Algorithmique des graphes

LICENCE INFORMATIQUE – UNIVERSITE BORDEAUX I

ANNEE 2004 - 2005

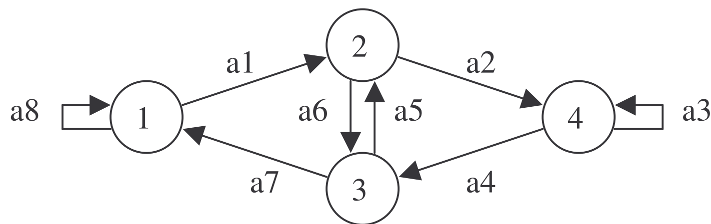
1 GRAPHES DEFINITIONS

1.1 GRAPHES ORIENTES

1.1.1 Généralités

Un graphe orienté est noté G = (X, U), avec X l'ensemble des sommets et U l'ensemble des arcs.

1.1.1.1 Exemple



X = {1, 2, 3, 4}

U = {a_1, a_2, …, a_8}

1.1.1.2 Fonctions et notations

On définit deux fonctions :

T : U → X : à un arc on associe son extrémité terminale.
$a_1 \to 2$

I : U → X : à un arc on associe son extrémité initiale.
$a_1 \to 1$

On note les arcs $a_1 = \overrightarrow{12}$.

Si $\overrightarrow{xy} \in U$ alors :

- y est successeur de x
- x est prédécesseur de y

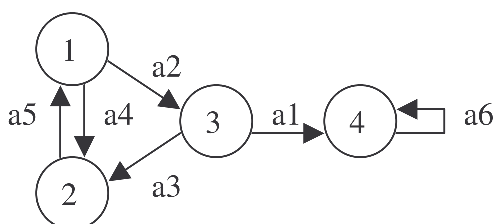
1.1.2 Représentation des graphes orientés

Soit G = (X, U), |X| = n (il y a n sommets {1, 2, …,n}) et |U|=m (il y a m arcs).

1.1.2.1 Matrice d'adjacence

La matrice d'adjacence notée A[G] = $(a_{ij})_{1 \le i \le n \text{ et } 1 \le j \le n}$ est une matrice carrée nxn, a_{ij} représente le nombre d'arcs allant du sommet i au sommet j.

Exemple :



$$A[G] = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ place mémoire : O(n²)

1.1.2.2 Matrice d'incidence

La matrice d'incidence notée B[G] $= (b_{ij})_{1\leq i\leq n,\ 1\leq j\leq m}$ (U = {a_1, … , a_m}) est une matrice nxm telle que b_{ij} est :

- 1 si a_{ij} a pour extrémité initiale i
- -1 si a_{ij} a pour extrémité terminale i
- 0 si l'arc est une boucle ou autre cas.

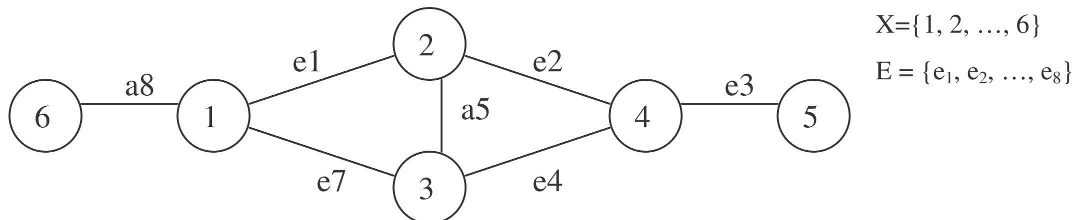
Pour le graphe exemple ci-dessus, la matrice d'incidence est :

$$B[G] = \begin{bmatrix} 0 & 1 & 0 & 1 & -1 & 0 \\ 0 & 0 & -1 & -1 & 1 & 0 \\ 1 & -1 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$ place mémoire O(nxm).

1.2 GRAPHES NON ORIENTES

On note G=(X, E) un graphe non orienté avec X l'ensemble des sommets et E l'ensemble des arêtes. E $\subseteq$ P(X) (partie à deux éléments de X). On peut avoir des arêtes parallèles.

1.2.1 Exemple



X={1, 2, …, 6}

E = {e_1, e_2, …, e_8}

1.2.2 Remarques

Un graphe non orienté est simple s'il n'a ni boucles, ni arêtes multiples.

On définit pour les graphes non orientés, de même A[G] la matrice d'adjacence et B[G] la matrice d'incidence mais avec :

- A[G] = $(a_{ij})_{1\leq i\leq n \text{ et } 1\leq j\leq n}$ le nombre d'arêtes entre i et j
- B[G] $= (b_{ij})_{1\leq i\leq n,\ 1\leq j\leq m}$ le nombre de fois que i est incident à a_j. (donc ici, les boucles apparaissent).

1.3 ADJACENCE ET VOISINAGE

1.3.1 Définition 1

Deux sommets d'un graphe G=(X,E) non orienté (ou G=(X,U) orienté) sont dit adjacents si xy $\in$ E (ou $\overrightarrow{xy}$ $\in$ U ou $\overrightarrow{yx}$ $\in$ U).

1.3.2 Définition 2

Deux arêtes (arcs) sont adjacentes si elles ont une extrémité commune.

1.3.3 Définition 3

x est incident à l'arête (arc) xy si :

- Le voisinage d'un sommet x est $\Gamma(x) = \{y \in V \,/\, xy \in E\}$.
- Le degré d'un sommet x : $d(x) = |\Gamma(x)|$, nombre d'arêtes incidentes (attention, une boucle est répétée deux fois).

1.3.4 Définition 4

Soit G = (X, U) graphe orienté, x ∈ X. On définit :

- Le voisinage extérieur (sortant) de x : $\Gamma^{+}(x) = \left\{ y \in V \,/\, \overrightarrow{xy} \in E \right\}$
- Le voisinage intérieur (entrant) de x : $\Gamma^{-}(x) = \left\{ y \in V \,/\, \overrightarrow{yx} \in E \right\}$

1.3.5 Propriété 1

Soit G=(X, E) un graphe non orienté simple (sans boucle ni arêtes multiples), alors :

- $m \leq \frac{1}{2} n(n-1)$ avec m, le nombre d'arêtes et n le nombre de sommets.
- $\sum_{x \in X} d(x) = 2m$
- Il y a un nombre pair de sommets de degrés impairs.

1.3.6 Propriété 2

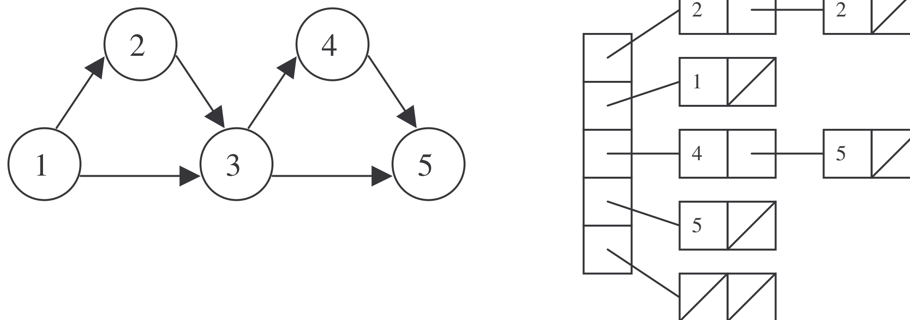
Si G=(X, U) est un graphe orienté alors $\sum_{x \in X} d^{+}(x) = \sum_{x \in X} d^{-}(x) = m$ (avec m le nombre d'arcs).

1.4 REPRESENTATION DES GRAPHES EN MACHINES

1.4.1 Structures peu pratiques

- Matrice d'adjacence
- Matrice d'incidence

1.4.2 Liste <chemin des successeurs>



1.4.2.1 Existence d'un arc ij

- Avec une matrice d'adjacence, O(1) opération.
- Avec une liste des successeurs $O(d^{+}(i))$ opérations

1.4.2.2 Degré d'un sommet

La recherche du degré d'un sommet nécessite O(n) opérations (somme de la ligne i) avec une matrice d'adjacence. Avec la liste des successeurs, il n'y a que $O(d^{+}(i))$ opérations à faire.

Ainsi, pour obtenir le degré de chaque sommet, il y a $O(n^2)$ opérations avec la matrice d'adjacence et O(n*m) opérations avec la liste.

1.4.3 Remarque

Pour les graphes non orientés, on code symétriquement. L'arête ij est codée par les arcs $\overrightarrow{ij}$ et $\overrightarrow{ji}$.

2 EXPLORATIONS DES GRAPHES

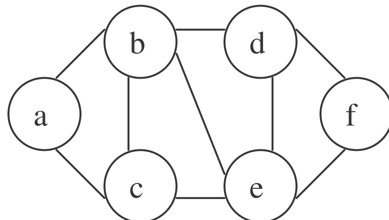
2.1 DEFINITIONS

Soit G = (X, E) un graphe non orienté :

- Un parcours est une suite de sommets $\mu = (x_1, x_2, \ldots, x_k)$ telle que $x_i x_{i+1} \in E$, $i \in \{1, \ldots, k\text{-}1\}$;
- Le parcours est fermé si $x_1 = x_k$;
- Une chaîne est un parcours sans répétition d'arêtes ;
- Un cycle est une chaîne fermée ;
- Une chaîne (cycle) élémentaire est une chaîne (cycle) sans répétition de sommets.

2.1.1 Exemple

Soit le graphe suivant :



Parcours fermé : (a, b, c, d, e, c, b, a)

Une chaîne : (a, c, d, e, c, b, d, f)

Un cycle : (f, d, e, f)

Une chaîne élémentaire (a, c, e, f)

2.1.2 Propriétés et remarques

De toute chaîne on peut extraire une chaîne élémentaire.

Pour les graphes orientés, on appelle chemins (circuit) tous les arcs dans le même sens. La distance de sommets x et y du graphe G est le nombre d'arêtes (arcs) qui les relient.

2.2 PARCOURS EN LARGEUR

2.2.1 Principe

L'algorithme de parcours en largeur consiste à visiter les voisins du sommet de départ, puis les voisins des voisins (et ainsi de suite) non déjà visités. Pour cela, on utilise le principe de « coloration » des sommets :

- Si un sommet n'est pas atteint, il est « blanc »
- Si on découvre le sommet, il est marqué « gris »
- Dès que l'on a examiné tous les voisins d'un sommet, il devient « noir ».

Pour gérer les sommets, on utilise une file FIFO.

2.2.2 Algorithme

```
0    PL (G, s)
1    Pour chaque sommet U ∈ X – {s} faire
2         Couleur[u] ← blanc
3         D[u] ← ∞ (distance de s à u)
4         P[u] ← NIL (père de u)
5    Couleur[s] ← gris
6    D[s] ← 0
7    P[s] ← NIL
8    F ← {s} (s est inséré dans la file)
```

```
9        Tant que F ≠ ∅ (tant que la file n'est pas vide) faire
10           u ← Tete[F]
11           Pour chaque v ∈ Adjacent[u] faire
12               Si Couleur[v] = blanc
13               Alors
14                   Couleur[v] = gris
15                   D[v]←D[u]+1
16                   P[v] ← u
17                   Enfiler(F, v)
18           Défiler(F)
19           Couleur[u] ← noir
20       Fin Tant Que
21       Fin
```

2.2.2.1 Remarque

Au départ, les sommets sont tous blancs. Un sommet découvert devient gris. Un sommet dont on a visité tous les sommets voisins devient noir. On construit alors un arbre des plus courtes chaînes.

2.2.2.2 Complexité

$|X| = n$ et $|E| = m$.

L'initialisation (lignes 1 à 4) de fait en O(n).

Le test sur la couleur (ligne 12) garanti le fait qu'un sommet est enfilé au plus une seule fois (opération en O(n)).

Dans la boucle « tant que » on fait les opérations $\sum_{x\in X} d(x) = 2m$ et la liste d'adjacence d'un sommet n'est parcouru qu'une seule fois (complexité O(n)). Le coût total est donc O(m+n).

2.3 PLUS COURTES CHAINES

Dans tout ce qui suit, on note $\delta(s, u)$ le nombre minimum d'arcs (ou arêtes) dans un chemin de s à u. $\delta(s, u) = \infty$ si le chemin n'existe pas

2.3.1 Propriété 1

Soit $s \in X$. $\forall u, v \in E$, on a $\delta(s, v) \leq \delta(s, u) + 1$

Preuve :

Si u est accessible depuis s, v l'est aussi et le plus court chemin de s à v ne peut pas être plus long que le plus court chemin de s à u plus 1 (pour l'arc uv). Si u n'est pas accessible, la propriété est encore vérifiée : $\delta(s, u) = \infty$

2.3.2 Proposition 1

Soit $G = (X, E)$. PL (parcours en largeur) exécuté à partir de $s \in X$. Alors quand PL se termine, $d(v) \geq \delta(s, v)$.

