DESS MINT
Mise à niveau
"Graphes et Réseaux"

1 Généralités sur les graphes

1.1 Le concept de graphe

D'une façon formelle, un graphe $G = (X, A)$ est le couple constitué:
(i) par un ensemble fini $X = \{x_1, x_2, \dots, x_n\}$ d'éléments appelés *sommets*;
(ii) par un sous-ensemble fini $A = \{a_1, a_2, \dots, a_m\}$ dont les éléments sont des *couples ordonnés* de sommets appelés *arcs.* A est donc une famille d'éléments du produit cartésien

$$X \times X := \{(x, y) : x, y \in X\}.$$

Pour un arc $a = (x, y)$, le sommet x est son *extrémité initiale* et le sommet y son *extrémité terminale.* Dans ce cas, on dit que y est le *successeur* de x. D'autre part, x est le *prédécesseur* de y. L'ensemble des successeurs de x est noté $\Gamma^+(x)$ est l'ensemble de prédécesseurs de x est noté $\Gamma^-(x)$. Soit

$$\Gamma(x) := \Gamma^+(x) \cup \Gamma^-(x).$$

Pour étudier certaines propriétés "non orientées" des graphes, il peut arriver que la direction des flèches n'importe pas; seul importe de savoir les paires de sommets reliées. De ce fait, on considère au lieu de l'arc $a = (x, y)$, l'ensemble formé par les points x et y, que l'on notera $e = xy$, et que l'on appellera *arête* du graphe. On dit que x et y sont *incidents* à e, que x et y sont les *extrémités* de e et que x et y sont *adjacents.* La famille des m arêtes du graphe G sera notée par E.

Le *graphe complet* (ou *clique*) à n sommets, noté K_n, est le graphe non orienté dont deux sommets quelconques sont adjacents. Il est donc de taille $n(n-1)/2$.

On appelle *graphe partiel* de $G = (X, E)$ un graphe ayant même ensemble de sommets X que G et ayant pour ensemble d'arêtes une partie de E.

Étant donné un sommet x d'un graphe G non orienté, on appelle *degré* de x et on note $d(x)$ le nombre d'arêtes incidentes à x. Les autres extrémités de ces arêtes sont les *voisins* de x.

Proposition 1.1. *Pour un graphe G non orienté*

$$\sum_{x \in X} d(x) = 2m.$$

En particulier, le nombre des sommets de G avec un degré impair est toujours pair.

Preuve. Si on calcule la somme des degrés de toutes les sommets de G, chaque arête sera comptée deux fois, donc

$$m = \frac{1}{2} \sum_{x \in X} d(x).$$

Si x est un sommet d'un graphe G orienté, on appelle *demi-degré extérieur* de x, noté $d^+(x)$, le nombre d'arcs d'origine x; le demi-degré intérieur de x, noté $d^-(x)$, est le nombre d'arcs admettant x comme extrémité terminale.

Soit $G = (X, E)$ un graphe non orienté. On appelle *chaîne* (chemin) une suite $x_0 e_1 x_1 e_2 \dots x_{k-1} e_k x_k$ avec $x_0, x_1, \dots, x_k \in X$, $e_1, e_2, \dots, e_k \in E$, et $e_i = x_{i-1} x_i$ pour $i = 1, \dots, k$.

Une chaîne est dite *élémentaire* si tous les sommets de cette chaîne sont deux à deux distincts. La *longueur* d'une chaîne élémentaire est le nombre d'arêtes dans cette chaîne. Un *cycle* est une chaîne dont les deux extrémités coïncident. Un graphe est dit *connexe* si, pour toute paire de sommets, il existe une chaîne les joignant. On dit que deux sommets d'un graphe G sont équivalents si et seulement s'ils sont extrémités d'une même chaîne dans G. Les classes d'équivalence de G s'appellent *composantes connexes* de G. Dans un graphe connexe G, la distance $d_G(x,y)$ entre deux sommets x et y est la longueur d'une plus court chemin ayant x et y comme extrémités.

Un *arbre* est un graphe connexe et sans cycle.

Proposition 1.2. *Pour un graphe G avec n sommets et m arêtes, les conditions suivantes sont équivalentes:*

(i) G est un arbre;

(ii) G est connexe et on a $m = n - 1$;

(iii) G est sans cycle et on a $m = n - 1$;

(iv) G est connex et la suppression d'une arête quelconque le déconnecte;

(v) G est sans cycle et l'ajout d'une arête quelconque crée un cycle;

(vi) entre deux sommets quelconques, il existe une chaîne élémentaire unique.

Preuve. Pour prouver les implications suivantes (i)⇒(ii)⇒(iv)⇒(vi) et (i)⇒(iii)⇒(v)⇒(vi)⇒(i), nous établissons d'abors deux lemmes.

