ses fils de plus grande valuation. Le nombre d'opérations élémentaires dépend donc de la hauteur de l'arbre.

Exemple : Suppression de la racine : etape 1 pour recomposer l'arbre



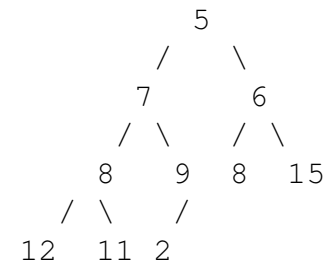
Etape 2 pour recomposer l'ordre partiel



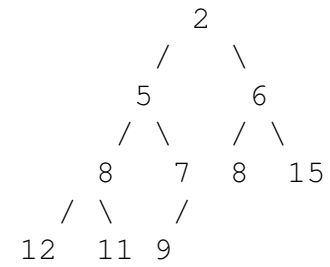
Pour insérer un nouvel élément dans l'arbre, on réalise la procédure " inverse ". On commence par placer l'élément en question aussi loin que possible à gauche au

plus bas niveau de l'arbre. Puis, on le pousse aussi haut que possible dans l'arbre de la façon suivante : si son père à une valuation supérieure à la sienne, il faut les échanger, et on réitère les comparaisons jusqu'à ce que l'élément inséré se trouve à la racine, ou bien, ait une priorité inférieure à celle de son père. Le nombre d'opérations élémentaires dépend donc également de la hauteur de l'arbre.

Exemple : Insertion du noeud de valeur 2 : Etape 1



Etape 2 :



Le **tas** est une structure de données pouvant être utilisée pour représenter en machine un arbre binaire équilibré. Soit un arbre contenant n sommets. Dans le tas, on utilise les n premières cellules d'un tableau unidimensionnel T de la façon suivante : les noeuds remplissent les cellules $T[0], T[1], \ldots, T[n-1]$ niveau par niveau à partir du haut, et à l'intérieur d'un même niveau de la gauche vers la droite. Ainsi, le fils gauche, s'il existe, du sommet $T[i]$ se trouve en $T[2i+1]$ et le fils droit, s'il existe, en $T[2i+2]$; le père de $T[i]$ se trouve dans T à l'indice $\lfloor \frac{i-1}{2} \rfloor$.

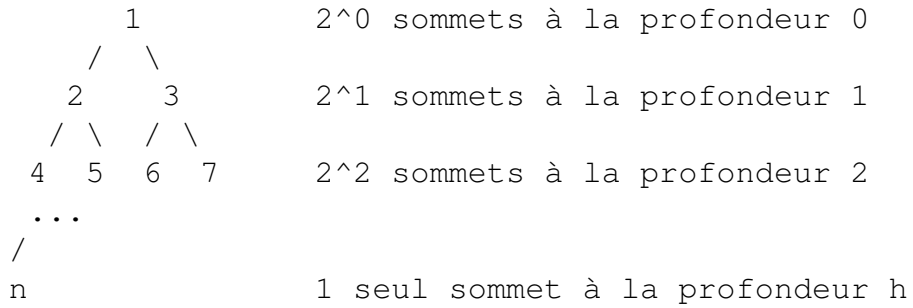
Exemple : Le tas T associé au dernier minimier de l'exemple

T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]
2	5	6	8	7	8	15	12	11	9

Pour trouver les fils du noeud valué par 8 stocké à l'indice i=3 du tas, il suffit d'aller à l'indice $2i+1 = 7$ du tas pour trouver le fils gauche du noeud 8 et à l'indice $2i+2 = 8$ pour trouver le fils droit.
Pour trouver le père du noeud valué par 7 stocké à l'indice $i = 4$ du tas, il suffit d'aller à l'indice $\lfloor \frac{4-1}{2} \rfloor = 1$ du tas.

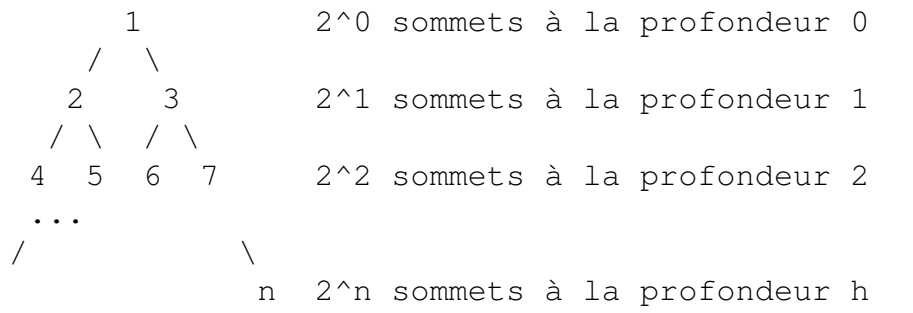
Dans un arbre binaire équilibré, la hauteur h dépend du nombre n de noeuds, on a : $\log_2 n - 1 < h \leq \log_2 n$.