2.3.2.1 Preuve

La preuve se fait sur le nombre total de sommets insérés dans la file F.

Soit La base de la récurrence (ligne 8 de l'algorithme PL) :

- $d(s) = 0 = \delta(s, s)$
- $\forall v \neq s, d(v) = \infty \geq \delta(s, v)$

Donc la propriété est vrai. On peut passer à l'étape inductive. On considère alors un sommet v BLANC découvert pendant la visite des sommets adjacents à un sommet u. Puisque u est déjà rentré dans F alors $d(u) \geq \delta(s, u)$. Après la ligne 14 de l'algorithme on obtient $d(v) = d(u) + 1 \geq \delta(s, u) + 1 \geq \delta(s, v)$. A la ligne 16, v est inséré dans la file et il ne sera jamais plus réinséré (car il est devenu gris). Donc d(v) ne sera plus modifié. On aura donc à la fin $d(v) \geq \delta(s, v)$.

2.3.2.2 remarques

Pour montrer que $d(v) = \delta(s, v)$ à la fin de l'algorithme PL (voir théorème plus loin) pour chaque v, on commence à montrer que $d(v) \geq d(s, v)$ par la proposition 1. On a ensuite besoin d'étudier le comportement de la file. On va prouver que à tout moment, il existe au plus deux valeurs d distinctes pour les sommets dans la file (par la proposition 2).

2.3.3 Proposition 2

Si F (file) = $<v_1, \ldots, v_n>$ alors $d(v_r) \leq d(v_r)+1$ et $d(v_i) \leq d(v_i+1)$

La preuve peut se faire par induction sur le nombre d'opérations sur la file. Au départ F = {s} et d(s) = 0. L'étape inductive consiste à montrer que le résultat reste vrai après un défilement ou après un enfilement dans F :

- Défilement : on enlève la tête v_1 de F :
  - Cas 1 : si F devient vide, la proposition est vrai
  - Cas 2 : v_2 devient tête de file et on a alors $d(v_1) \leq d(v_1) + 1 \leq d(v_2) + 1$
- Enfilement : A la ligne 16, on ajoute v_{r+1} à F. A ce moment, v_1 est le sommet u dont on est en train de parcourir la liste d'adjacence. Donc, $d(v_{r+1}) = d(v) = d(u) +1 = d(v_1) + 1$. Donc, $d(v_{r+1}) \leq d(v_1) + 1$. On a aussi $d(v_r) \leq d(v_1) + 1 = d(u) + 1 = d(v) = d(v_{r+1})$. Donc la proposition reste vrai lorsqu'on défile et lorsqu'on enfile un sommet.

2.3.4 Théorème

Soit G = (X, E) et PL(G, s) est exécuté :

- $\forall v \in X$, accessible $d(v) = \delta(s, v)$;
- $\forall v \in X$, $v \neq s$, un des plus courts chemins (chaînes) est $s \rightarrow P[v]$ et on complète par (P[v], v) .

Pour la preuve, il faut considérer deux cas :

- Si v n'est pas accessible, v n'est jamais découvert et $d(v) = \delta(s, v) = \infty$
- On Pose $X_k=\{v \in X, \delta(s, v)=k\}$. On montre par induction sur k que, $\forall v \in X, \exists$ t (temps) tel que :
  - V est colorié en gris
  - D(v)=k
  - Si $v \neq s$ alors P[v] a la valeur u pour un $u \in X_{k-1}$
  - V est inséré dans la file.

2.4 RAPPELS O/Ω/θ

2.4.1 Définition 1

Soient deux fonctions f et t de N dans R^+.

On dit que t(n) est en O(f(n)) s'il existe deux constantes $c \in R^{*+}$ telles que pour tout $n \geq n_0$ $t(n) \leq c.f(n)$. O(f(n)) est une classe de fonctions. Donc on peut écrire $T(n) \in O(f(n))$, ou bien T(n)=O(f(n)).

2.4.1.1 Exemple 1

$t(0) = 1$, $t(1) = 4$ et $(n+1)^2 = n^2 + 2n + 1 \leq 4n^2$ donc $t(n) \in O(n^2)$

2.4.1.2 Exemple 2

3^n n'est pas en $O(2^n)$ (preuve à faire en exercice)

2.4.2 Définition 2

Soit une fonction $g : N \rightarrow R^+$

On dit que t(n) est en Ω(g(n)) s'il existe deux constantes $c \in R^{*+}$, n_0 telles que pour tout $n \geq n_0$ $t(n) \geq c.g(n)$. On écrit $t(n) \in \Omega(g(n))$. De même, par abus de langage, on écrit $t(n) = \Omega(g(n))$.

2.4.2.1 Exemple

$t(n) = n^3 + 2n^2$ et c = 1. Alors $t(n) > cn^3$. Donc $t(n) = \Omega(n^3)$.

2.4.2.2 Propriété

$f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$

2.4.3 Définition 3

On note $\theta(f(n)) = O(f(n)) \cap \Omega(f(n))$.

2.4.3.1 Propriété

$f(n) = \theta(g(n)) \Leftrightarrow g(n) = \theta(f(n))$.

2.4.3.2 Exemple

```
Algo1 (T, n, k) (avec T : tableau) :          complexité : O(n)
        Pour i de 1 à n faire T(i)←k

Algo2 (T, U)                                  complexité : O(n²)
        Pour i de 1 à n
                Faire Pour j de 1 à n
                        Faire U(i) = T(i) + T(j)
```

2.5 PARCOURS EN PROFONDEUR

2.5.1 Principe

On visite le graphe en partant d'un sommet. Lorsqu'on arrive à un sommet, on n'explore pas tout ses voisins. On visite un voisin non atteint. On prolonge ainsi la chaîne (ou le chemin).

Si un sommet a été exploré (tous les voisins ont été visités), on l'abandonne pour examiner le sommet qui le précède dans la chaîne. A chaque étape, on choisit une arête (ou arc) reliant le nouveau sommet au sommet marqué comme le plus récent.

2.5.2 Algorithme

2.5.2.1 Notation

Soit un sommet $u \in X$, on note :

- $d(u)$ = la date de début de visite
- $f(u)$ = la date ou on a fini de visiter les voisins de u.

On remarquera alors que $1 \le d(u) \le f(u) \le 2|X|$

Au début, tous les sommet sont blancs (avant d(u)). Ils sont gris entre d(u) et f(u) et noir après f(u).

2.5.2.2 Algorithme

PP[G]

```
1       Pour chaque sommet u ∈ X faire
2               Couleur[u] ← blanc
3               Père[u] ← NIL
4       Temps ← 0
5       Pour chaque sommet u ∈ X faire
6               Si Couleur[u] = blanc
7               Alors Visiter_PP[u]
```

Visiter_PP[u]

```
1       Couleur[u] ← gris
2       d[u] ← temps  ← temps + 1
3       Pour chaque v ∈ Adj[u] faire
4               Si Couleur[v] = blanc
5               Alors    Père [v] ← u
```

6 Visiter_PP[v]
7 Couleur[u] ← Noir
8 F(u) ← temps ← temps + 1

2.5.2.3 Complexité

Le temps total d'exécution est θ(n + m)

2.5.2.4 Exemple

Soit le graphe suivant :

u	d(u)	f(u)	Pere(u)
A	1	10	NIL
C	2	9	A
B	3	8	C
E	4	7	B
D	5	6	E
G	11	20	NIL
F	12	19	G
J	13	18	F
I	14	17	J
H	15	16	I

2.5.3 Théorème

Dans tout parcours en profondeur d'un graphe G = (X, U) (idem pour G = (X, E)), pour deux sommets quelconques u et v, une et une seule des conditions suivantes est vérifiée :

- $[d(u), f(u)] \cap [d(v), f(v)] = \varnothing$;
- $[d(u), f(u)] \subset [d(v), f(v)]$ et u est un descendant de v dans l'arborescence ;
- $[d(v), f(v)] \subset [d(u), f(u)]$ et v est un descendant de u dans l'arborescence.

2.5.4 Tri Topologique (application de l'algorithme de parcours en profondeur)

2.5.4.1 Définition

Soit un graphe G = (X, U). On appelle tri topologique toute injection $f : X \to \mathbb{N}$ telle que pour tout $x \in X$ et tout $y \in X$ si $xy \in U$ alors $f(x) < f(y)$.

2.5.4.2 Exemple

Sur l'exemple suivant on numérote les 5 cinq sommets de manière topologique.

2.5.4.3 Théorème

Un graphe orienté admet un tri topologique si et seulement si il est sans circuit.

Remarque : Un graphe sans circuit est appelé un DAG (Directed, Acyclic Graph).

Preuve du théorème :

Supposons que G possède un circuit, s'il existe un tri topologique f pour G, on se retrouve dans le cas d'un cycle avec f(a) < f(b) < … < f(h) < f(a) (avec a, b…, h sommets). Ce qui est impossible. Donc, un tri topologique implique qu'il n'y a pas de circuit.

Réciproquement, si G est sans circuit alors G possède une source et un puit. Il existe donc une partition des sommets $X = N_0 \cup N_1 \cup \ldots \cup N_k$ telle que $x \in N_i \Leftrightarrow$ i est la longueur d'un plus court chemin se terminant en x.

2.5.4.4 Algorithme de tri topologique

TriTopologique :

> Appeler PP(G) // parcours en profondeur pour calculer f(v) (fin) pour tout $v \in X$
> Chaque fois que le traitement d'un sommet s'achève, l'insérer au début d'une liste chainée.
> Retourner la liste chainée

2.5.4.5 Lemme 1

Un graphe orienté ne contient pas de circuit si et seulement si le parcours en profondeur ne génère aucun arc retour.

2.5.4.6 Théorème

L'algorithme de tri topologique effectue le tri topologique d'un graphe orienté sans circuit.

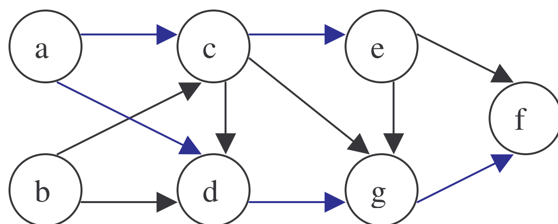
Preuve :

On va montrer que si $uv \in U$ alors f(v) < f(u). Lorsqu'on traverse uv :

- v n'est pas gris, sinon v est un ancêtre de u et alors uv serait un arc retour
- Si v est blanc il est un descendant de u, donc f(v) < f(u)
- Si v est noir on a bien f(v) < f(u) (traitement de v terminé avant le traitement de u).

2.5.4.7 Exemple

Ci-dessous un exemple d'application de l'algorithme de tri topologique. Soit le graphe suivant :

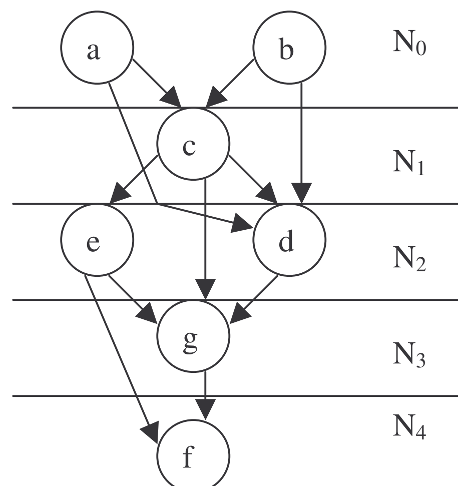


On obtient par PP(G) :

Sommets	d(u)	f(u)
a	1	12
d	2	7
g	3	6
f	4	5
c	8	11
e	9	10
b	13	14

La liste chaînée obtenue est alors la suivante (insertion des sommets en début de la liste) : b a c e d g f.

On obtient alors la partition suivante :



2.5.5 Composantes fortement connexes (application de l'algorithme de parcours en profondeur)

Soit G = (X, U).

2.5.5.1 Définition

On note $x\vec{C}y$ (ou $\vec{C}$ est une relation binaire définie sur les sommets) telle que :

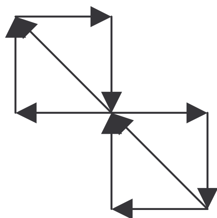
- x = y, ou
- il existe un chemin de x à y et il existe un chemin de y à x.