Lemme 1.1. *Si G est connexe, alors $m \geq n - 1$.*

Preuve du Lemme 1.1. Récurrence sur n. Le résultat est évident pour $n = 1, 2$. Supposons il est vrai pour $n' < n$ et soit $G = (X, E)$ un graphe connexe avec n sommets. Soit $x \in X$. Supposons que le graphe $G - x := (X - \{x\}, E - \Gamma(x)\}$ contient k parties connexes $G_1, G_2, \ldots, G_k$. D'après l'hypothèse de récurrence, $m_i \geq n_i - 1$ pour chaque G_i (m_i est n_i sont le nombre d'arêtes et le nombre de sommets du graphe G_i). Puisque G est connexe, x a au moins un voisin dans chacun des G_i; donc

$$m = \sum_{i=1}^{k} m_i + d(x) \geq \sum_{i=1}^{k} (n_i - 1) + k = n - 1.$$

Lemme 1.2. *Si G est sans cycle, alors $m \leq n - 1$.*

Preuve du Lemme 1.2. Récurrence sur n. Si dans un graphe fini tout sommet est de degré au moins 2, alors le graphe contient au moins un cycle, puisque dans une promenade on peut toujours repartir d'un sommet par une arête autre que celle par laquelle on est arrivé, donc après un nombre fini d'etapes on doit visiter un sommet pour la deuxième fois. Donc si un graphe est sans cycle, il contient un sommet de degré 0 ou 1. Soit x un tel sommet. Alors $G - x$ est sans cycle et par hypothèse de récurrence, la taille de $G - x$ est au plus $n - 2$, donc la taille de G est au plus $n - 1$.

1.2 Représentation des graphes en machine

Supposons les sommets de G numérotés $x_1, \dots, x_n$ et les arêtes de G numérotées $e_1, e_2, \dots, e_m$.

Matrices d'adjacence (sommets-sommets) On représente le graphe par une matrice carrée Adj à n lignes et n colonnes, telle que

(a) pour les graphes non orientés:

$$Adj(i,j) = 1 = Adj(j,i) \text{ si } x_i \text{ et } x_j \text{ sont adjacents}$$

$$Adj(i,j) = 0 \text{ sinon .}$$

(b) pour les graphes orientés:

$$Adj(i,j) = 1 \text{ et } Adj(j,i) = -1 \text{ si } (x_i, x_j) \text{ est un arc de } G$$

$$Adj(i,j) = 0 \text{ sinon.}$$

Matrices d'incidence (sommets-arêtes) On représente le graphe par une matrice Inc à n lignes et m colonnes, telle que

(a) pour les graphes non orientés:

$$Inc(i,j) = 1 \text{ si } x_i \text{ et } e_j \text{ sont incidents}$$

$$Inc(i,j) = 0 \text{ sinon.}$$

(b) pour les graphes orientés:

$$Inc(i,j) = +1 \text{ si } x_i \text{ est l'origine de l'arc } e_j$$

$$Inc(i,j) = -1 \text{ si } x_i \text{ est l'extrémité finale de l'arc } e_j$$

$$Inc(i,j) = 0 \text{ sinon .}$$

Tableau de listes d'adjacence (pour les graphes non-orientés) On définit un tableau de pointeurs de taille n. Le pointeur en position i est la tête d'une liste chaînée qui contient les voisins du sommet x_i : un voisin x_j de x_i doit appartenir à la liste correspondant au sommet x_i et x_i à la liste de x_j.

2 L'arbre couvrant de poids minimum

Étant donné un graphe non orienté dont les arêtes sont munies d'une valuation ("coût", "poids" ou "longueur"), on cherche un graphe partiel de ce graphe qui soit un arbre et qui soit de coût minimum. On dit qu'il s'agit d'un *arbre couvrant* ("minimum spanning tree").

Une application aux réseaux (de communication ou d'interconnexion): ayant estimé le coût des liaisons directs entre toutes les paires d'objets à relier, on cherche à réaliser un réseau connexe de coût minimum.

2.1 Algorithme de Kruskal (dit "algorithme glouton")

Pour déterminer un arbre couvrant de poids minimum d'un graphe connexe à n sommets, on sélectionne les arêtes en itérant $n-1$ fois l'opération suivante: choisir une arête de poids minimum ne formant pas cycle avec les arêtes précédemment choisies.

L'algorithme de Kruskul se déroule en deux phases:

(i) trier les arêtes par ordre de poids croissants;

(ii) tant que l'on n'a pas retenu $n-1$ arêtes, considérer, dans l'ordre du tri, la première arête non examinée; si elle forme un cycle avec les arêtes précédemment choisies, la rejeter, sinon la garder.

Pour une implémentation de la phase (ii) nous devons résoudre le problème suivant: comment déterminer si une arête xy choisie à l'étape i forme un cycle avec les arêtes précédemment choisies? Pour que xy ferme un cycle, il faut que précédemment ses extrémités soient reliées par une chaîne, et donc aient été dans une même composante connexe. Donc il faudra gérer l'évolutions des composantes connexes, au fur et à mesure du choix des arêtes. Initialement, chaque sommet constitue une composante connexe et nous initialisons pour chaque sommet x_i l'indice i. Chaque fois qu'une arête xy est candidate, on compare les valeurs des indices de x et y. Si elles sont égales, l'arête xy créerait un cycle et on ne la retient pas. Sinon on garde l'arête xy et on donne comme valeur à l'indice associé à composante de y l'indice associé à x, ceci voulant dire que toutes les sommets de la composante connexe de x et de la composante connexe de y, après le choix de xy forment une seule composante et portent donc le même indice de composante connexe. En tenant compte du tri initial, on voit que la complexité de l'algorithme de Kruskal est de $O(n^2 + mlogn)$. En utilisant des structures de données dynamiques, on peut obtenir une implémentation en $O((n+m)logn)$.