En effet, un arbre binaire équilibré contenant n sommets de hauteur h peut être de la forme



=> $\sum_{i=0}^{h-1} 2^i + 1 = 2^h - 1 + 1 = 2^h$
Donc $n \geq 2^h$ => $\log_2 n \geq h$

Et, l'arbre binaire complet équilibré de hauteur h est



=> $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$
Donc $n \leq 2^{h+1} - 1$ => $n < 2^{h+1}$ => $\log_2 n < h + 1$ => $h > \log_2 n - 1$

Ce qui implique : $\log_2 n - 1 < h \leq \log_2 n$. En conséquence, $h = \lfloor \log_2 n \rfloor$. Et tout chemin entre la racine et une feuille du niveau h comporte $\lfloor \log_2 n \rfloor$ arêtes.

La suppression de la racine a une complexité en $O(log_2 n)$ car aucun chemin de l'arbre ne contient plus de $log_2 n$ arcs.

L'insertion d'un nouvel élément a également une compléxité en $O(log_2 n)$.

4.3.2 Algorithme de Dijkstra en utilisant une structure de Tas

Les sommets i de G et leur marque $\lambda(i)$ sont stockées dans un tas ce qui permet de trouver le sommet de marque minimale en $O(1)$ opération.

Dans cette nouvelle version de l'algorithme de Dijkstra, on fait appel aux fonctions suivantes :

insere(i, T) : insère le sommet i dans le tas T
min(T) : retourne et supprime le sommet racine (de plus petite valuation) de T et recompose la structure
diminue$(valeur, i, T)$: modifie la valuation du sommet i dans T (elle passe à $valeur$) et recompose la structure

Le tas est constitué des sommets ayant des marques temporaires finies.

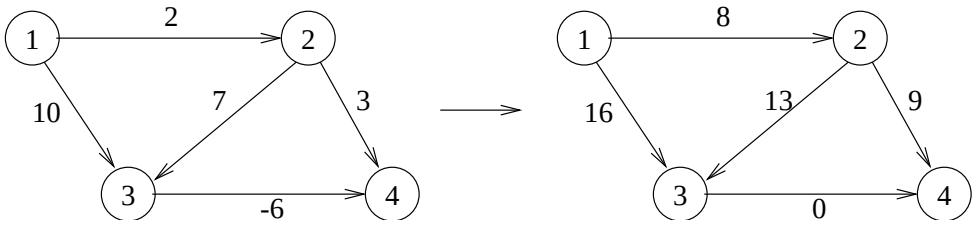
$T \leftarrow \emptyset$
$\lambda(i_0) \leftarrow 0$
$\lambda(i) \leftarrow +\infty \ \forall i \in X \setminus \{i_0\}$
$p(i) \leftarrow i \ \forall i \in X$
insere(i_0, T)
Tant que $T \neq \emptyset$ faire
  $i \leftarrow$ min(T)
  $S \leftarrow S \cup \{i\}$
  Pour tout $j \in \Gamma^+(i) \setminus S$ faire
    $v = \lambda(i) + l_{ij}$
    Si $\lambda(j) > v$ alors
      Si $\lambda(j) = +\infty$ alors
        $\lambda(j) \leftarrow v$
        $p(j) \leftarrow i$
        insere(j, T)
      Sinon
        $\lambda(j) \leftarrow v$
        $p(j) \leftarrow i$
        diminue(v, j, T)
      FinSi
    FinSi
  FinPour
FinTantque

A chaque itération, on sélectionne le sommet i de plus petite marque et on recompose le tas : la complexité est en $O(log_2 n)$. Puis, on remet à jour les marques de tous les successeurs du sommet i sélectionné en $O(d^+(i) \times log_2 n)$. Comme il y a n itérations, on a une complexité totale de $O(nlog_2 n) + O(mlog_2 n) \approx O(mlog_2 n)$.

4.4 Plus court chemin d'origine fixée dans un graphe avec longueurs quelconques

Remarque préliminaire

Soit un graphe G présentant sur certains arcs une valeur négative. Envisageons de rendre les valeurs toutes positives ou nulles en ajoutant à chaque valeur une constante.



Ceci change la nature du problème puisque, comme on le voit dans l'exemple ci-dessus, le plus court chemin allant de 1 à 4 dans le graphe initial (de valeur 3) n'est plus optimal dans le graphe transformé.

4.4.1 La classe des graphes sans circuit : algorithme de Bellman

Soit $G = (X, U)$ un graphe sans circuit dont les arcs sont munis de longueur réelles quelconques. On cherche les pcch allant de i_0 à tous les autres sommets du graphe.