On remarquera que $\vec{C}$ est une relation d'équivalence.

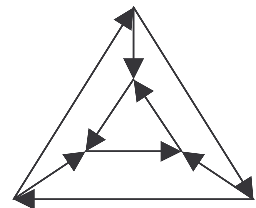
Une classe d'équivalence $\dot{x} = \{t / x\vec{C}y\}$ est une composante fortement connexe. Les composantes fortement connexes sont des partitions du graphe.

Un graphe est fortement connexe s'il ne contient qu'une seule composante fortement connexe.

2.5.5.2 Exemples



Le graphe à gauche est fortement connexe (il contient une seule composante fortement connexe). Le graphe à droite n'est pas fortement connexe, il est composé de deux composantes fortement connexes (les triangles intérieur et extérieur).



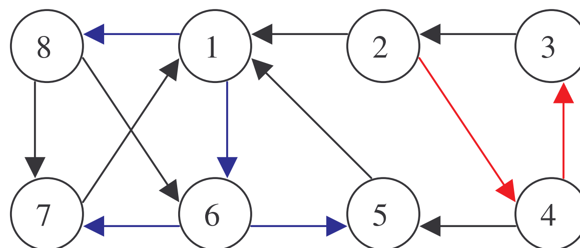
2.5.5.3 Algorithme

L'algorithme de calcul des composantes fortement connexes est le suivant :

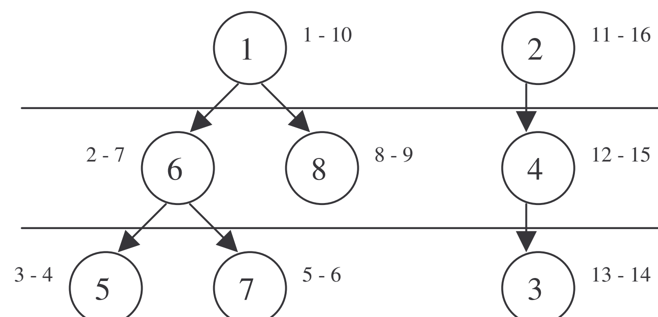
- Appeler le parcours en profondeur sur G : PP(G) ;
- Calculer G^{-1} (graphe obtenu à partir de G en renversant le sens des arcs) ;
- Appeler le parcours en profondeur sur G^{-1} : PP(G^{-1}) en considérant les sommets dans l'ordre décroissant des f(u) obtenus par PP(G) à l'étape 1 ;
- Les arborescences de G^{-1} de la forêt obtenue par PP(G^{-1} à l'étape 3 sont les composantes fortement connexes du graphe.

2.5.5.4 Exemple

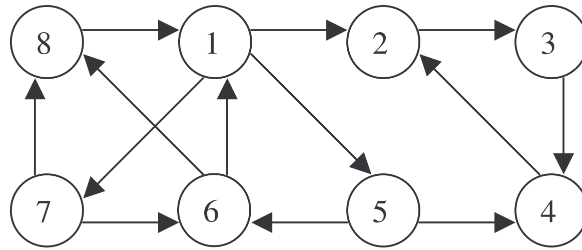
Ci dessous un exemple d'application de l'algorithme de Composante Connexes. Soit le graphe suivant :



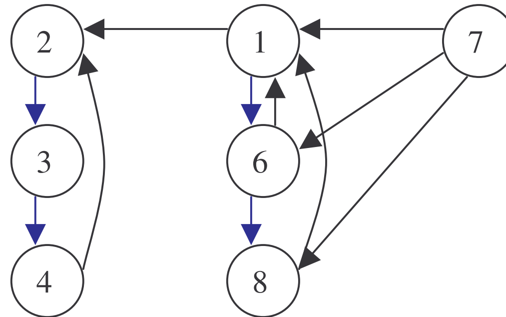
On applique Parcours Profondeur sur le graphe : PP(G). On obtient les niveaux suivants (définis en bleu et rouge sur le graphe ci dessus) :



On prend ensuite G^{-1} :



Et on applique $PP(G^{-1})$ en tenant compte de l'ordre des f(u) obtenus par G. On obtient alors les composantes connexes suivantes (reliées en bleu sur le schéma ci-dessous) :



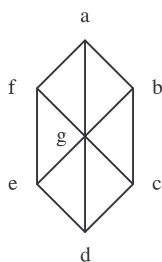
3 ARBRES RECOUVRANT DE POIDS MINIMUM

3.1 DEFINITIONS

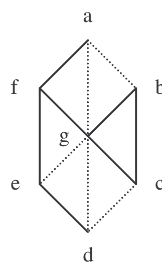
Soit G = (X, E) un graphe non orienté.

- Si G = (X, E) est un graphe, on appelle graphe partiel un graphe G' = (X, F) avec $F \subset E$.

Exemple :



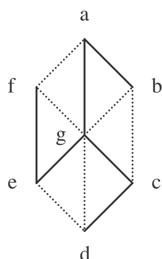
G = (X, E)



G' = (X, F) est un graphe partiel

- Un arbre est un graphe connexe sans cycle ;
- Un arbre couvrant d'un graphe G est un graphe partiel qui est un arbre.

Exemple, avec le même graphe G = (X, E) que pour l'exemple précédent :



T = (X, F) est un graphe couvrant

3.2 THEOREME

Les propriétés suivantes sont équivalentes :

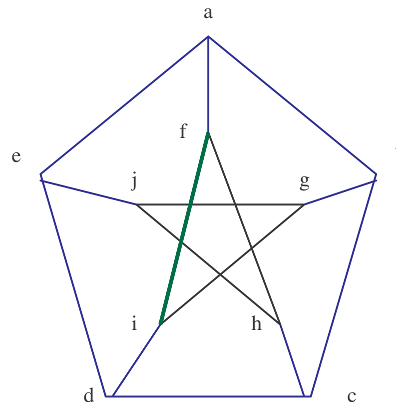
- T est un arbre
- T est connexe et m = n – 1, avec m = nombre d'arêtes et n = nombre de sommets
- T est sans cycle et m = n – 1
- T est connexe et toute arête est un isthme
- T est sans cycle et si on joint deux sommets quelconques non adjacents, on crée un cycle (unique).
- Pour toute paire de sommets a et b il existe une chaîne unique reliant a et b.

3.3 DEFINITIONS (SUITE)

Soit G = (X, E) un graphe et T = (X, F) un arbre couvrant de G. Le cycle fondamental associé à e ∈ E – F est l'unique cycle $C_T(e)$ de T U {e}.

Exemple :

Soit le graphe suivant (en bleu l'arbre couvrant). L'arc u = if permet de construire l'unique cycle $C_T(u)$ = {i, f, a, e, d} tel que $C_T(u)$ ∈ T U {u} :

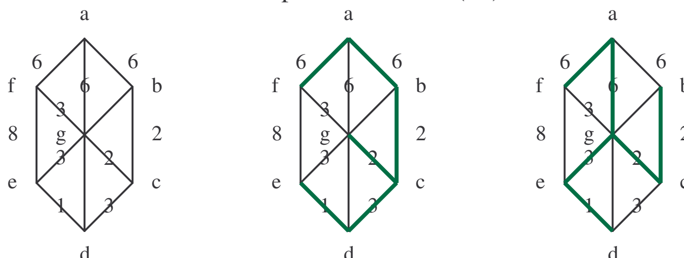


Soit G = (X, E) un graphe non orienté. Soit la fonction poids p : E → R. Soit T = (X, F) un arbre couvrant de G.

Le poids de T est la somme des poids des arêtes appartenant à l'arbre, i.e. : $P(T) = \sum_{e \in F} P(e)$.

Exemple :

Les deux arbres verts sont de poids minimum (20)



3.4 ALGORITHME DE KRUSKAL (1956)

3.4.1 Idées

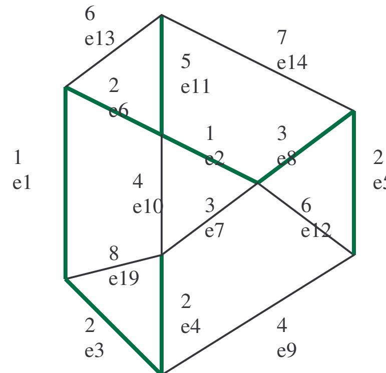
On note m le nombre d'arêtes et n le nombre de sommets. On considère un ensemble d'arête S qui va grossir de 1 à n – 1. On pose i = | S |. Au début, on à donc S = ∅ et i = 0.

- On effectue un tri des arêtes par ordre de poids croissant.

- Tant que $i \leq n - 2$ (S est alors constitué des arêtes $e_1, e_2, \ldots, e_i$), on cherche une arête e_j qui n'appartient pas à S telle que :
- Le sous graphe engendré par $e_1, e_2, \ldots, e_i, e_j$ est sans cycle
  - $P(e_j)$ est de poids minimum parmi les arêtes de poids minimum.
  - Puis on pose $e_{i+1} = e_j$ et $S \leftarrow S \cup \{e_{i+1}\}$, $i \leftarrow i + 1$
  - Cet algorithme est glouton.

3.4.2 Exemple

Soit le graphe suivant (en vert l'arbre recouvrant de poids minimum). Les arêtes ont été numérotés dans l'ordre croissant de leur poids :



3.4.3 Preuve de l'algorithme

On note $T^* = \{e_1, e_2, \ldots, e_{n-1}\}$ un arbre. On utilise le théorème « tout arbre construit par l'algorithme de Kruskal est un arbre couvrant de poids minimum ». Soit T* cet arbre. On suppose que T* n'est pas de poids minimum. Si T est un arbre couvrant de poids minimum, $T \neq T^*$, on note $\mathrm{Inf}(T) = \mathrm{Inf}\{i \mid i \notin T\}$. On choisit T, arbre couvrant de poids minimum ayant f(T) maximum. $f(T) = k$, $k \leq n - 2$.

On a :

- $T^* = e_1\ e_2 \ldots e_{k-1}\ e_k \ldots e_{n-1}$
- $T = e_1\ e_2 \ldots e_{k-1}\ f_k \ldots f_{n-1}$

Les e_i sont classés par ordre croissant de poids. $T \cup \{e_k\}$ contient alors un cycle fondamental $C_T(e_k)$. Et dans ce cas, il existe $e'_k \in C_T(e_k)$ tel que $e'_k \notin T^*$, alors $e'_k = f_i$ pour $i \geq k$. Soit $T' = T \cup \{e_k\} - \{e'_k\}$ (avec, rappelons le, e_k l'arête ajoutée pour former le cycle et e'_k l'arête supprimé dans le même cycle pour supprimer celui ci). T' est un arbre (on enlève une arête pour la remplacer par une autre dans le cycle).

Donc $P(T') = P(T) + P(e_k) - P(e'_k)$.

Le sous graphe engendré par $e_1, e_2, \ldots, e_{k-1}, e'_k$ est un graphe sans cycle, par l'algorithme de Kruskal $p(e_k) \leq p(e'_k)$. Donc $P(T') \leq P(T)$. Or T arbre recouvrant de poids minimum donc, $P(T') = P(T)$. Donc, T' est un arbre couvrant de poids minimum. $F(T) \geq k + 1$, $f(T) = k$. Mais f(T) était maximum. Il y a donc contradiction et T* est un arbre couvrant de poids minimum.

3.4.4 Algorithme de Kruskal

On suppose que les arêtes sont numérotés dans l'ordre croissant des poids.

$S \leftarrow \varnothing$
Pour j de 1 à m faire
    Si (X, $S \cup e_j$) ne contient pas de cycles
    Alors $S \leftarrow S \cup \{e_j\}$

3.4.5 Test d'acyclicité

On numérote les sommets de 1 à n. On prend une arête que si les extrémités ont des numéros différents. On prend une arête e_j, tous les sommets qui ont même numéro que l'une des extrémités sont re-numérotés avec le plus petit numéro des extrémités.