Proposition 2.1. *L'algorithme de Kruskal retourne un arbre couvrant de poids minimum de G.*

Preuve. Le résultat est clairement un arbre couvrant T du graphe G. Supposons qu'il ne soit de poids minimum. Représentons T par sa suite des arêtes dans l'ordre où elles ont été sélectionnées: $T = (e_1, e_2, \ldots, e_{n-1})$. Parmi les arbres couvrantes de poids minimum, choisissons en un, T_0, coincidant avec T le plus longtemps possible: $T_0 = (e_1, \ldots, e_k, f_{k+1}, \ldots, f_{n-1})$. Comme $e_{k+1} \notin T_0$, donc $T_0 \cup \{e_{k+1}\}$ contient un cycle (voir Proposition 1.2.). Ce cycle n'est pas constitué uniquement d'arêtes de T puisque T est sans cycle. Soit f_i une arête de ce cycle n'appartenant pas à T. Alors $T_0 \cup \{e_{k+1}\} - \{f_i\}$ est à nouveau un arbre couvrant de G et, puisqu'il coincide davantage avec T que T_0, il n'est pas de poids minimum. Donc le poids de e_{k+1} est strictement supérieur au poids de f_i. Mais alors on aurait dû examiner f_i avant e_{k+1} et comme f_i a été pas pris en T, elle doit former un cycle avec les arêtes $e_1, \ldots, e_k$, une contradiction avec l'inclusion $e_1, \ldots, e_k, f_i \in T_0$.

2.2 Algorithme de Prim

Le principe de l'algorithme de Prim est le suivant: on étend de proche en proche un arbre couvrant une partie des sommets du graphe, en atteignant un sommet supplémentaire à chaque étape, et en prenant pour cela l'arête la plus légère parmi celles qui joignent l'ensemble des sommets déjà couverts à l'ensemble des sommets non encore couverts. On appelle T l'arbre en cours de construction et on note $p(x,y)$ le poids de l'arête xy. À chaque étape, on adjoint à T un sommet et une arête. Soit R l'ensemble des sommets non encore dans T. Pour chaque sommet x de R soit p_x le sommet de T qui est tel que le poids de l'arête xp_x soit minimum sur l'ensemble des sommets de T : ce poids est dit *distance* $d(x)$ de x à T. On choisit alors de faire entrer dans T le sommet x de R dont la distance $d(x)$ à T est minimum. Ce sommet sera le *pivot* de l'étape suivante. On met ensuite à jour les attributs des sommets z qui restent dans R et qui sont voisins de pivot, en comparant l'ancienne distance $d(y)$ de y à T avec $p(pivot, y)$: si $p(pivot, y) < d(y)$ est plus petit que $d(y)$, alors faire $p_y \leftarrow pivot$ et $d(z) \leftarrow p(pivot, y)$. La complexité de l'algorithme est de $O(n^2)$.

Proposition 2.2. *L'algorithme de Prim retourne un arbre couvrant de poids minimum de G.*

Preuve. Appelons $T = (e_1, \ldots, e_k, e_{k+1}, \ldots, e_{n-1})$ l'arbre couvrant construit par l'algorithme de Prim. Supposons qu'il n'est pas de poids minimum, et soit T_0 un arbre couvrant de poid minimum coincidant aussi loin que possible avec T, c'est-à-dire que $T_0 = (e_1, \ldots, e_k, f_{k+1}, \ldots, f_{n-1})$ avec k le plus grand possible. L'arête e_{k+1} n'appartient pas à T_0, donc $T_0 \cup \{e_{k+1}\}$ contient un cycle C. Au moment où on a choisi l'arête e_{k+1} dans T, on a relié une partie R de G a $G - R$. Si on considère la chaîne $C - e_{k+1}$, elle joint R à $G - R$, donc au moins une de ses arêtes a une extrémité dans R et l'aute dans $G - R$. Soit f une telle arête. Les arêtes $e_1, \ldots, e_k$ ayant toutels les deux extrémités dans $G - R$, f n'est pas l'une d'entre elles. D'autre part, f étant une arête de $C - e_{k+1}$, elle est dans T_0. D'après le choix de e_{k+1} on a $poids(f) \geq poids(e_{k+1})$. Si l'inégalité est stricte, l'arbre $T' := (T_0 \cup e_{k+1}) - f$ est de poids strictement inférieur à celui de T_0, une contradiction avec le fait que T_0 est de poids minimum. S'il y égalité, l'arbre T' est de poids minimum et coincide davantage evec T que T_0, une contradiction avec le choix de T_0.