Lorsque le graphe ne présente pas de circuit, il faut déterminer une numérotation des sommets allant de 1 à n qui constitue un ordre topologique (le sommet de départ i_0 ayant le numéro 1) et marquer les sommets dans cet ordre.

$\lambda(1) \leftarrow 0$
$p(1) \leftarrow 1$
Pour $j \leftarrow 2$ à n faire
  $\lambda(j) = \min_{i \in \Gamma^-(j)} \lambda(i) + l_{ij}$
  $p(j) = \text{argmin}_{i \in \Gamma^-(j)} \lambda(i) + l_{ij}$
FinPour

4.4.2 Algorithme général de Ford-Bellman

Condition d'optimalité

Un ensemble de valeurs $\lambda^*(i)$ pour $i = 1, \ldots, n$, avec $\lambda^*(i_0) = 0$, représente les longueurs des pcch allant de i_0 aux autres sommets du graphe si et seulement si :

$$\forall (i,j) \in U : \ \lambda^*(j) \leq \lambda^*(i) + l_{ij}$$

Soit $G = (X, U)$ un graphe dont les arcs sont munis de longueur réelles quelconques. On cherche les pcch de i_0 à tous les autres sommets du graphe.

Dans l'algorithme de Ford-Bellman, les marques $\lambda(i)$ des sommets sont modifiées itérativement de façon à converger vers la condition d'optimalité.

Pour parcourir la liste des arcs du graphe et converger vers la condition d'optimalité, on regarde pour chaque sommet i l'ensemble de ses prédécesseurs. Et, à une itération k donnée, on ne va pas s'intéresser à tous les sommets du graphe mais seulement à ceux dont la marque a été modifiée au cours de l'itération précédente. L'algorithme calcule donc à chaque itération k un ensemble de marques, notées $\lambda^k(j)$ pour tout $j \in X$. On note $M = \{j \in X | \lambda^k(j) < \lambda^{k-1}(j)\}$, l'ensemble des sommets dont les marques ont été modifiées à l'iteration k, et seuls les sommets appartenant à $\Gamma^+(M)$ peuvent voir leurs marques modifiées au cours de l'itération $k+1$. En fait les marques $\lambda^k(j)$ ont une interprétation très précise : c'est la valeur du meilleur chemin de i_0 à j ne contenant pas plus de k arcs. Ainsi, en l'absence de circuit absorbant dans le graphe, l'algorithme termine nécessairement à l'issue de l'itération n car tout chemin élémentaire a une longueur maximale égale à $n-1$. Si une ou plusieurs marques sont modifiées à l'itération n, cela signifie que le graphe présente un circuit de valeur négative (car il existe un pcch de longueur $> n-1$ qui emprunte donc nécessairement un circuit de valeur < 0 !). L'algorithme de Ford-Bellman peut donc s'écrire comme suit :

$k \leftarrow 0$
$\lambda^0(i_0) \leftarrow 0$
$\lambda^0(i) \leftarrow +\infty \; \forall i \in X \setminus \{i_0\}$
$p(i) \leftarrow i \; \forall i \in X \setminus \{i_0\}$
$M \leftarrow \{i_0\}$
TantQue $k \leq n$ et $M \neq \emptyset$ faire
    $k \leftarrow k+1$
    $M' \leftarrow \emptyset$
    Pour tout $j \in \Gamma^+(M)$ faire
        $\lambda^k(j) \leftarrow \min\{\lambda^{k-1}(j); \lambda^{k-1}(i) + l_{ij}, \; i \in \Gamma^-(j) \cap M\}$
        Si $\lambda^k(j) < \lambda^{k-1}(j)$ Alors
            $M' \leftarrow M' \cup \{j\}$
            $p(j) \leftarrow i^*$ avec $i^* \in \Gamma^-(j) \cap M$ tel que $\lambda^k(j) = \lambda^{k-1}(i^*) + l_{i^*j}$
        FinSi
    FinPour
    $M \leftarrow M'$
FinTantQue
Si $M \neq \emptyset$ Alors $\exists$ un circuit de valeur négative FinSi

A chaque itération, on remet à jour, dans le pire cas, les marques de tous les sommets en regardant l'ensemble des arêtes : $O(m)$ opérations. Le nombre maximal d'itérations étant n, la complexité totale est en $O(nm)$.