Etudions la complexité de cet algorithme :

- Le tri des arêtes m(logm)
- Comparer les extrémités θ(m)
- La re-numérotation est $O(n^2)$

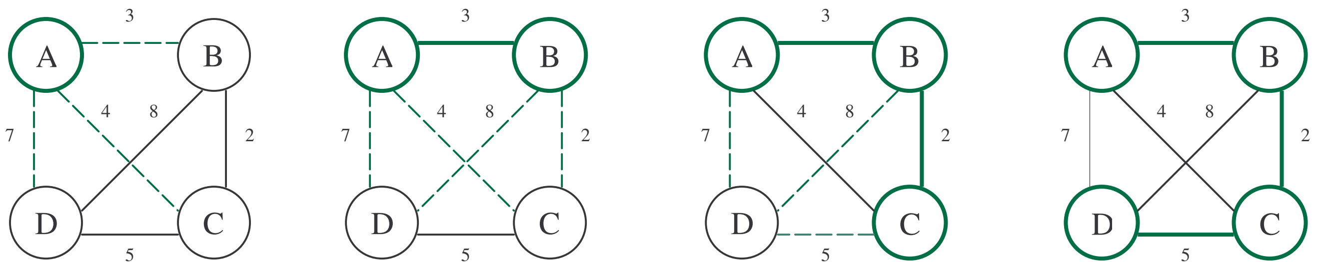
Donc la complexité est en $O(Sup(n^2, mlogm))$

3.5 ALGORITHME DE PRIM

3.5.1 Principe de l'algorithme

Soit G = (X, E) un graphe non orienté et la fonction poids p : E → R. On construit de proche en proche un ensemble d'arêtes S (qui donnera un arbre). On choisit l'arête qui sort de S et de poids minimum.

3.5.2 Exemple



On part du sommet A et on regarde les arêtes qui sortent de l'ensemble S (l'ensemble S est coloré en vert sur le schéma et les arêtes sortantes de S sont en pointillés verts). A chaque tour, on sélectionne l'arête sortant de S de poids minimum et on ajoute le sommet atteint à S. On s'arête lorsque on a n – 1 arêtes dans S.

3.5.3 Algorithme

3.5.3.1 Algorithme simplifié

S ← ∅
A ← r (avec A l'ensemble des sommets marqués)
Tant que |S| < n – 1 faire
    Choisir u = xy tel que x ∈ A et y ∉ A de poids minimum
    S = S U u
    A = A U y

3.5.3.2 Preuve de l'algorithme

On construit un graphe connexe avec n – 1 arêtes. Donc il s'agit d'un arbre. On montre par induction qu'à chaque étape, si S est l'ensemble des arêtes sélectionnées, alors sil existe un arbre couvrant de poids minimum contenant S.

Initialement, la propriété est vide car S est vide.

A une étape i, on suppose que l'ensemble des arêtes S est contenu dans un arbre couvrant T* de poids minimum. On note A l'ensemble des sommets marqués.

A l'étape i + 1, on choisit l'arête xy tel que x ∈ A et y ∉ A, poids(x, y) est minimum :

- Soit xy ∈ T* et c'est fini
- Si xy ∉ T*, alors il existe x', y' ∈ R* tels que x' ∈ A et y' ∉ A. Donc T' = T* U {xy} – {x'y'} est un arbre tel que P(T') = P(T*) + p(xy) – p(x'y'). Or xy est l'arête de poids minimum qui sort de A alors

p(xy) ≤ p(x'y'). Donc P(T') ≤ P(T*). Comme T* est un arbre couvrant de poids minimum alors P(T') = P(T*). Donc T' est un arbre couvrant de poids minimum et S U {xy} ⊆ T'.

3.5.3.3 Algorithme de Prim

```
Prim (G, r)
F ← X
Pour chaque x ∈ F faire
        Clé (x) ← +∞
Clé (r) ← 0
Père (r) ← NIL
Tant que F ≠ ∅ faire
        x ← Extraire_Min(F)
        Pour chaque y ∈ Adj(x) faire
                Si y ∈ F et p(xy) < Clé(y)
                Alors   Père(y) ← x
                        Clé(y) ← p(xy)
```

On note F une file de priorité (un tas) des sommets. Tous les sommets qui ne sont pas dans l'arbre sont dans la file de propriété dépendant d'une clé. Clé(x) est de poids minimal d'une arête reliant x à un sommet de l'arbre. Si une telle n'existe pas, clé(x) = +∞. Père(x) est le père de x dans l'arbre couvrant.

3.5.3.4 Complexité

La construction du tas se fait en O(log(n)). Pour extraire Min(x), que l'on fait n fois, il faut alors nlog(n). Pour mettre la clé à jour (lignes 8 à 11) il faut mlog(n). On obtient donc au total O(nlog(n) + mlog(n)). Par majoration on a O(mlogn).

4 PLUS COURTS CHEMINS

4.1 RAPPELS

4.1.1 Définitions

Soit G = (X, U) un graphe orienté. On note w : U → R la fonction poids. On rappelle que si u = $\overrightarrow{xy}$ alors I(u) = x et T(u) = y.

Un chemin est une séquence alternée de sommets et d'arcs. : $x_1u_1x_2u_2\ldots u_kx_{k+1}$ avec $u_k = x_k x_{k+1}$.

4.1.2 Propriétés

On retiendra les propriétés suivantes :

- Si p est un chemin de poids minimum alors $\omega(p) = \sum_{u\in p} \omega(x)$
- Un chemin élémentaire ne passe pas deux fois par le même sommet ;
- La racine de G est un sommet s tel qu'il existe un chemin de s à tous les autres sommets ;
- Un circuit est un chemin dont les extrémité coïncident ;
- Un circuit C est dit absorbant si $p(C) = \sum_{u\in C} p(u) < 0$.
- Une arborescence de racine s dans un graphe orienté g = (X, U) est un sous graphe dont le graphe non orienté sous-jacent est un arbre et il y a un chemin de s à tous les sommets de l'arborescence (dans l'arborescence) ;
- Un plus court chemin entre deux sommets x et y du graphe est un chemin P qui relie x et y de plus petit poids.

4.1.3 Théorème

Soit G = (X, U) un graphe orienté. Soit s ∈ X. Pour tout x ∈ X, il existe un chemin de s à x si et seulement si :

- s est racine ;
- G = (X, U, ω) ne contient pas de circuit absorbant.

4.2 PLUS COURT CHEMIN D'UN SOMMET A TOUS LES AUTRES

4.2.1 Cas 1 : longueurs positives ou nulles : Algorithme de Dijkstra

Remarque : dans ces cas la, il n'y a pas de circuit absorbant.

4.2.1.1 Principe

On part de s pour construire une arborescence. On augmente l'ensemble des arcs en prenant des arcs dont l'extrémité initiales est dans D (les sommets découverts) et l'extrémité terminale n'es pas dans D.

4.2.1.2 Notation

On note D l'ensemble des sommets découverts, d(x) la distance entre s et x et ε le mot vide.

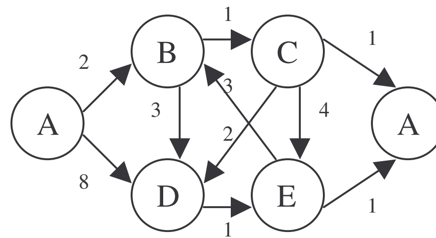
4.2.1.3 Algorithme

Dijkstra (G, s)

```
1 :     D ←{s} ; d(s) ← 0 ; A(s) ← ε ; k ← 1 ; x1 ← s
2 :     Pour chaque (x ∈ X et x ≠s) faire
3 :             d(x) ← +∞
4 :     Tant que (k < n et d(xk) < +∞) faire
5 :             Pour chaque u ∈ U tel que I(u) = xk et T(u) n'appartient pas à D faire
6 :                     x ← T(u)
7 :                     Si d(x) > d(xk) + ω(u)
8 :                     Alors   d(x) ← d(xk) + ω(u)
9 :                             a5x) ← u
10 :            Choisir x n'appartenant pas à D telq eu d(x) = Min(d(y)) et y n'est pas dans D
11 :            k ← k + 1
12 :            xk ← x
13 :    D ← D U {xk}
```

4.2.1.4 Exemple

Soit le graphe suivant :



On part de A au départ. Donc s = A. Puis, pour tout x ∈ X, x ≠ s, d(x) = ∞ et d(s) = 0

Les tableaux suivants marquent les différentes étapes pour trouver l'arbre de plus court chemin avec application de l'algorithme de Dijkstra.

Etape 1 : s_1 = A = s et D = A

	A	D	C	D	E	F
D(u)	0	2	∞	∞	8	∞
A(x)	0	AB			AE	

Etape 2 : x_2 = B (on prend le plus petit dans les distances parmi les non découverts) et D = {A, B}

	A	B	C	D	E	F
D(u)	0	2	3	∞	5	∞
A(x)	0	AB	BC		BE	

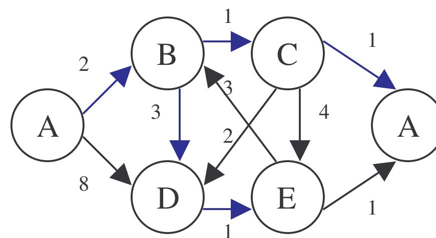
Etape 3 : on prend x_3 = C et D = {A, B, C}

	A	B	C	D	E	F
D(u)	0	2	3	4	5	7
A(x)	0	AB	BC	CD	BE	CF

Etape 4 ; on prend x_4 = D. Il n'y a pas de mise a jour possible car il n'y a pas d'arcs sortants. On prend donc x_5 = E et D = {A, B, C, D, E}. On obtient :

	A	B	C	D	E	F
D(u)	0	2	3	4	5	6
A(x)	0	AB	BC	CD	BE	EF

L'arbre recouvrant obtenu est donc le suivant (arcs bleus) :



4.2.1.5 Preuve de l'algorithme

Quand x est intégré dans D, la valeur de d(x) n'est pas changée (car déjà mise à jour avant). Si $d(x_{j-1}) \leq d(x_j, j \in \{2, \ldots, k\}$ (1) et $d(x_k) \leq d(x)$ $\forall$ x n'appartenant pas à D (2) sont vraie à l'itération k – 1 alors elles sont vraies à l'itération k. Comme le poids de l'arc u, $w(u) \geq 0$, la mise à jour de d(x) ne pet pas inverser les inégalités.

Montrons que d(x) (avec x choisi pour entrer dans D) représente la longueur d'un plus court chemin de s à x dans la classe des chemins $C_k(x)$ dont tous les sommets intermédiaires sont dans D.

Cette propriété est vraie pour k = 1.

Supposons qu'elle est vraie jusqu'à k – 1, démontrons la pour la $k^{ième}$ itération. Soit x intégré à la $k^{ième}$ itération. On choisit ^x un prédécesseur de x sur un plus court chemin C de $C_k(x)$. Si ^x = x_j, pour j < k, le chemin de s à x est dans D.

Hypothèse de récurrence : d(x) est la plus courte distance de s à x par un chemin de $C_k(x)$. Donc ^x = x_k est vrai pour la même raison.

A la fin de l'algorithme s est racine sur X = D, donc la distance d'un plus court chemin de s à x.

4.2.1.6 Complexité

Les ligne 2 – 3 : O(n) affectations, l 5 à 9 : O(n) et l 10 (pour trouver le minimum O(n²). Donc algorithme en O(n²).

4.2.2 Cas des graphes sans circuits

4.2.2.1 Principe

On cherche les plus courtes distances de proche en proche. On grossit à chaque étape une arborescence (i.e. on ajoute un sommet et un arc).

4.2.2.2 Algorithme de Bellman

1 : D ← {s} ; d(s) ← 0 ; A(s) ← ε ; ∀ x ∈ X, x ≠ s, d(x) ← ∞
2 : Tant qu'il existe x n'appartenant pas à D dont tous les prédécesseurs appartiennent à D faire
3 : d(x) ← Min [d(I(u)) + w(u)]
4 : Soit ~u tel que d(x) = d(I(~u)) + w(u))
5 : A(x) ← ~u ; D ← D U {x}

4.2.2.3 Preuve de l'algorithme

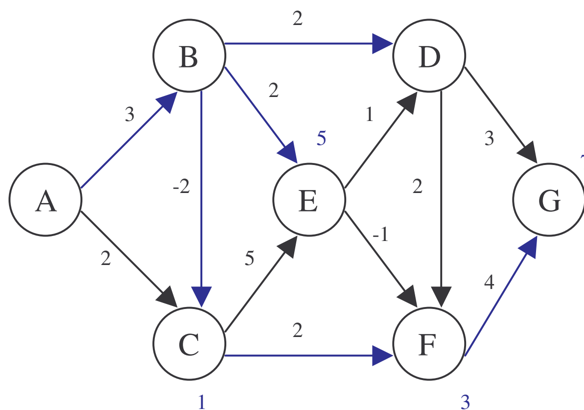
S'il n'existe pas de sommets de $x \in X$ tel que tous les prédécesseurs sont dans D alors D est l'ensemble des sommets dont s est racine. Soit $D \neq X$, il existe $y \in X - D$ qui n'a pas de prédécesseurs dans D. On peut trouver un suite infinie z_i en dehors de D tel que l'on ait les arcs $z_i z_{i+1}$. Mais le graphe est fini, la seule possibilité implique qu'il y a un circuit (impossible par hypothèse).