3 Graphes bipartites et graphes planaires

On dit qu'un graphe G est *planaire* s'il est possible de le représenter sur un plan de sorte que les sommets soient des points distincts, les arêtes des courbes simples, et deux arêtes ne se rencontrent pas en dehors de leurs extrémités. Une telle représentation de G s'appele un *graphe planaire topologique* soit une *carte géographique.* Une *face* de G est une région de plan limitée par des arêtes et telle que deux points arbitraires dans cette région peuvent toujours être reliés par un trait continu ne rencontrant ni sommets ni arêtes. On notera qu'il y a toujours une *face infinie.*

Proposition 3.1. (formule d'Euler) *Si, dans un graphe planaire topologique connexe G, il y a n sommets, m arêtes et f faces, alors on a*

$$n - m + f = 2.$$

Preuve. Le résultat est vrai si G est acyclique, donc un arbre (voir la Proposition 1.2.). Supposons le résultat vrai pour tout graphe de $\leq f - 1$ faces et montrons à partir de là qu'il est vrai pour un graphe planaire topologique G avec f faces. Un arête e de G appartient à deux faces de G. En supprimant e on obtient un graphe G' avec $n' = n$ sommets, $m' = m - 1$ arêtes et $f' = f - 1$ faces, donc il vérifie la formule d'Euler $n' - m' + f' = 2$. Donc $n - (m - 1) + (f - 1) = 2$ et par corollaire $n - m + f = 2$.

Corollaire 3.1. *Si G est un graphe planaire avec n sommets, m arêtes et f faces, alors $m \leq 3n - 6$ et $f \leq 2n - 4$. Par conséquence, G contient un sommet x de degré $d(x) \leq 5$.*

Preuve. Supposons sans perdre la généralité que G est un graphe planaire avec un nombre maximum d'arêtes et des faces (sinon, on peut transformer G dans un tel graphe en ajoutant des arêtes). Donc chaque face de G est un triangle et comme chaque arête appartient à deux faces, par conséquence, on $3f = 2m$, i.e. $f = 2m/3$. D'après la formule d'Euler, on peut écrire $n - m + 2m/3 = 2$, donc $m = 3n - 6$. De même façon, $f = 2n - 4$.

Si tout sommet est l'extrémité d'au moins 6 arêtes, alors d'après la Proposition 1.1. $2m \geq 6n$, d'où l'absurdité.

On appelle *nombre chromatique* $\gamma(G)$ d'un graphe G le plus petit nombre de couleurs nécessaire pour colorier les sommets, de sorte que deux sommets adjacents distincts ne soient pas de même couleur. Un des problèmes célèbres (ouvert pendant longtemps) est le suivant: est-il possible de colorier tout graphe planaire en 4 couleurs? Ici, au moins, on peut affirmer:

Proposition 3.2. *Si G est un graphe planaire, alors il est 5-colorable.*

Preuve. Supposons cet énoncé vérifié pour tout graphe planaire avec $< n$ sommets, et démontrons-le pour un graphe G avec n sommets. D'après le Corollaire 3.1., il existe un sommet x de degré ≤ 5. Colorions $G - x$ avec 5 couleurs $c_1, \ldots, c_5$. La coloration de x avec une de ces couleurs est immédiate, sauf si $d(x) = 5$ et les 5 voisins v_1, v_2, v_3, v_4, v_5 de x (supposés dans cet ordre en tournant autour de x) sont tous de couleurs différentes. Supposons que v_i soit de couleur c_i pour $i = 1, \ldots, 5$. Soit G_{ij} le sous-graphe de G constitué par les sommets de couleurs i et j. La composante connexe de G_{13} qui contient v_1 contient aussi v_3 (dans le cas contraire, on pourrait interchanger c_1 et c_3 dans cette composante, et colorier x avec c_1). De même, la composante G_{14} qui contient v_1 contient aussi v_4. Donc, dans le graphe bicolore G_{25}, les sommets v_2 et v_5 ne peuvent être connectés par une chaîne, et en interchangeant le couleurs c_2 et c_5 dans G_{25}, on pourra colorier x avec c_2.

Un graphe $G = (X, E)$ est un *graphe biparti* s'il est possible de partitionner X dans deux sous-ensembles X_0 et X_1 tels que chaque arête ait exactement une extrémité dans X_0 et une extrémité dans X_1. Remarquons qu'un graphe G est biparti si est seulement si il est bichromatique, c'est-à-dire ssi $\gamma(G) = 2$.

Proposition 3.3. *Un graphe G est biparti si est seulement si toutes les cycles de G sont de longueur paire.*

Preuve. La preuve qui dans un graphe biparti toutes les cycles sont pairs est simple. Pour l'inverse, on choisi un sommet x_0 de G et soit

$$X_0 = \{x \in X : d(x_0, s) \equiv 0(\mod 2)\}, \quad X_1 = \{x \in X : d(x_0, x) \equiv 1(\mod 2)\}.$$

Si deux sommets dans la même partie X_0 or X_1 sont adjacentes, on obtient un cycle impair, une contradiction. Donc G est un graphe biparti.

4 Parcours de graphes

4.1 Parcours en largeur

Le *parcours en largeur* (Breadth First Search) est la base de nombreux algorithmes importantes sur les graphes. Etant donnés un graphe $G = (X, A)$ et un sommet *origine* s, le parcours en largeur emprunte systématiquement les arcs de G pour découvrir tous les sommets accessibles depuis s. Il calcule la distance entre s est tous les sommets accessibles. Il construit également une *arborescence en largeur* de racine s. Pour tout sommet v accessible depuis s, le chemin reliant s à v dans cette arborescence est un plus court chemin de s vers v.