4.5 Plus courts chemins entre toutes les paires de sommets : algorithme de Floyd

Soit la matrice $A = \{a_{ij}\}$ de taille $n \times n$ avec :

$$a_{ij} = \begin{cases} 0 \text{ si } i = j \\ l_{ij} \text{ si } (i,j) \in U \\ +\infty \text{ sinon} \end{cases}$$

L'algorithme de Floyd permet de calculer les pcch entre tous les couples de sommets de la façon suivante. A la première itération, on cherche le pcch entre chaque couple (i, j) passant éventuellement par le sommet 1 ; à l'itération l (avec $l > 1$), on cherche le pcch entre chaque couple (i, j) passant par des sommets d'indice inférieur ou égal à l. Une description formelle de l'algorithme est donnée ci-dessous.

Pour $l = 1$ à n Faire
    Pour $i = 1$ à n Faire
        Pour $j = 1$ à n Faire
            $a_{ij} = \min\{a_{ij}, a_{il} + a_{lj}\}$
        FinPour
    FinPour
FinPour

La complexité totale est en $O(n^3)$.

5 Flots dans les graphes

Le problème des flots dans les graphes concerne la circulation de matière sur les arcs. Parmi les nombreuses applications qui relèvent de ce problème, on trouve le transport de marchandises entre différents points, les télécommunications dans les réseaux,...

5.1 Définitions

Un **réseau de transport** $R = (N, A)$ est un 1-graphe connexe tel que :

— il existe une **source** (il s'agit d'un sommet racine) notée s, qui représente la point d'entrée de la matière,

— il existe un **puits** (il s'agit d'un sommet antiracine) notée t, qui représente le point de sortie de la matière,

— chaque arc $u = (i, j) \in A$ est muni d'une **capacité** c_u entière positive ou nulle (et éventuellement un coût d_u entier positif ou nul).

Un **flot** $\phi = (\phi_u)$ dans $R = (N, A)$ est un vecteur de R^m (dont l'élément générique ϕ_u est appelé **flux** sur l'arc $u = (i, j)$) tel que :

(i) $\forall u = 1, \ldots, m, \phi_u \geq 0$

(ii) En tout sommet $i \in N \setminus \{s, t\}$, la première loi de Kirchhoff est vérifiée (cette loi est également connue sous le nom de loi de conservation aux noeuds) :

$$\sum_{u \in \omega^+(i)} \phi_u = \sum_{u \in \omega^-(i)} \phi_u$$

avec $\omega^+(i)$ est l'ensemble des arcs ayant le sommet i comme extrémité initiale et $\omega^-(i)$ est l'ensemble des arcs ayant i comme extrémité terminale.

Un flot compatible ϕ dans un réseau de transport $R = (N, A)$ est un flot tel que :

$$\forall u \in A,\ \phi_u \leq c_u$$

La **valeur** f_ϕ d'un flot **compatible** ϕ dans un réseau de transport est égale à la somme des flux sortant de s, elle-même égale à la somme des flux entrant en t :

$$f_\phi = \sum_{u \in \omega^+(s)} \phi_u - \sum_{u \in \omega^-(s)} \phi_u = \sum_{u \in \omega^-(t)} \phi_u - \sum_{u \in \omega^+(t)} \phi_u$$

Le problème du **flot maximum** dans un réseau de transport s'énonce comme suit : déterminer quelle est la valeur maximale du flot qu'il est possible de faire parvenir de la source s au puits t. On supposera par la suite qu'il n'existe pas de chemin de capacité infinie entre s et t dans R car, si un tel chemin existe, la solution n'est pas bornée supérieurement et la valeur maximale du flot vaut $+\infty$.

Le problème de flot maximal de s à t peut se formuler de la façon suivante :

$$\begin{cases} \max f_\phi \\ \sum_{u\in\omega^+(i)} \phi_u - \sum_{u\in\omega^-(i)} \phi_u = 0 \ \forall i \in N \setminus \{s,t\} \\ \sum_{u\in\omega^+(s)} \phi_u - \sum_{u\in\omega^-(s)} \phi_u = f_\phi \\ \sum_{u\in\omega^+(t)} \phi_u - \sum_{u\in\omega^-(t)} \phi_u = f_\phi \\ 0 \leq \phi_u \leq c_u \ \forall u \in A \end{cases}$$

5.2 Graphe d'écart et chemin augmentant

Il est simple de trouver une solution (non nécessairement optimale) : on fait circuler une quantité de flux entre la source et le puits sur les chemins comportant de la capacité résiduelle.

Un flot compatible ϕ dans un réseau de transport $R = (N, A)$ est un flot **complet** si tous les chemins allant de s à t comportent au moins un arc saturé.

Soit ϕ un flot compatible sur R. Le **graphe d'écart** associé à ϕ sur R est le graphe $R^e(\phi) = [N, A^e(\phi)]$ défini comme suit : $\forall u = (i,j) \in A$

— Si $\phi_u < c_u$, $(i,j) \in A^e(\phi)$ avec la capacité (résiduelle) $c'_{(i,j)} = c_u - \phi_u$

— Si $\phi_u > 0$, $(j,i) \in A^e(\phi)$ avec la capacité $c'(j,i) = \phi_u$.