On calcule d(x) que si on a calculé les plus courtes distances de s à tous ses prédécesseurs.

La complexité de cet algorithme est en $O(n^2)$.

4.2.2.4 Exemple

Le chemin obtenu en bleu sur le graphe suivant est le résultat de l'application de l'algorithme de Bellman (les valeurs indiquées en bleues correspondent aux distances des sommets par rapport à la racine A) :



4.2.3 Plus court chemin avec tri topologique

4.2.3.1 Principe

L'algorithme de plus cour chemin d'un sommet à tous les autres en utilisant le tri topologique ne concerne que les graphe sans circuit. On a s racine. On note denum(j) le sommet de numéro j obtenu par le résultat du tri topologique du graphe.

4.2.3.2 Algorithme de Bellman pour le tri topologique

```
1 :     A(s) ← ε
2 :     d(s) ← 0
3 :     Pour tout x ∈ X et x ≠ s faire
4 :             d(x) ← ∞
5 :     Pour j ← 2 à m faire
6 :             y ← denum (j)
7 :             d(y) ← Min [d(x) + w(u)], u/y = T(u)
8 :             Soit ~u tel que d(y) = d(x) + w(~u)
9 :             A(y) ← ~u
```

4.2.3.3 Complexité :

La complexité de cet algorithme est en $O(n^2)$.

4.3 RECHERCHE D'UN PLUS COURT CHEMIN POUR TOUT COUPLE DE SOMMETS

4.3.1 Algorithme de Floyd (1962)

On calcule les distances entre tout couple de sommets. Soit G = (X, U) un graphe valué avec des coûts quelconques (orienté). On suppose qu'il n'y a pas de circuit absorbant.

4.3.1.1 Principe

Soit X = {1, 2, … n}.

On regarde un couple (i, j) quelconque. On considère les chemins qui vont de i à j dont tous les sommets intermédiaires seront à l'étape k dans {1, 2, … k}. Soit P un tel chemin de poids minimum (chemin élémentaire car pas de circuit absorbant). On note :

- d_k (i, j) le coût d'un plus court chemin de i à j passant par les sommets appartenant à {1, 2, …k}.
- pred (i, j) le prédécesseur de j d'un tel chemin.

4.3.1.2 Notation

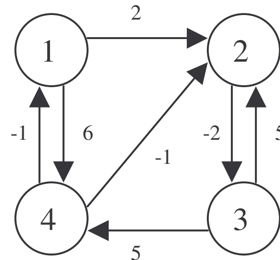
Soit A = $M_{n,\,n}$ une matrice telle que $a_{ij} = \begin{cases} w(ij) & si \quad \vec{ij} \in U \\ +\infty & si \quad \vec{ij} \notin U \end{cases}$.

A chaque étape k, cette matrice contient la distance $d_k(i, j)$.

Soit P = $M_{n,\,n}$ une matrice prédécesseur telle que p_{ij} = prédécesseur à l'itération k de l'arc ij. Au début, p_{ij} = i.

4.3.1.3 Exemple

Soit le graphe suivant



A l'étape k, on considère le chemin de i à j passant par des sommets de numéro ≤ k. On peut avoir un chemin plus court à l'étape k que celui obtenu à l'étape k – 1.

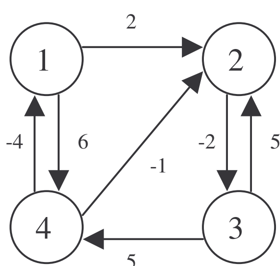
On obtient la définition récursive suivante :

- A k = 0, d_k (ij) = w(ij)
- A l'étape k, $d_k(ij)$ = Min [$d_{k-1}(ij)$, $d_{k-1}(ik) + d_{k-1}(kj)$]. Si $d_k(ij) = d_k(ik) + d_k(kj)$ alors $pred_k(ij) = pred_{k-1}(kj)$.

4.3.1.4 Algorithme

```
1 :     Pour i ← 1 à n faire
2 :             Pour j ← 1 à n faire
3 :             A[i, j] ← w(i, j)
4 :             P[i, j] ← i
5 :     Pour k ← 1 à n faire
6 :             Pour i ← 1 à n faire
7 :                     Pour j ← 1 à n faire
8 :                             Si A[i, k] + A[k, j] < A[i, j]
9 :                             Alors   A[i, j] ← A[i, k] + A[k, j]
10 :                                    P[i, j] ← P[k , j]
```

4.3.1.5 Exemple d'exécution



Reprenons le graphe de l'exemple précédent et appliquons l'algorithme de Floyd.

A l'initialisation, nous avons les matrices suivantes :

$$A = \begin{vmatrix} 0 & 2 & \infty & 6 \\ \infty & 0 & -2 & \infty \\ \infty & 5 & 0 & 5 \\ -4 & -1 & -1 & 0 \end{vmatrix} \text{ et } P = \begin{vmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{vmatrix}.$$

Première itération. k = 1 : on regarde si les distances sont diminuées entre les sommets si l'on passe par le sommet intermédiaire k = 1. C'est à dire, on regarde si (on remarquera, par leur absurdité, que de nombreux cas ne sont pas à traiter) :

- A[1, 1] > A[1, 1] + A[1, 1], inutile à traiter, ainsi que tous les cas de la diagonale, i.e. les couples (1, 1), (2, 2), …
- A[1, 2] > A[1, 1] + A[1, 2], inutile à traiter car A[i, i] = 0 ∀ i donc l'inégalité A[i, j] > 0 + A[i, j] est fausse.
- A[1, 3] > A[1, 1] + A[1, 3], idem,
- A[1, 4] > A[1, 1] + A[1, 4], idem,
- A[2, 1] > A[2, 1] + A[1, 1], idem
- A[2, 2] > A[2, 1] + A[1, 2], inutile à traiter. Nous rappelons que les graphes utilisés pour cet algorithme ne sont pas absorbant. Donc il ne peut pas y avoir de chemin allant de 2 à 2 et passant par d'autres sommets tels que cette inégalité soit vraie.
- A[2, 3] > A[2, 1] + A[1, 3], cas à traiter,
- A[2, 4] > A[2, 1] + A[1, 4], cas à traiter,
- A[3, 1] > A[3, 1] + A[1, 1], inutile à traiter,
- A[3, 2] > A[3, 1] + A[1, 2], cas à traiter
- A[3, 3] > A[3, 1] + A[1, 3], inutile à traiter,
- A[3, 4] > A[3, 1] + A[1, 4], cas à traiter,
- A[4, 1] > A[4, 1] + A[1, 1], inutile à traiter,
- A[4, 2] > A[4, 1] + A[1, 2], cas à traiter,
- A[4, 3] > A[4, 1] + A[1, 3], cas à traiter,
- A[4, 4] > A[4, 1] + A[1, 4] , inutile à traiter.

Après un premier nettoyage, on obtient alors que les seuls cas à traiter sont :

- A[2, 3] > A[2, 1] + A[1, 3],
- A[2, 4] > A[2, 1] + A[1, 4],
- A[3, 2] > A[3, 1] + A[1, 2],
- A[3, 4] > A[3, 1] + A[1, 4],
- A[4, 2] > A[4, 1] + A[1, 2],
- A[4, 3] > A[4, 1] + A[1, 3],

Remarque 1 :

Il est alors bon de remarquer de manière générale (vraie pour la première itération du programme ainsi que les suivantes) que les seuls cas à étudier à l'itération k, sont les couples de sommets (i, j) tels que :

A[i, j] > A[i, k] + A[k, j] avec : i ≠ j ≠ k.

Remarque 2 :

Il suffit que l'un des deux membres de droite (dans l'inégalité) soit infini pour savoir que le résultat ne sera pas modifié.

Reprenons :

Le premier cas à traiter est : A[2, 3] > A[2, 1] + A[1, 3] → -2 > ∞ + ∞. Ce qui est faux, donc A[2, 3] n'est pas modifié. Les trois cas suivants A[2, 4] > A[2, 1] + A[1, 4], A[3, 2] > A[3, 1] + A[1, 2] et A[3, 4] > A[3, 1] + A[1, 4], ainsi que le dernier cas A[4, 3] > A[4, 1] + A[1, 3] donnent des résultats similaires. Les valeurs ne sont donc pas modifiées.

En revanche, le cas A[4, 2] > A[4, 1] + A[1, 2] donne –1 > -4 + 2 = -2 qui est vrai. Donc on va modifier les matrices telle que a_{42} = -2 et p_{42} = p_{12} = 1. On obtient les matrices suivantes :

$$A = \begin{vmatrix} 0 & 2 & \infty & 6 \\ \infty & 0 & -2 & \infty \\ \infty & 5 & 0 & 5 \\ -4 & -2 & -1 & 0 \end{vmatrix} \text{ et } P = \begin{vmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 1 & 4 & 4 \end{vmatrix}.$$

On passe à la deuxième itération k = 2. Les cas à vérifier sont donc :

- A[1, 3] > A[1, 2] + A[2, 3] → ∞ > 2 + (-2), vraie donc A[1, 3] ← 0 et P[1, 3] ← P[2, 3] = 2
- A[1, 4] > A[1, 2] + A[2, 4] → fausse car A[2, 4] = ∞
- A[3, 1] > A[3, 2] + A[2, 1] → idem
- A[3, 4] > A[3, 2] + A[2, 4] → idem
- A[4, 1] > A[4, 2] + A[2, 1] → idem
- A[4, 3] > A[4, 2] + A[2, 3] → -1 > -2 + (-2) = -4 donc : A[4, 3] ← -4 et P[4, 3] ← P[2, 3] = 2

On obtient alors les matrices suivantes :

$$A = \begin{vmatrix} 0 & 2 & 0 & 6 \\ \infty & 0 & -2 & \infty \\ \infty & 5 & 0 & 5 \\ -4 & -2 & -4 & 0 \end{vmatrix} \text{ et } P = \begin{vmatrix} 1 & 1 & 2 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 1 & 2 & 4 \end{vmatrix}.$$

On passe à la troisième itération k = 3 (on traite sur le troisième sommet comme intermédiaire). Les cas à vérifier sont donc :

- A[1, 2] > A[1, 3] + A[3, 2] → 2 > 0 + 5, donc A[1, 2] n'est pas modifiée
- A[1, 4] > A[1, 3] + A[3, 4] → 6 > 0 + 5, donc A[1, 4] ← 5 et P[1, 4] ← P[3, 4] = 3
- A[2, 1] > A[2, 3] + A[3, 1] → ∞ > -2 + ∞, donc A[2, 1] n'est pas modifiée
- A[2, 4] > A[2, 3] + A[3, 4] → ∞ > -2 + 5 = 3, donc A[2, 4] ← 3 et P[2, 4] ← P[3, 4] = 3
- A[4, 1] > A[4, 3] + A[3, 1] → -4 > -4 + ∞, donc A[4, 1] n'est pas modifiée
- A[4, 2] > A[4, 3] + A[3, 2] → -2 > -4 + 5 = 1, donc pas de modification

On obtient alors les matrices suivantes :

$$A = \begin{vmatrix} 0 & 2 & 0 & 5 \\ \infty & 0 & -2 & 3 \\ \infty & 5 & 0 & 5 \\ -4 & -2 & -4 & 0 \end{vmatrix} \text{ et } P = \begin{vmatrix} 1 & 1 & 2 & 3 \\ 2 & 2 & 2 & 3 \\ 3 & 3 & 3 & 3 \\ 4 & 1 & 2 & 4 \end{vmatrix}.$$

On effectue enfin la dernière itération sur k (traitement sommet 4). Les cas à traiter sont les suivants :

- A[1, 2] > A[1, 4] + A[4, 2] → 2 > 5 + (-2) = 3 → pas de modification
- A[1, 3] > A[1, 4] + A[4, 3] → 0 > 5 + (-4) = 1 → pas de modification
- A[2, 1] > A[2, 4] + A[4, 1] → ∞ > 3 + (-4) =-1 donc A[2, 1] ← -1 et P[2, 1] ← P[4, 1] = 4
- A[2, 3] > A[2, 4] + A[4, 3] → -2 > 3 + (-4) → pas de modification
- A[3, 1] > A[3, 4] + A[4, 1] → ∞ > 5 + (-4) = 1 donc A[3, 1] ← 1 et P[3, 1] ← P[4, 1] = 4
- A[3, 2] > A[3, 4] + A[4, 2] → 0 > 5 + (-2) → pas de modification.