Le parcours en largeur colorie chaque sommet en blanc, gris, ou noir. Tous les sommets sont blancs au départ, et peuvent devenir gris, puis noirs. Un sommet est *decouvert* la première fois qu'il est rencontré au cours de la recherche; il perd alors sa couleur blanche. Les sommets gris ou noirs ont été découverts, et la distinction entre gris et noir permet de garantir que la recherche se poursuit en largeur d'abord. Si $(u, v) \in A$ et si le sommet u est noir, alors le sommet v est gris ou noir. Autrement dit, tout sommet adjacent à un sommet noir a déjà été découvert. Les sommets gris peuvent avoir des sommets adjacents blancs; ils représentent la frontière entre les sommets découverts et les sommets non découverts.

Pour implémenter ce parcours on utilise une file d'attente F. Au départ F contient seulement le sommet origine s. A une étape de parcours en largeur, on examine le sommet u en tête du F. A chaque découverte d'un sommet v blanc pendant le balayage de la liste d'adjacence de u, le sommet v et l'arc (u, v) sont ajoutés à l'arborescence, ensuite v devient gris et il est ajouté à la queue de la file F. Après le balayage complet de $\Gamma^+(u)$ le sommet u est supprimé de la liste F et il est coloré en noir. On dit que u est le *père* de v dans l'arborescence en largeur.

4.2 Parcours en profondeur

La stratégie d'un parcours en profondeur DFS(r) (Depth First Search) à partir d'un sommet r est de descendre plus profondément dans le graphe chaque fois que c'est possible. Lors d'un parcours en profondeur, les arcs sont explorés à partir du sommet v découvert le plus récemment et donc on n'a pas encore exploré tous les arcs incidents. Lorsque tous les arcs de v ont été explorés (voir marqués), l'algorithme revient en arrière pour explorer les arcs qui partent du sommet u à partir duquel v a été découvert (le père de v). Ce processus se répète jusqu'à ce que tous les sommets accessibles à partir de r aient été découverts. Chaque sommet est initialement blanc, puis grisé quand il est découvert pendant le parcours, et enfin noirci en fin de traitement, c'est-à-dire quand sa liste d'adjacence a été complètement examinée.

L'algorithme peut s'écrire de façon itérative (sa complexité est $O(|X| + |A|)$) :

Parcours en profondeur
tant que l'algorithe n'est pas terminé:
  noter u le sommet courant
  s'il existe un arc (u, v) non encore traversé, alors
    traverser l'arc (u, v)
    si v est blanc, alors
    grisér v et poser $pere(v) = u$
    avancer en v, c'est-à-dire poser que le sommet courant est v
  sinon
    noircir u
    si $u \neq r$, reculer au père de u, c'est-à-dire que le sommet courant devient $pere(u)$
  sinon, l'algorithme est terminé.

Comme dans le parcours en largeur, le sous-graphe de liaison “père-fils” forme un *arbre en profondeur* (en effet, si G n'est pas connexe, on obtient une forêt en profondeur).

Le parcours en profondeur *date* chaque sommet. Chaque sommet v porte deux dates: la première, $g(v)$, marque le moment où v a été découvert pour la première fois (et colorié en gris), et la seconde, $n(v)$, enregistre le moment ou le parcours a fini d'examiner la liste d'adjacence de v (et le colorie en noir). Un propriété importante du parcours en profondeur est que la date de découverte et celle de fin de traitement des sommets ont une *structure parenthésée*:

pour deux sommets quelconques u et v, une et une seule des conditions ci-dessous est vérifiée:

- les intervalles $[g(u), n(u)]$ et $[g(v), n(v)]$ sont disjoints;
- l'intervalle $[g(u), n(u)]$ est inclus dans l'intervalle $[g(v), n(v)]$ et u est un descendant de v dans l'arbre en profondeur;
- l'intervalle $[g(v), n(v)]$ est inclus dans l'intervalle $[g(u), n(u)]$ et v est un descendant de u dans l'arbre en profondeur.

Dans un parcours en profondeur d'un graphe non orienté $G = (V, E)$ chaque arête de G est soit une arête de liaison, soit une arête de retour.

Lemme 4.1. *Dans un arbre en profondeur d'un graphe orienté $G = (X, A)$ un sommet v est un descendant d'un sommet u ssi à la date $g(u)$ il existe un chemin de G constitué de sommets blancs de u à v.*

Preuve. $\Rightarrow$ Supposons que v soit un descendant de u. Soit w un sommet quelconque sur le chemin de u à v dans l'arbre en profondeur. Comme w est un descendant de u, d'après le parcours en profondeur on a $g(u) < g(w)$, donc w est blanc à la date $g(u)$.

$\Leftarrow$ Supposons qu'à la date $g(u)$ il existe un chemin blanc P de G de u à v, mais que v ne devienne pas un descendant de u dans l'arbre en profondeur. On peut, sans perte de généralité, supposer que tous les autres sommets du chemin sont descendants de u. Soit w le sommet de P adjacent avec v. Comme w est un descendant de u et v n'est pas un descendant de u, on deduit que $n[w] \leq n[u]$ et que v doit être découvert après u, mais avant que le traitement de w soit terminé. Donc $g(u) < g(v) < n(w) \leq n(u)$. La structure parenthésée implique alors que l'intervalle $[g(v), n(v)]$ est entiérement inclus dans l'intervalle $[g(u), n(u)]$, donc v est finalement un descendant de u.

Une autre propriéte intéressante du parcours en profondeur est que le parcours peut servir à classer les arc du graphe $G = (X, A)$.