Dans le premier cas, un arc est valué par sa capacité résiduelle, c'est-à-dire l'augmentation de flux possible. Dans le second cas, l'idée est de diminuer le flux sur l'arc (i,j) en faisant passer une quantité de flux sur l'arc (j,i).

Soit ϕ un flot compatible sur R et $R^e(\phi) = [N, A^e(\phi)]$ le graphe d'écart associé. Soit μ un chemin allant de s à t dans $R^e(\phi)$ et $\delta = min_{u\in\mu} c'_u$. Ce chemin est appelé **chemin augmentant** car il est possible d'augmenter la valeur du flot sur R de δ de la façon suivante $\forall (i,j) \in \mu$:

— si $u = (i,j) \in A$, alors $\phi_u \leftarrow \phi_u + \delta$,

— si $u = (j,i) \in A$, alors $\phi_u \leftarrow \phi_u - \delta$.

Un flot ϕ compatible sur R n'est de valeur maximale s'il existe un chemin allant de s à t dans $R^e(\phi) = [N, A^e(\phi)]$.

5.3 Recherche d'un flot maximal

5.3.1 Algorithme basé sur le graphe d'écart

Soit un réseau R, un flot compatible ϕ dans R et le graphe d'écart $R^e(\phi)$. Pour déterminer un flot maximum, l'algorithme basé sur le graphe d'écart consiste, à chaque itération, à chercher un chemin μ allant de s à t dans $R^e(\phi)$. Si un tel chemin existe, on augmente le flot ϕ de la quantité $\delta = \min_{u\in\mu} c'_u$. Sinon l'algorithme termine et ϕ

est le flot de valeur maximale.

$\phi \leftarrow 0$
Tant que $R^e(\phi)$ contient un chemin de s à t faire
  Identifier un chemin μ de s à t
  $\delta = \min_{u \in \mu} c'_u$
  Augmenter de δ unités le flot ϕ sur ce chemin
  Mettre à jour $R^e(\phi)$
FinTantque

Cet algorithme ne précise pas de quelle façon déterminer un chemin μ de s à t. Dans la section suivante, nous présentons un algorithme de marquage qui, sans passer par le graphe d'écart, permet d'exhiber un chemin augmentant dans $R^e(\phi)$ (ou de façon équivalente une chaîne augmentante dans R) en travaillant directement sur R.

5.3.2 Algorithme de marquage de Ford-Fulkerson

Faire
  marque(i) $\leftarrow 0 \; \forall i \in N$
  pred(i) $\leftarrow i \; \forall i \in N$
  marque(s) $\leftarrow +\infty$ et LISTE $\leftarrow \{s\}$
  Tant que LISTE $\neq \emptyset$ et marque(t) $= 0$ faire
    Sélectionner $i \in$ LISTE
    LISTE $\leftarrow$ LISTE $\setminus \{i\}$
    Pour chaque arc $u = (i, j) \in \omega^+(i)$ faire
      Si marque(j)$= 0$ et $c_u > \phi_u$ alors
        marque(j) $\leftarrow c_u - \phi_u$ (marquage de type +)
        pred(j) $\leftarrow i$
        LISTE $\leftarrow$ LISTE $\cup \{j\}$
      FinSi
    FinPour
    Pour chaque arc $u = (j, i) \in \omega^-(i)$ faire
      Si marque(j)$= 0$ et $\phi_u > 0$ alors
        marque(j) $\leftarrow -\phi_u$ (marquage de type -)
        pred(j) $\leftarrow i$
        LISTE $\leftarrow$ LISTE $\cup \{j\}$
      FinSi
    FinPour
  FinTantque
  Si marque(t)$\neq 0$ alors **augmenter**
Tant que marque(t) $\neq 0$

Procédure augmenter
Identifier la chaîne μ à l'aide de pred(t) : $\mu = (s, i_0, i_1, \ldots, i_k, t)$
$\delta \leftarrow \min_{i \in \mu} |$ marque(i) $|$
$\phi_{(s,i_0)} \leftarrow \phi_{(s,i_0)} + \delta$
Pour $j \leftarrow 1$ à k faire
    Si marque(i_j) < 0 alors
        $\phi_{(i_{j-1},i_j)} \leftarrow \phi_{(i_{j-1},i_j)} - \delta$
    Sinon
        $\phi_{(i_{j-1},i_j)} \leftarrow \phi_{(i_{j-1},i_j)} + \delta$
    FinSi
FinPour
$\phi_{(i_k,t)} \leftarrow \phi_{(i_k,t)} + \delta$