A la fin de l'exécution de l'algorithme, les matrices obtenues sont donc les suivantes :

$$A = \begin{vmatrix} 0 & 2 & 0 & 5 \\ -1 & 0 & -2 & 3 \\ 1 & 5 & 0 & 5 \\ -4 & -2 & -4 & 0 \end{vmatrix} \text{ et } P = \begin{vmatrix} 1 & 1 & 2 & 3 \\ 4 & 2 & 2 & 3 \\ 4 & 3 & 3 & 3 \\ 4 & 1 & 2 & 4 \end{vmatrix}.$$

Méthode d'utilisation des matrices obtenues :

Pour retrouver le chemin de i à j on regarde la matrice des précédents. Par exemple, pour retrouver le chemin de 4 à 3 :

- Pred (4, 3) = 2, donc on regarde
- Pred (4, 2) = 1, donc regarde
- Pred (4, 1) = 4.

En remontant dans les résultats on obtient que le plus court chemin de 4 à 3 est $4 \rightarrow 1 \rightarrow 2 \rightarrow 3$

4.3.1.6 Complexité

Cet algorithme est de complexité $O(n^3)$

4.3.2 Algorithme de Warshall

4.3.2.1 Principe

On cherche s'il existe des chemins de i à j pour tout couple (i, j).

4.3.2.2 Remarque

Soit un graphe orienté G = (X, U) avec X = {1, 2, …, n}. Soit $A = M_{n,n}$ une matrice booléenne telle que $a_{ij} = 1$ si l'arc ij appartient à U.

4.3.2.2.1 Proposition

Soit p un entier strictement positif et A la matrice précédente : $A^p = (a_{ij}^p)_{1\leq i\leq n,\ 1\leq j\leq n}$, $A^p = A^{p-1}.A$, et dans ce cas : $a_{ij}^p = 1 \Leftrightarrow$ il existe un chemin de i à j de longueur p.

4.3.2.2.2 Preuve

Si p = 1 alors c'est vrai

Supposons que la propriété est vraie pour p. On le montre pour p + 1 :

$$A^{p+1} = A^p + 1$$

$$(a_{ij}^{p+1}) = \sum_{k=1}^{n} a_{ik}^p . a_{kj}$$

$$donc \quad (a_{ij})^{p+1} = 1 \Leftrightarrow \exists k / a_{ik}^p . a_{kj} = 1 \Leftrightarrow a_{ik}^p = 1 \wedge a_{kj} = 1$$

Or :

- $a_{ik}^p = 1 \Leftrightarrow$ il existe un chemin de longueur p qui va de i à k
- $a_{kj} = 1 \Leftrightarrow$ il existe un arc de k à j

Donc : il existe un chemin de i à j de longueur p + 1.

4.3.2.2.3 Résultat

Dans l'algorithme de Warshall, on calcule la fermeture transitive. Soit un graphe G = (X, U). La fermeture transitive de G est le graphe G* = (X, U*) tel que, il existe un arc ij de U* $\Leftrightarrow$ il existe un chemin de i à j dans G.

4.3.2.3 En bref

On prend une matrice $A = M_{n,n}$ telle que $a_{ij} = 1$ si i = j ou ij $\in$ U, 0 sinon. On va ensuite chercher pour tout sommet k si $a_{ij} = a_{ij}$ ou (a_{ik} et a_{kj}).

4.3.2.4 Algorithme de Warshall

```
1 :     Pour tout i de 1 à n faire
2 :             Pour j de 1 à n faire
3 :                     A[i, j] ← c(i, j)
4 :     Pour tout k de 1 à n faire
5 :             Pour tout i de 1 à n faire
6 :                     Pour tout j de 1 à n faire
```

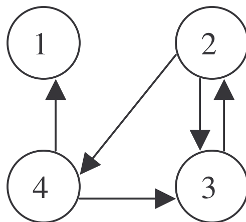
7 : A[i, j] ← A[i, j] ou (A[i, k] et A[k, j])

avec c(i, j) = 1 si ij ∈ U ou i = j, 0 sinon.

Complexité de l'algorithme : O(n^3).

4.3.2.5 Exemple

Soit le graphe suivant :



Au départ (après initialisation) on obtient la matrice $A = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{vmatrix}$.

Premier tour : k = 1. On sait que tout chemin déjà existant (i.e. à 1 dans la matrice) reste à 1. On regarde donc que les autres en regardant le résultat binaire de A[i, j] + A[i, k].A[k, j]. De plus sachant que dans les cas que l'on regarde A[i, j] = 0, il ne suffit de regarder que A[i, k].A[k, j]. Si l'une des deux valeurs est nulle, le résultat est alors nul (ET logique).

On obtient à la fin du premier tour (k = 1) la matrice $A = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{vmatrix}$.

Ensuite, pour k = 2 : $A = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{vmatrix}$, pour k = 3 : $A = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{vmatrix}$ et pour k = 4 : $A = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{vmatrix}$.

La matrice obtenue est la fermeture transitive du graphe. On rappelle qu'un graphe est fortement connexe si et seulement si la fermeture transitive est le graphe complet symétrique. (le graphe si dessus n'est donc pas fortement connexe).

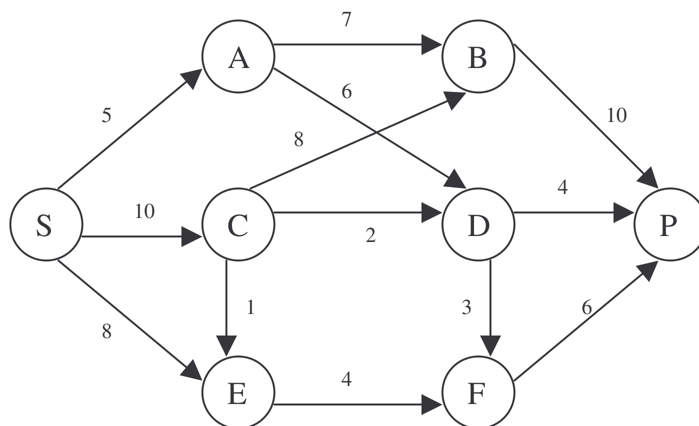
5 FLOTS, ALGORITHME DE FORD-FULKERSON

5.1 INTRODUCTION

5.1.1 Projet de réseau routier

Soit le projet de réseau routier suivant :

- G = (X, U) graphe du projet
- Les valeurs des arcs représentent un nombre maximal de véhicule à l'heure (capacités)



5.1.2 Problème

Trouver le débit horaire total maximal de véhicule susceptible de s'écouler entre les villes S et P. Le nombre de véhicules entrant dans une ville doit être égal au nombre de véhicules qui en sortent.

5.1.3 Exemple

Pour la ville D il y a :

- Deux entrants : 6 + 2 = 8
- Deux sortants : 4 + 3 = 7

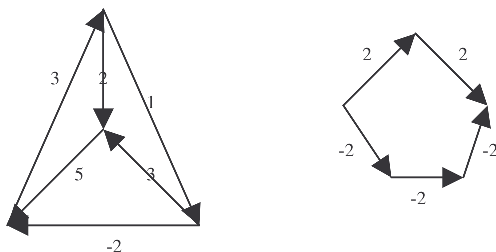
Bien que la capacité d'entrée soit 8, on ne peut en faire rentrer que 7.

5.2 FLOTS

5.2.1 Définitions

Soit G = X, U) un graphe orienté. Un flot est une application f : U → R telle que pour tout sommet de $x \in X$ on ait $\sum_{u/I(u)=x} f(u) = \sum_{u/T(u)=x} f(u)$. (loi de Kirchhoff)

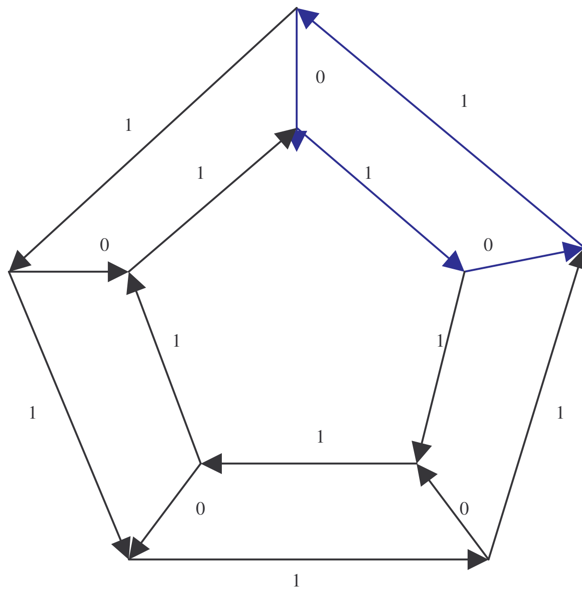
5.2.2 Exemples



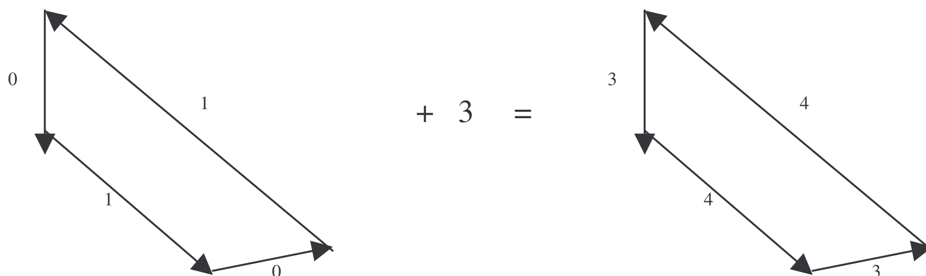
Les deux graphes si dessus vérifient la loi de Kirchhoff.

5.2.3 Remarque

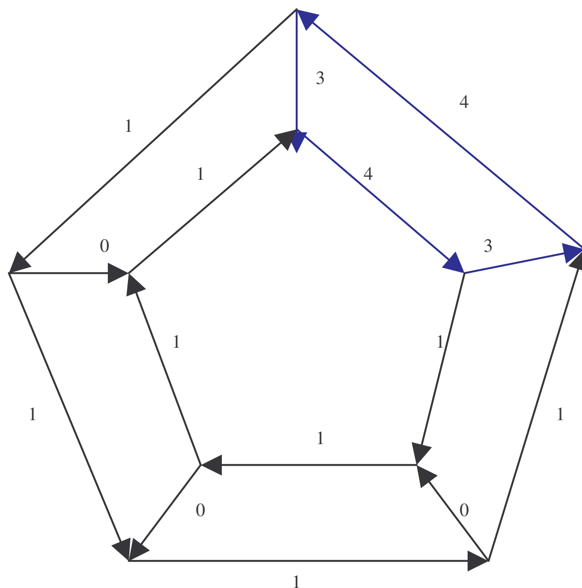
Soit le graphe G suivant :



Ce graphe respecte la loi de Kirchhoff. Si l'on prend un circuit dans le graphe (par exemple le circuit en bleu sur le schéma ci dessus) et que l'on ajoute à toutes les valeurs d'arcs la même quantité, le graphe vérifie toujours la loi de Kirchhoff.



Le graphe G' obtenu respecte toujours la loi de Kirchhoff (i.e. est un flot) :



Remarque importante : On pourra remarquer que si l'on n'a pas un circuit mais un cycle on peut conserver de même la loi de Kirchhoff. On choisit un sens de rotation dans le cycle et lorsqu'on est sur un arc « dans le sens de rotation », on ajoute la valeur et lorsqu'on est sur un arc qui est « à contre sens », on retranche la valeur.

5.3 PROBLEME DU FLOT MAXIMUM

5.3.1 Définition

Soit R = (X, U', c) le graphe issu du graphe G = (X, U) contenant deus sommets S (source) et P (puits) et tel que c : U' → R^+ U {+∞}. On note $U' = U \cup \{\overrightarrow{ps}\}$, u_r l'arc retour de P à S et $c(u_r) = +\infty$.

Le nombre d'arc est noté m.

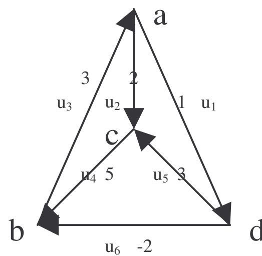
5.3.2 Problème du flot maximum

Le problème du flot maximum consiste à trouver un vecteur $f \in R^m$ avec $f : U \to R^+$ tel que :

- F est un flot (loi de Kirchhoff respectée) ;
- $O \leq f(u) \leq c(u)\ \forall\, u \in U$;
- $F(u_r)$ est maximum pour les deux conditions précédentes.