1. Les *arcs de liaison* sont les arcs de l'arbre (la forêt) en profondeur.
2. Les *arcs retour* sont les arcs (u, v) reliant un sommet u à un ancêtre v dans l'arbre en profondeur.
3. Les *arcs avant* sont les arcs (u, v) qui ne sont pas des arcs de liaison, et qui reliant un sommet u à un descendant v dans l'arbre en profondeur.
4. Les *arcs couvrants* sont tous les autres arcs. Ils peuvent relier deux sommets tant que l'un des sommets n'est pas ancêtre de l'autre; ils peuvent aussi relier deux sommets appartenant à des arborescences en profondeur différentes.

Lorsqu'on effectue un parcours en profondeur, on peut associer d'une part ce qu'on appelle la *numérotation préfixe* des sommets, d'autre part la *numérotation postfixe* des sommets. Dans la version récursive, si on veut déterminer les numérotations préfixe et posfixe, on réécrit DFS(u) de la façon suivante:

marquer le sommet u et lui attribuer son numéro préfixe
pour tout arc (u, v), traverser (u, v) et, si v n'a pas encore été marqué:
  poser $pere(v) = u$
  appeler DFS(v)
attribuer à u son numéro postfixe.

4.3 Applications des parcours en profondeur

Cette section monte comment utiliser le parcours en profondeur.

4.3.1 Tri topologique

Le *tri topologique* d'un graphe orienté acyclique $G = (X, A)$ consiste à ordonner linéairement tous ses sommets de sorte que si G contient un arc (u, v), u apparaisse avant v. (Si le graphe n'est pas acyclique, aucun ordre linéaire n'est pas possible). Le tri topologique peut être vue comme un alignement de ses sommets le long d'une ligne horizontale de manière que tous les arcs soient orientés de gauche à droite.

TRI-TOPOLOGIQUE (G)

1 utiliser le parcours en profondeur pour calculer les temps de fin de traitement $n(v)$ pour chaque sommet v

2 chaque fois que le traitement d'un sommet s'achève, l'insérer au début d'une liste chaînée

3 retourner la liste chaînée des sommets.

On peut effectuer un tri topologique en $O(|X| + |A|)$. On démontre la validité de cet algorithme à l'aide du caractérisation ci-dessous des graphes orientés acycliques.

Lemme 4.2. *Un graphe orienté G est acyclique si est seulement si un parcours en profondeur de G ne génère aucun arc retour.*

Preuve. $\Rightarrow$ On suppose qu'il existe un arc retour (u, v). Alors, le sommet v est un ancêtre du sommet u dans l'arbre en profondeur. Il existe donc un chemin de v à u dans G, et l'arc retour (u, v) clôt un circuit.

$\Leftarrow$ Supposons que G contienne un circuit C. On montre qu'un parcours en profondeur de G génère un arc retour. Soit v le premier sommet découvert dans C, et soit (u, v) l'arc précédent dans C. Nous affirmons que u et tous les autres sommets de C sont descendants de v dans l'arbre en profondeur. A la date $g(v)$, il existe un chemin blanc de v à u. D'après le Lemme 4.1., le sommet u devient un descendant de v dans l'arbre en profondeur, donc (u, v) est un arc retour.

Proposition 4.1. Tri-Topologique*(G) effectue le tri topologique d'un graphe orienté acyclique G.*

Preuve. Il suffit de montrer que pour une paire quelconque de sommets u, v, s'il existe un arc de u à v, alors $n(v) < n(u)$. Lorsque un arc (u, v) est exploré par le parcours en profondeur, v ne peut pas être gris, car alors v serait un ancêtre de u, et (u, v) serait un arc de retour, en contradiction avec le Lemme 4.2. Donc v est soit blanc soit noir. Si v est blanc, il devient un descendant de u, donc $n(v) < n(u)$. Si v est noir, $n(v) < n(u)$ également. Donc, pour un arc (u, v) quelconque du graphe orienté acyclique, on a $n(v) < n(u)$, ce qui démontre la proposition.

4.3.2 Composantes fortement connexes

Étant donné un graphe orienté $G = (X, A)$, on définit une relation d'équivalence R sur S par xRy ssi il existe un chemin de x à y et il existe un chemin de y à x. Les classes d'équivalence de x pour cette relation s'appellent les *composantes fortement connexes* (CFC) de G. Si le graphe n'admet qu'une seule composante connexe, on dit qu'il est *fortement connexe.*

Composantes-Fortement-Connexes(G)

Phase 1. Choisir un sommet initial, lancer DFS à partir de ce sommet et numéroter les sommets en ordre postfixe. Si il reste des sommets non numérotés, relancer DFS à partir de tels sommets jusqu'à ce que tous les sommets soient numérotés: on obtient ainsi une forêt d'arborescences. Les sommets de la première arborescence sont numérotés de 1 à p, ceux de la suivante de $p + 1$ à q et ainsi de suite.

Phase 2. Inverser l'orientation de tous les arcs. Le nouveau graphe obtenu est noté G^-. Relancer DFS sur G^-, en commençant par le sommet de plus grand numéro par rapport à la numérotation postfixe précédente. Lorsque la recherche s'arrête, les sommets marqués constituent une composante fortement connexe de G. S'il reste des sommets non numérotés, relancer DFS à partir du sommet non marqué de plus grand numéro: les sommets marqués lors de ce deuxième parcours constituent à nouveau une composante fortement connexe et ainsi de suite.