5.4 Théorème de Flot maximum et coupe minimum

Dans un réseau de transport $R = (N, A)$, soit S un sous-ensemble de N avec $s \in S$ et $t \notin S$. L'ensemble des arcs, noté $\mathcal{C}(S, \bar{S})$, ayant une extrémité dans S et l'autre dans $\bar{S} = X \setminus S$, est appelé **coupe séparant s de t** ou **$s - t$ coupe**. Dans $\mathcal{C}(S, \bar{S})$, on distingue l'ensemble des arcs, noté $\omega^+(S)$, dont l'extrémité initiale est dans S de l'ensemble, noté $\omega^-(S)$, dont l'extrémité terminale est dans S. On a donc

$$\mathcal{C}(S, \bar{S}) = \omega^+(S) \cup \omega^-(S)$$

.

La **capacité d'une coupe**, notée $v(\mathcal{C}(S, \bar{S}))$, est donnée par la somme des capacités des arcs "sortant" de S, c'est-à-dire ceux appartenant à $\omega^+(S)$:

$$v(\mathcal{C}(S, \bar{S})) = \sum_{u \in \omega^+(S)} c_u$$

Théorème 1 : La valeur d'un flot quelconque ϕ dans R est inférieure ou égale à la capacité de toute $s - t$ coupe.

Preuve
Soit S un sous-ensemble de N avec $s \in S$ et $t \notin S$. Considérons les égalités représentant la loi de conservation aux sommets des sommets appartenant à S.

$$\begin{array}{l} \sum_{u \in \omega^+(i)} \phi_u - \sum_{u \in \omega^-(i)} \phi_u = 0 \ \forall i \in S \setminus \{s\} \\ \sum_{u \in \omega^+(s)} \phi_u - \sum_{u \in \omega^-(s)} \phi_u = f_\phi \end{array}$$

On somme ces égalités et on obtient :

$$\sum_{i \in S} \sum_{u \in \omega^+(i)} \phi_u - \sum_{i \in S} \sum_{u \in \omega^-(i)} \phi_u = f_\phi$$

En distinguant les arcs ayant une extrémité dans S et l'autre dans $X \setminus S$, on obtient :

$$\sum_{u=(i,j)\in A: i\in S, j\in S} \phi_u + \sum_{u=(i,j): i\in S, j\in \bar{S}} \phi_u - \sum_{u=(j,i): j\in S, i\in S} \phi_u - \sum_{u=(j,i): j\in \bar{S}, i\in S} \phi_u = f_\phi$$

Ce qui se simplifie en

$$\sum_{u=(i,j): i\in S, j\in \bar{S}} \phi_u - \sum_{u=(j,i): j\in \bar{S}, i\in S} \phi_u = f_\phi$$

En substituant $\phi_u \leq c_u$ et $\phi_u \geq 0$ on obtient :

$$\sum_{u=(i,j): i\in S, j\in \bar{S}} c_u \geq f_\phi$$

$$v(\mathcal{C}(S, \bar{S})) \geq f_\phi \ \ \forall S$$

On a donc :

$$\min_S v(\mathcal{C}(S, \bar{S})) \geq f_\phi$$

En conséquence, si on trouve un flot de valeur f égal à la capacité d'une coupe $\mathcal{C}(S, \bar{S})$, ce flot est de valeur maximal et cette coupe est minimale. Le théorème de Flot max/coupe min ci-dessous stipule qu'il existe un flot dont la valeur est égale à la capacité d'une coupe.

Théorème 2 (Flot max/coupe min) : La valeur d'un flot maximum ϕ dans R est égale à la plus petite des capacités des $s-t$-coupes.

5.5 Exactitude de l'algorithme de Ford et Fulkerson

A chaque itération, soit on trouve une chaîne augmentante, soit on ne parvient pas à marquer t et l'algorithme termine. Dans ce dernier cas, il nous faut montrer que le flot ϕ, de valeur f_ϕ, obtenu à la dernière itération est bien le flot de valeur maximal.