5.3.3 Exemple de flot

Soit le graphe suivant (il respecte la loi de Kirchhoff, nous allons le vérifier) :



Le vecteur f initial représente la valeur pour les arcs tels qu'ils sont notés sur le schéma. On a donc $f = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 5 \\ 3 \\ -2 \end{bmatrix}$.

On prend la matrice d'adjacence A du graphe et on la multiplie avec le vecteur f obtenu. Si le vecteur colonne résultat est nul, alors le graphe est un flot :

$$\begin{vmatrix} 1 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & -1 \\ 0 & -1 & 0 & 1 & -1 & 0 \\ -1 & 0 & 0 & 0 & 1 & 1 \end{vmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 5 \\ 3 \\ -2 \end{bmatrix} = \begin{bmatrix} 1+2-3 \\ 3-5+2 \\ -2+5-3 \\ -1+3-2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

5.3.4 Programme linéaire associé

On note F_n le programme linéaire associé tel que A.f = 0, $0 \leq f \leq c$ et $f(u_r) = z(\max)$ avec :

- A : matrice d'incidence du graphe, $A = M_{m,\,n}$ du graphe à n sommets et m arcs.
- Avec $a_{ij} = 1$ si $ij \in U$, -1 si $ji \in U$ et 0 sinon.

5.3.5 Principe de l'algorithme de Ford-Fulkerson

5.3.5.1 Idée

On appelle flot réalisable (compatible) un flot tel que $0 \leq f(u) \leq c(u)$.

On part d'un flot réalisable et on cherche un chemin de S à P. Ce chemin n'a pas d'arc saturé (rappel : un arc est saturé si $f(u) = c(u)$). Soit $\delta = \min \{ c(u) - f(u) \}$, u étant un chemin.

En faisant $f(u) = f(u) + \delta$ sur le circuit, on a :

- $f'(u) = f(u) + \delta$ si $u \in \Gamma$ (circuit = chemin + u_r)
- $f'(u) = f(u)$ si $u \notin \Gamma$

On obtient alors un flot $f(u_r) = f(u_r) + \delta$

Le but est de parcourir le graphe en faisant un marquage sur les arcs.

5.3.5.2 Marquage direct

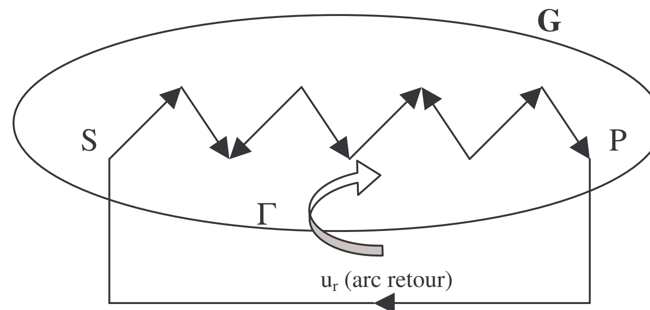
Pour le marquage direct, on part de S et on suppose qu'on marque un sommet $x \in X$ (on lui ajoute $\delta(x)$). S'il existe un arc xy = u et y non marqué et $f(u) < c(u)$ alors on marque y et $\delta(y) = \min (\delta(x), c(u) - f(u))$.

On note de plus $A(y) \leftarrow u$ (on mémorise l'arc qui mène à y).

5.3.5.3 Marquage indirect

Pour le marquage indirect, on suppose que x est marqué et que l'on a $\delta(x)$. S'il existe un arc yx = u avec y non marqué et $f(u) > 0$ alors $\delta(y) = \min (\delta(x), f(u)$, $A(y) = u$ et y devient marqué.

5.3.5.4 Conclusion



Dans le graphe G' = G U $\{u_r\}$ le graphe « réseau » et son arc retour entre le puits P et la source S, on note Γ le sens de rotation (de déplacement) dans le circuit (ou le cycle).

Dans ce cas, on note sur les arcs le flot f' tel que :

$$f' = \begin{cases} f(u) + \delta & si \quad u \in \Gamma^+ \\ f(u) - \delta & si \quad u \in \Gamma^- \\ f(u) \quad si \quad u \notin \Gamma \end{cases}$$

5.3.6 Algorithme de Ford-Fulkerson

5.3.6.1 Algorithme de Marquage

```
0 :    Marquage (G, f, c)          // G : graphe, f : flot, c : capacités.
1 :    δ ← δ(s) ← c(u_r) -  f(u_r) ; Y ← {s}          //Y : ensemble des sommets déjà marqués
2 :    Tant que p (puits) n'est pas dans Y et δ > 0 faire
3 :            S'il existe u = xy, x ∈ Y, y ∉ Y et f(u) < c(u) alors
4 :                    Y ← Y U {y}
5 :                    A(y) ← u          //Arborescence
6 :                    δ(y) ← Min { δ(x), c(u) – f(u) }
```

```
7 :		Sinon	S'il existe u = yx, x ∈ Y, y ∉ Y et f(u) > 0 alors
8 :				Y ← Y U {y}
9 :				A(y) ← u
10 :				δ(y) ← Min {δ(x), f(u)}
11 :			Sinon δ ← 0
12 :	Si p ∈ Y alors δ ← δ(p)
```

5.3.6.2 Algorithme de changement de flot

Selon que le sens de l'arc en partant de p (puits), on ajoute ou retranche δ (le delta trouvé par le calcul précédent).

```
0 :	Changement_flot (G, f, c, δ)
1 :	x ← p ; f(u_r) ← f(u_r) + δ
2 :	Tant que x ≠ s faire
3 :		u ← A(x)
4 :		Si x = T(u) alors  f(u) ← f(u) + δ et x ← I(u)
5 :			Sinon	f(u) ← f(u) - δ et x ← T(u)
```

5.3.6.3 Algorithme du flot maximum

Cet algorithme utilise les deux algorithmes précédents. Il consiste, tant que c'est possible (en fonction des capacités des arcs) en une itération de marquages et de changement de flots. L'arrêt s'effectue si δ = 0.

```
Pour tout u ∈ U faire f(u) ← 0
Itérer {
	Marquage
	Arrêt si δ = 0
	Changement de flot
	}
```

5.3.7 Coupe minimum

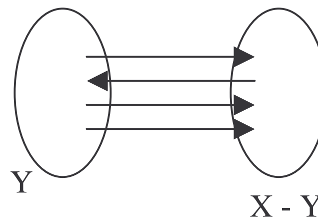
5.3.7.1 Définition d'un flot

On appelle flot dans un graphe orienté G = (X, U) l'application f : U → R tel que $\sum_{x \in T(u)} f(u) = \sum_{x \in I(u)} f(u)$.

5.3.7.2 Définition d'un cocycle

Soit Y ⊆ X, on appelle Ω(Y) cocyle, l'ensemble des arcs ayant exactement une extrémité dans Y.

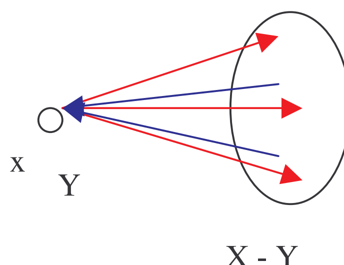
Par exemple : les arcs représentés dans le graphe ci dessous sont dans le cocyle Ω(Y).



On note :

- $\Omega^+(Y) = \{u \in U / I(u) \in Y \wedge T(u) \notin Y\}$
- $\Omega^-(Y) = \{u \in U / T(u) \in Y \wedge I(u) \notin Y\}$

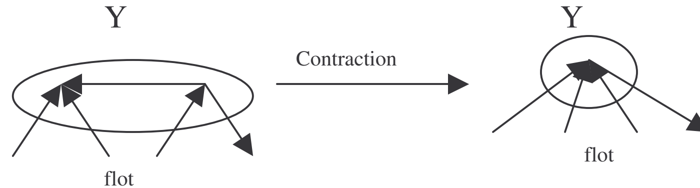
De plus, pour Y = {x}, on a (en bleu sur le schéma ci-dessous) $\Omega^-(Y)$ et (en rouge) $\Omega^+(Y)$:



5.3.7.3 Théorème

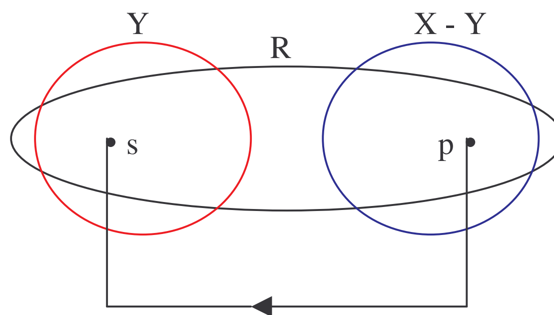
Une application f : U → R est un flot si et seulement si pour tout cocyle $\Omega(Y)$ on a $\sum_{u\in\Omega^+(Y)} f(u) = \sum_{u\in\Omega^-(Y)} f(u)$.

La preuve se fera en exercice. On peut cependant remarquer que si Y, un ensemble de sommet est un flot, par contraction, on a toujours un flot :



5.3.7.4 Coupe minimum

Soit un réseau R = (X, U, c) et u_r l'arc retour de p vers s.



C est une « coupe qui sépare p de s » si on peut trouver Y tel que Y $\subseteq$ X avec s $\in$ Y et p $\notin$ Y. Ainsi, on obtient $C = \{u \in U / I(u) \in Y \wedge T(u) \notin Y\} = \Omega^+(Y)$.

La capacité de la coupe C séparant p de s est $c(C) = \sum_{u\in C} c(u)$.

5.3.7.5 Théorème

Pour tout flot réalisable f sur R = (X, U, c) (avec $c(u_r) = \infty$ et toute coupe C séparant p de s, on a $f(u_r) \leq c(C)$.

Preuve :

Soit C une coupe qui sépare p de s. On a C = Ω^+(Y). f est un flot réalisable => f(u) ≥ 0 et f(u) ≤ c(u) ∀ u. Donc, f est un flot => $\sum_{u\in\Omega^+(Y)} f(u) = \sum_{u\in\Omega^-(Y)} f(u)$.

Donc $u \in \Omega^-(Y) \Rightarrow f(u) \leq \sum_{u\in\Omega^-(Y)} f(u) = \sum_{u\in\Omega^+(Y)} f(u) \leq \sum_{u\in\Omega^+(Y)} c(u) = c(C)$.

5.3.7.6 Corollaire

Si on termine l'algorithme de Ford-Fulkerson avec un flot f réalisable sans qu'on ait pu marquer p, f est alors une solution optimale du problème du flot maximum de s à p.

Preuve :

On fait tourner l'algorithme de Ford-Fulkerson. Il se termine sans avoir p. On a donc Y l'ensemble des sommets marqués. Si u $\in$ Ω^+(Y), on a f(u) = c(u). Si u $\in$ Ω^-(Y), on a f(u) = 0. On prend comme coupe C = Ω^+(Y) la coupe qui sépare p de s. Comme u_r est dans Ω^-(Y) et que f est un flot réalisable, on obtient :

$$f(u_r) = \sum_{u\in\Omega^-(Y)} f(u) = \sum_{u\in\Omega^+(Y)} f(u) = \sum_{u\in C} c(u) = c(C)$$ qui est donc solution optimale.

5.3.7.7 Théorème de la coupe minimum

La valeur maximum pour un flot réalisable sur R = (X, U, c) est égale à la capacité d'une coupe séparant p de s.

Pour le prouver, on a besoin de rappeler ici deux notions :

- Le problème du flot maximum qui s'écrit $F_M = \begin{cases} f = 0, u \le f \le c \\ f(u_r) = z_{\max} \end{cases}$
- Le théorème fondamental de la programmation linéaire : Si un programme linéaire admet une solution réalisable et que la fonction objective est bornée, alors il admet une solution optimale.

Ainsi :

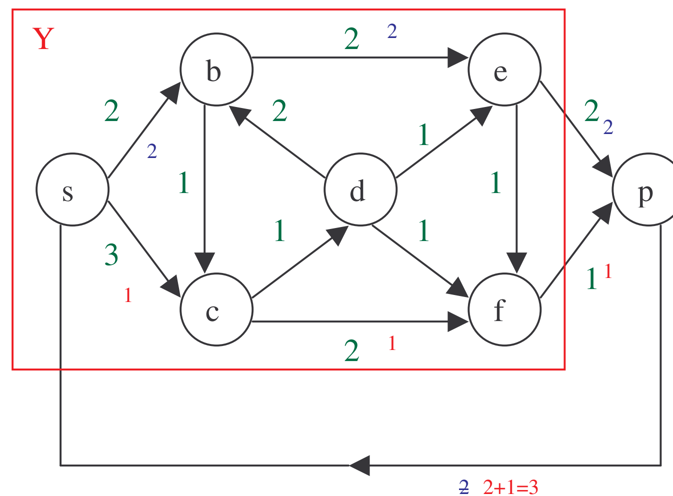
On obtient une solution optimale f du problème du flot maximum $f(u_r) \le c(C)$. On applique l'algorithme de Ford-Fulkerson au réseau en partant de $\hat{f}$ (donc on ne peut pas marquer p). Soit Y l'ensemble des sommets marqués.,

on a donc : $\hat{f}(u_r) = \sum_{u\in\Omega^-(Y)} \hat{f}(u) = c(C), C = \Omega^+(Y)$.