Proposition 4.2. Composantes-Fortement-Connexes*(G) permet de déterminer les composantes fortement connexes d'un graphe orienté G.*

Preuve. Dire que x et y appartiennent à la même CFC de G est équivalent à dire que dans G et G^- il existe un chemin de x à y et un autre de y à x. Supposons que x et y appartiennent à une même CFC de G et que dans la phase 2 x a été marqué avant y. Supposons que x appartienne à la $i$me arborescence obtenue dans la phase 2 et y appartienne à la $j$me arborescence. Comme $g(x) < g(y)$, $i \leq j$. Mais comme il existe un chemin de x à y, $i \geq j$, donc $i = j$.

Pour prouver la réciproque, soit x un sommet quelconque et notons r la racine de l'arborescence A_i^- en profondeur de G^- a laquelle appartient x dans la phase 2. Montrons que x et r appartiennent à la même CFC de G; ce résultat entraînera que deux sommets quelconques de A_i^- appartiennent aussi à une même CFC. Soit $x \neq r$. Dans G^- il existe un chemin de r à x, donc dans G il existe aussi un chemin $x_0 := x, x_1, \ldots, x_p = r$ de x à r. Il reste à montrer qu'il existe aussi dans G un chemin de r à x. Par le choix de r en tant que racine de A_i^-, le numero postfixe de r est supérieur à ceux des sommets x_j $(0 \leq j < p)$.

Supposons que r a été marqué dans la phase 1 avant x. D'après la remarque ci-dessus le numéro postfixe de r est supérieur à celui de x, donc le parcours DFS(r) dans G a commencé avant le parcours DFS(x) mais en revanche il s'est terminé après. Cela indique que x est descendant de r dans cette arborescence et il y a dans G un chemin de r à x.

Supposons maintenant que x a été marque dans la phase 1 avant r. Dans ce cas tous les sommets $x = x_0, x_1, \ldots, x_p = r$ appartiennent à la même arborescence en profondeur de G. Soit $j (0 \leq j < p)$ un indice tel que x_j ait été marqué avant r et $x_{j+1} = r$ ou bien x_{j+1} est marqué après r. Quand DFS(x_j) est lancé dans G ni r ni x_{j+1} ne sont pas marqués. Lorsque DFS(x_j) se termine, x_{j+1}, étant un successeur de x_j a nécessairement été marqué, ce que entraîne que r a aussi été marqué durant ce parcours. Donc r est un descendent de x_j dans l'arborescence construite durant DFS(x_j), ce qui entraîne que le numéro postfixe de x_j est strictement supérieur à celui de r, une contradiction.

4.3.3 Détermination des sommets d'articulations et des composantes 2-connexes d'un graphe non orienté

On appele *sommet d'articulation* d'un graphe non orienté $G = (X, E)$ un sommet dont la suppression augmente le nombre de composantes connexes du graphe. Un graphe est dit *2-connexe* ssi soit il est une arête soit il n'a pas de sommet d'articulation. On appelle *composante 2-connexe* tout sous-graphe 2-connexe maximal pour inclusion.

L'algorithme de détermination des sommets d'articulations est basé sur le critére suivant:

Proposition 4.3. *Soit $G = (X, E)$ un graphe connexe non orienté et soit T un arbre en profondeur de G. Un sommet s est un sommet d'articulation de G ssi:*

–ou bien s est la racine de T est s possède au moins deux fils dans T;

–ou bien s n'est pas la racine de T et, pour un fils f de s dans A, il n'y a pas d'arc retour entre n'importe quel descendant de f (y compris f) et un ancêtre propre de s.

Nous avons besoin, pour déterminer les sommets d'articulation, de deux numérotations. L'une d'elles est la numérotation préfixe. L'autre, appelée *numérotation basse*, est définie comme suit: *basse*(v) est la plus petite valeur préfixe d'un sommet dans un chemin que commence à v et qui soit la concaténation d'un chemin d'origine v dans l'arbre en profondeur et d'un arc retour. On peut remarquer qu'on a alors la relation:

$$basse(v) = \min \{prefixe(v), prefixe(z), basse(u)\}$$

pour tout z tel que (v, z) est un arc retour et tout fils u de v. Pour déterminer $basse(v)$ pour tout sommet v, nous considérons les sommets en ordre postfixe. Alors, nous pouvons caractériser les sommets d'articulation de façon suivant:

—le point de départ r de DFS est sommet d'articulation ssi il a au moins deux fils dans l'arbre;

—un sommet $s \neq r$ est un sommet d'articulation ssi il a au moins un fils f dans l'arbre tel que $basse(f) \geq prefixe(s)$.