Soit M l'ensemble des sommets marqués à la dernière itération (avec $s \in M$) et $\bar{M}$ l'ensemble des sommets non marqués (avec $t \in \bar{M}$). Comme il s'agit de la dernière itération, il n'est plus possible de marquer un sommet de $\bar{M}$ à partir d'un sommet de M. En conséquence :

$$\forall i \in M,\ \forall j \in \bar{M}\ :\ u = (i,j) \in A,\ c_u = \phi_u \tag{5}$$

$$\forall i \in M,\ \forall j \in \bar{M}\ :\ u = (j,i) \in A,\ \phi_u = 0 \tag{6}$$

Or, si on somme sur M les équations de conservation de flot, on obtient (cf. preuve du théorème flot max - coupe min) :

$$\sum_{u=(i,j): i\in M, j\in \bar{M}} \phi_u - \sum_{u=(j,i): j\in \bar{M}, i\in M} \phi_u = f_\phi$$

Et (5) et (6) impliquent :

$$f_\phi = \sum_{u \in \omega^+(M)} c_u = v(\mathcal{C}(M, \bar{M}))$$

Le flot ϕ de valeur f_ϕ est donc égal à la capacité de la coupe engendrée par les sommets marqués à la dernière itération de Ford et Fulkerson. ϕ est donc un flot de valeur maximale et $C(M, \bar{M})$ une coupe de valeur minimale.

A la dernière itération de l'algorithme de Ford et Fulkerson, on a déterminé un flot maximum, mais également une coupe minimum. Cette coupe est engendrée par l'ensemble des sommets marqués à la dernière itération.

Théorème d'intégrité :
Si toutes les capacités sont entières, le problème du flot maximum a une solution optimale entière.

Preuve :
Le flot de départ est le flot nul. Comme toutes les capacités sont entières, la chaîne augmentante trouvée, si elle existe, dispose d'une capacité δ entière (car toute les capacités résiduelles sont entières). Le nouveau flot obtenu est donc à coordonnées entières. Et ainsi de suite jusqu'à la dernière itération. Au pire cas, on augmente à chaque itération la valeur du flot d'une unité ($\delta = 1$) et, comme le flot maximum ne peut pas dépasser la valeur entière d'une coupe quelconque, l'algorithme termine en un nombre fini d'itérations.

La complexité de l'algorithme est maintenant évidente à calculer. Chaque itération comporte $O(m)$ opérations élémentaires car la méthode de marquage examine chaque arc et chaque sommet au plus une fois. En conséquence, la complexité totale est $O(m)$ fois le nombre d'augmentation. Si chaque capacité est entière et bornée par U, la valeur du flot est bornée par la capacité de la coupe engendrée par $\{s\}$ de valeur maximale $(n-1)U$. Et la complexité totale est en $O(nmU)$.

5.6 Une application du problème du flot maximal : le couplage dans un graphe biparti

Un **couplage** dans un graphe $G = (X, U)$ est un sous-graphe de G où tous les sommets sont de degré 0 ou 1 (les arêtes n'ont pas d'extrémit'es communes). La cardinalité d'un couplage est égale à la cardinalité de l'ensemble U.

Un couplage est dit **parfait** lorsque tous les sommets sont de degré 1.

Un graphe est dit **biparti** si l'ensemble des sommets peut être partitionné en deux sous-ensembles X_1 et X_2 de façon à ce que chaque arête ait une extrémité dans X_1 et l'autre dans X_2.

Exemple : Le problème d'affectation.
Lorsqu'on cherche à affecter des personnes à des tâches (une personne par tâche), on cherche un couplage dans un graphe simple où les sommets représentent les individus et les tâches, et les arêtes représentent les affectations (individu, tâche) possibles. Ce graphe, dit d'affectation, est un graphe biparti. On peut rechercher un couplage de cardinalité maximale lorsqu'on cherche à couvrir le plus de tâches possibles. On peut aussi rechercher un couplage maximal de poids minimal si le graphe d'affectation est valué par des coûts d'affectation.

Pour trouver un couplage maximal dans un graphe biparti il suffit de déterminer un flot de valeur maximale dans le graphe biparti transformé comme suit : on ajoute une source reliée à tous les sommets de X_1 et un puits relié à tous les sommets de X_2, ces nouveaux arcs ayant une capacité 1 (la capacité des arcs du graphe biparti peut valoir 1 ou $+\infty$).

5.7 Recherche d'un flot maximal à coût minimal

Soit f^* la valeur du flot maximal sur R. Le problème du flot maximal à coût minimal dans R peut s'écrire comme suit :

$$\begin{cases} \min \sum_{u \in A} d_u \phi_u \\ \sum_{u \in \omega^+(i)} \phi_u - \sum_{u \in \omega^-(i)} \phi_u = 0 \ \forall i \in N \setminus \{s,t\} \\ \sum_{u \in \omega^+(s)} \phi_u - \sum_{u \in \omega^-(s)} \phi_u = f^* \\ \sum_{u \in \omega^+(t)} \phi_u - \sum_{u \in \omega^-(t)} \phi_u = -f^* \\ 0 \leq \phi_u \leq c_u \ \forall u \in A \end{cases}$$