Comme on a $f(u_r) \le c(C)$, pour toute coupe séparant p de s, on a $C = \Omega^+(Y)$ est la coupe de capacité minimale.

5.3.8 Exemple d'application de l'algorithme de Ford-Fulkerson

Soit le réseau suivant : le flot est nul à tous les arcs au départ. Les capacités sont données en vert sur le schéma.



5.3.8.1 Première itération :

Y = {s} et $\delta(s) = \infty$.

On choisit l'arc u = sb, Y = {s, b} et $\delta(b) = \min\{\delta(s), c(u) - f(u)\} = 2$ (car $\delta(s) = \infty$ et $c(u) - f(u) = 2 - 0 = 2$). On note de plus l'arborescence A(b) = sb (i.e. on est arrivé à b par l'arc sb).

On choisit ensuite de marquer le sommet e. On a donc A(e) = de et Y = {s, b, e}. $\delta(e) = 2$.

Enfin, on choisit de marquer p et Y = {s, b, e, p} A(p) = ep et $\delta(p) = 2$.

A la fin de la première itération, on obtient alors $\delta = \delta(p) = 2$. On effectue donc le premier changement de flot, en partant de p et en remontant dans l'arborescence créée, on ajoute δ au flot initial (On rappelle qu'à l'initialisation, les flots de tous les arcs sont mis à 0).

On obtient les flots marqués en bleus sur le schéma.

5.3.8.2 Deuxième itération

On recommence : Y = {s}. De là, on ne peut plus marquer b car f(b) = c(b). On marque alors c. On obtient alors A(c) = sc, Y = {s, c} et δ(c) = min {δ(s), c(u) – f(u)} = 3 car δ(s) = ∞ et c(u) – f(u) = 3 – 0 = 3.

On peut ensuite marquer f en utilisant l'arc u = ef : A(f) = ef, Y = {s, c, f} et δ(f) = min {δ(c), c(u) – f(u)} = 2 car δ(c) = 3 et c(u) – f(u) = 2 – 0 = 2.

Enfin, on marque p avec l'arc u = fp. Y = {s, c, f, p} et δ(p)=1. Donc, δ=1, on ajoute 1 à tous les flots en remontant dans l'arborescence (ne pas oublier le cycle avec l'arc u_r). On obtient les flots indiqués en rouge.

5.3.8.3 Troisième itération

Y = {s} et δ(s) = ∞. On peut encore accéder à c. Donc on passe par l'arc u = sc. Y = {s, c} et δ(c) = 2. Puis on va vers d car c'est celui d'on le flot est le plus faible (encore à 0) par l'arc cd. On passe donc par u = cd. On a alors A(d) = cd et δ(d) = 1.

Puis on décide d'aller vers b. A(b) = db et δ(b) = 1.

Puis, par d on peut aussi accéder à e. Donc A(e) = de, δ(e) = 1 et Y = {s, c, d, b, e}.

Enfin, on peut accéder à f. Donc A(f) = ef, δ(f) = 1 et Y = {s, c, d, b, e, f}. Mais, ensuite, on est bloqué, on ne peut plus accéder à p car tous les flots sont maximum. L'ensemble des sommets marqués ainsi obtenus par cette dernière itération permet d'obtenir Y (encadré en rouge sur le schéma ci dessus).

$C = \Omega^+(Y)$ et c(C) = 3 = $f(u_r)$. On a donc obtenu le résultat optimal. C est la coupe minimum.

5.3.9 Complexité de l'algorithme de Ford-Fulkerson

On cherche une chaîne améliorante (ou augmentante). Chaque arc est considéré une seule fois :

- Si les capacités sont entières, on augmente $f(u_r)$ d'au moins une unité. Si la valeur du flot maximum est $v_M f(u_r)$, la complexité est en $O(m. v_M f(u_r))$, avec m le nombre d'arêtes.
- Si les capacités sont réelles, il peut y avoir des boucles infinies. On peut cependant calculer la capacité minimum d'une coupe qui sépare p de s. Si on a n sommets, on a 2^{n-2} coupes.

Quelques complexités obtenues au cours d'études d'application de cet algorithme :

- Edmonds Karpe : 1970 – 72 → $O(n^5)$
- Dimic : 1970 → $O(n^2m)$
- Kazamov : 1974 → $O(n^3)$
- Sleater : 1980 → $O(nm\log^2 n)$

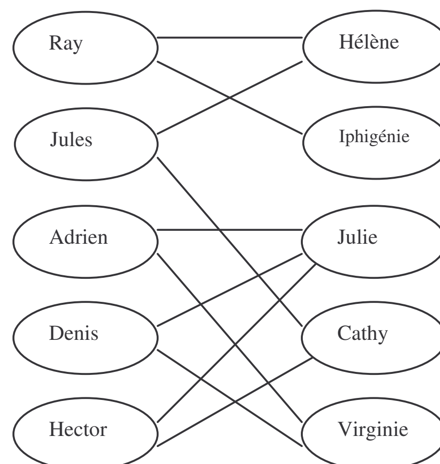
5.4 APPLICATION DU FLOT MAXIMUM

5.4.1 Définition du problème

Supposons que l'on ait une agence matrimoniale dans laquelle sont enregistrés H hommes et F femmes. Après des entretiens organisés par l'agence, on constate que certains mariages sont possibles et pas d'autres. Ces mariages peuvent être représentés par le tableau et le graphe suivant.

Le but de l'agence est d'organiser le maximum de mariage.

	Hélène	Iphigénie	Julie	Cathy	Virginie
Ray	X	X			
Jules	X			X	
Adrien			X		X
Denis			X		X
Hector			X	X	



On veut trouver un couplage maximum. Le problème consiste donc en la recherche d'un couplage maximum dans un graphe biparti.

5.4.2 Définition

Soit un graphe G = (X ; E) non orienté. Si X = Y U Z, tel que $Y \cap Z = \varnothing$ (i.e. le graphe est biparti), Y et Z sont des ensembles indépendants (stables).

5.4.3 Théorème

Un graphe est biparti ⇔ il ne contient pas de cycle de longueur impaire.

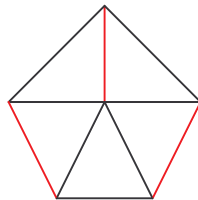
- Preuve => évidente
- Preuve <= voir exercice

5.4.4 Définitions

5.4.4.1 Définition 1

Un couplage dans un graphe est un ensemble K d'arêtes 2 à 2 non adjacentes.

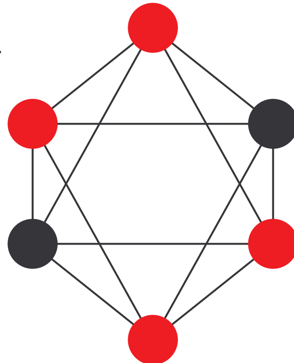
Exemple : (K en rouge)



5.4.4.2 Définition 2

Un transversal est un ensemble de sommets T tel que X – T soit un stable (i.e. toute arête a au moins une extrémité dans T)

Exemple : En rouge : T transversal.



5.4.4.3 Définition 3

K est un couplage et T est un transversal => chaque arête a au moins une extrémité dans T.

$|K| \leq |T|$.

Si $|K| = |T|$ on a un couplage maximum et un transversal minimum.

5.4.5 Résolution du problème

Dans le problème de l'agence matrimoniale, il s'agit donc de trouver un couplage maximum dans un graphe biparti. Pour cela, on va dans un premier temps transformer le graphe pour y appliquer ensuite l'algorithme de recherche du flot maximum.

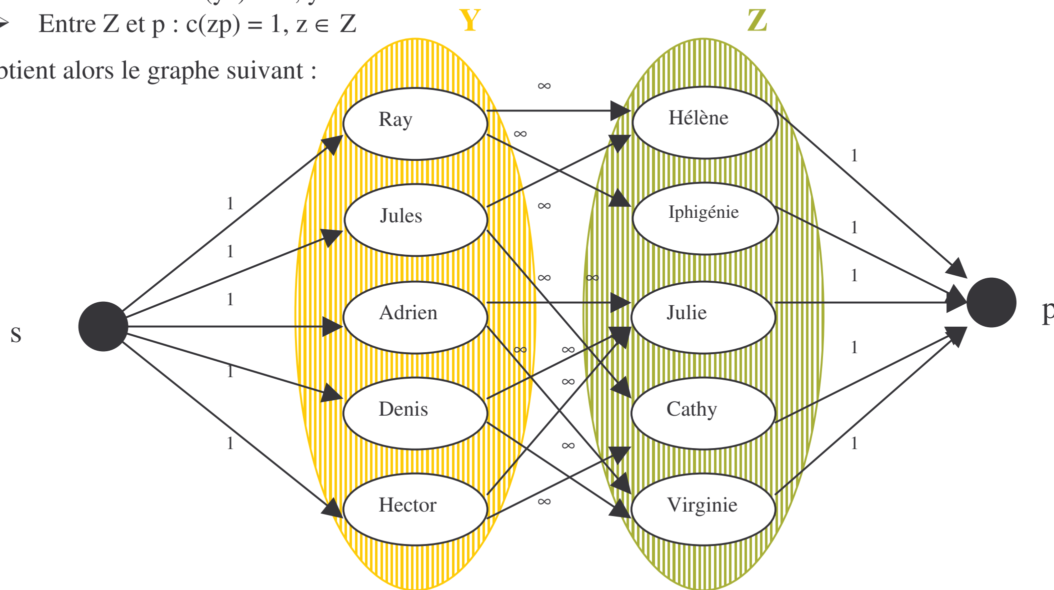
On ajoute donc un sommet source s et un sommet puits p. L'ensemble U des arcs est obtenu de la façon suivante :

- On oriente tous les arcs de s à Y
- On oriente tous les arcs de Y à Z
- On oriente tous les arcs de Z à p.

On ajoute ensuite les capacités :

- Entre s et Y : c(sy) = 1, y ∈ Y
- Entre Y et Z : c(yz) = ∞, y ∈ Y et z ∈ Z
- Entre Z et p : c(zp) = 1, z ∈ Z

On obtient alors le graphe suivant :



Il reste alors à résoudre le problème du flot maximum sur ce graphe (ce qui ne sera pas fait ici → voir le paragraphe concernant la résolution du flot maximum).

5.4.5.1 Quelques remarques

La loi de Kirchhoff implique que f(u) = 0 ou 1 si u = yz, y ∈ Y et z ∈ Z.Deux arcs yz et yz' tels que f(yz) = 1 et f(yz') = 1 sont des arcs non adjacents et K = {e ∈ E / f(e) = 1} est un couplage de G.

Considérons la coupe Ω(V) ou V = Y U {s}, alors :

- $f(u_r) + \sum_{u \in \Omega^-(V)} f(u) = \sum_{u \in \Omega^+(Y)} f(u)$
- $f(u_r) + \sum_{u \in \Omega^-(V)} f(u) = \sum_{u \in \Omega^+(V)} f(u)$
- $f(u_r) = \sum_{u \in \Omega^+(V)} f(u) = \sum_{c \in K} f(e) = |K|$

Soit S tel que $\Omega^+(S)$ = C la coupe de capacité minimum. $\Omega^+(S)$ ne contient pas d'arc de capacité infinie.

Si $\overrightarrow{yz} \in E, y \in \overline{S} \wedge z \in S$ alors $T = (\overline{S} \cap Y) \cup (S \cap Z)$ est un transversal de G. Les arcs Ω(S) $\overrightarrow{sy}, y \in \overline{S} \cap Y$ et $\overrightarrow{zp0}, p \in S \cap Z$. Les |T| arcs sont de capacités égale à 1. La capacité $c(\Omega(S)) = |T|$ et $c(C) = f(u_r) = |K|$. On obtient alors $|T| = |K|$ et K est un couplage maximum.

5.4.5.2 Théorème

Dans un graphe biparti, le nombre de sommets d'un transversal minimum est égal aux nombres d'arêtes d'un couplage maximum.