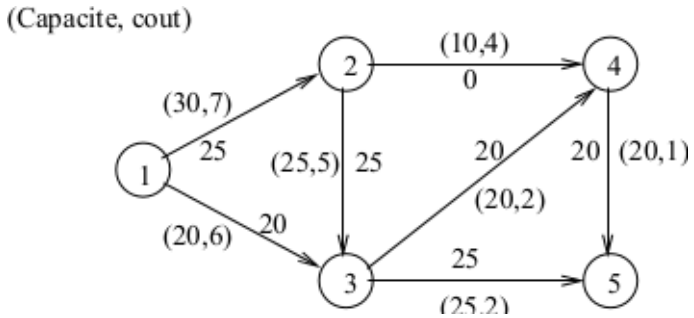
On retrouve aussi la notion de coût dans le graphe d'écart. Soit ϕ un flot admissible sur R. Le graphe d'écart associé à ϕ sur R est le graphe $R^e(\phi) = [N, A^e(\phi)]$ défini comme suit : $\forall u = (i,j) \in A$

— Si $\phi_u < c_u$, $(i,j) \in A^e(\phi)$ avec la capacité (résiduelle) $c'_{(i,j)} = c_u - \phi_u$ et le coût $d'_{ij} = d_u$
— Si $\phi_u > 0$, $(j,i) \in A^e(\phi)$ avec la capacité $c'(j,i) = \phi_u$ et le coût $d'_{ji} = -d_u$.

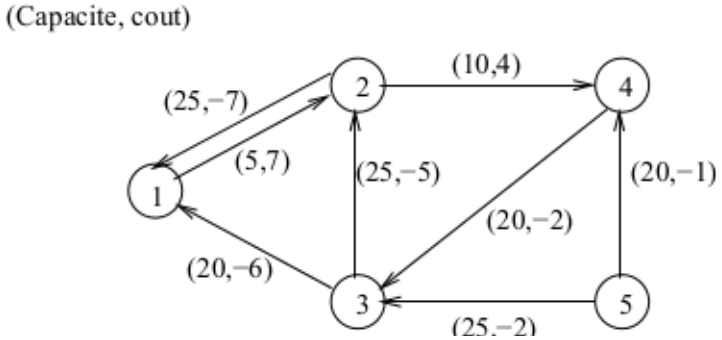
5.7.1 Condition d'optimalité

Théorème.
Une solution ϕ^* est une solution optimale du problème du flot maximal à coût minimal si et seulement si le graphe d'écart associé $R^e(\phi^*)$ ne comporte pas de circuit de longueur négative.

En effet, s'il comporte un circuit de longueur négative cela signifie qu'il est possible de passer du flot ϕ^* à un nouveau flot de coût inférieur.



Le flot de valeur 45 est le flot de valeur maximale (il suffit pour le montrer d'appliquer l'algorithme de marquage de Ford et Fulkerson et d'exhiber la coupe $\{(3,5),(4,5)\}$ de valeur minimale 45. Est-ce le flot de valeur maximale de coût minimal ? Pour cela, il suffit de nous assurer que la condition d'optimalité est vérifiée. Le graphe d'écart associé est représenté ci-dessous :

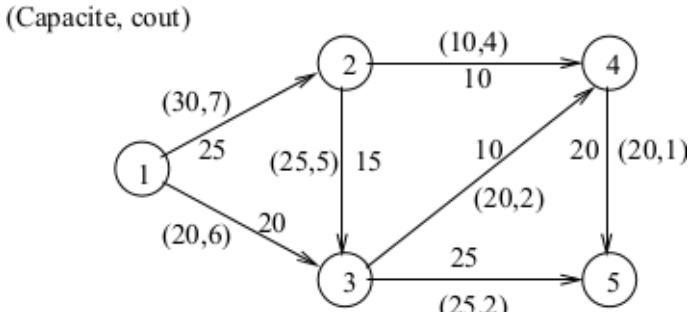


Il existe un circuit de valeur négative : (2,4,3,2) de valeur -3. Sur ce circuit, la capacité résiduelle est de 10. Cela signifie que l'on peut obtenir un nouveau flot de même valeur mais avec une diminution de coût de 30. On obtient une nouvelle solution de coût 500 :

Ce nouveau flot est de coût minimal car il respecte la condition d'optimalité : le graphe réduit associé à cette solution ne présente pas de circuit de valeur négative.

5.7.2 Un algorithme de détermination du flot maximal à coût minimal

De la condition d'optimalité présentée dans la section précédente, il est possible de définir un algorithme très simple de détermination de flot maximal à coût minimal



comme suit :

Etablir un flot de valeur maximale
Tant que le graphe réduit associé contient des circuits de valeur negative faire
Choisir un circuit de valeur négative avec δ la capacité résiduelle minimale du circuit
Augmenter de δ unités le flot dans le circuit et remettre à jour le graphe réduit associé
fin Tant que