5 Problèmes de plus courts chemins

Soit $G = (X, U)$ un graph orienté (ou non orienté) que nous supposons valué, c'est-à-dire qu'on a défini une application $p : U \to \mathbb{R}$, appelée *poids* ou *longueur* ou *coût*. On appelle *chemin* une suite $x_1, e_1, x_2, \dots e_n, x_{n+1}$ de sommets x_i et d'arcs e_j, telle que $e_j = (x_j, x_{j+1})$. On appelle poids (longueur) du chemin la somme des poids des arcs du chemin. La longueur d'un plus court chemin entre deux sommets x, y s'appelle *distance* $d(x, y)$ de x à y. On peut considérer les problèmes suivants:

- Étant donnés deux sommets x et y trouver un plus court chemin de x à y.
- Étant donné un sommet r trouver un plus court chemin de r à tous les sommets du graphe.
- Trouver la matrice des distance d'un graphe et trouver, pur toute paire de sommets x et y, un plus court chemin de x a y.

Les trois problèmes peuvent bien ne pas avoir de solution finie, si le graphe contient un circuit de longueur negatif. Il se trouve qu'il existe deux cas, fort importantes en pratique, où l'on est sûr que le graphe ne contient pas de circuit negatif:

1 lorsque tous les poids sont positifs ou nuls et on dispose de l'algorithme de Dijkstra;

2 lorsque le graphe est sans circuit (acyclique), et on dispose de l'algorithme de Bellman.

Dans le cas général (et pour le problèm 3), nous ferons appel à l'algorithme de Floyd-Warshall qui devra soit donner la solution s'il y en a une, soit mettre en évidence un circuit negatif s'il en existe.

On appelle *arborescence* de racine r un graphe orienté tel que le graphe non orienté sous-jacent est un arbre et pour tout sommet x, l'unique chaîne entre r est x dans l'arbre correspond à un chemin de r vers x dans l'arborescence.

5.1 Algorithme de Dijkstra

Soit $G = (X, U)$ un graphe orienté (ou non orienté) où tous les arcs ont des poids positifs ou nuls. On étend une arborescence initialement réduite à r, en gagnant à chaque étape un sommet non encore couvert et un arc dont ce sommet est l'extrémité, l'origine étant déjà dans l'arborescence. L'ensemble des sommets couverts par l'arborescence à une étape donnée est noté A. À la fin de l'algorithme, cette arborescence donne, pour chaque sommet x du graphe, un plus court chemin de r à x et est appelée *arborescence des plus courts chemins.*

On définit deux fonctions sur X : une fonction $\pi : X \to \mathbb{R}$ et une fonction $pere : X \to X$: à une iteration, $\pi(x)$ donne la longueur d'un plus court chemin de r à x n'utilisant comme sommets que des sommets déjà dans A et $pere(x)$ désigne le prédécesseur de x sur un tel chemin.

DIJKSTRA(G)

- 1 $A \leftarrow \{r\}, pivot \leftarrow r, \pi(r) \leftarrow 0$
- 2 pour tout sommet $x \neq r$ faire $\pi(x) \leftarrow +\infty$
- 3 pour j variant de 1 à $n - 1$ faire
- 3.1 pour tout sommet $y \notin A$ et successeur de $pivot$ faire

  si $\pi(pivot) + p(pivot) < \pi(y)$, alors

  $\pi(y) \leftarrow \pi(pivot) + p(pivot, y)$

  $pere(y) \leftarrow pivot$

- 3.2 chercher parmi les sommets non dans A un sommet y tel que $\pi(y)$ soit minimum
- 3.3 $pivot \leftarrow y$
- 3.4 $A \leftarrow A \cup \{pivot\}$.

À chaque étape, l'actualisation de la fonction π implique $O(d^+(pivot))$ calculs, donc au total $O(\sum_{x\in X} d^+(x)) = O(m)$ operations. D'autre part, à chaque étape, on doit déterminer le minimum d'un ensemble de $n-|A|$ valeurs, ceci pouvant ce faire à l'aide de $O(\sum_{i=1}^{n-1}(n-i)) = O(n^2)$ operations. L'algorithme de Dijkstra a donc une complexité en $O(n^2)$.

Proposition 5.1. *L'algorithme de Dijkstra est valid.*

Preuve. Nous montrons que $\pi(pivot)$ *est la longueur d'un plus court chemin de* r *à* $pivot$ *et que* $pere(pivot)$ *est le prédécesseur de pivot dans se plus court chemin.*

Supposons q'il existe un chemin L de longueur $< \pi(pivot)$ de r à $pivot$. Le chemin L doit sortir de l'arborescence A, donc L contient deux sommets adjacents z, y telles que $z \in L \setminus A, z \neq pivot$, $y \in L \cap A$ et $(y,z) \in U$. Comme tous les poids sont positifs, on obtient

$$\pi(z) < d(r,z) \leq d(r,pivot) < \pi(pivot),$$

une contradiction avec le choix du sommet-pivot.

5.2 Algorithme de Bellman et la méthode PERT

Soit $G = (X,U)$ un graphe orienté sans circuit (mais avec des poids arbitraires). Soit $1,\ldots,n$ la numérotation des sommets de G en utilisant le tri topologique. On peut supposer que r est le sommet numero 1 (en cas contraire, on peut supprimer toutes les sommes ayant un numero plus petit que r). Soit $de-num(i)$ le sommet dont son numero est i. On opère comme dans l'algorithme de Dijkstra en $n-1$ etapes, en grossissant, à chaque étape, d'un sommet et d'un arc une arborescence A.

Bellman(G)

- 1 $A \leftarrow \{r\}, \pi(r) \leftarrow 0$
- 2 pour tout sommet $x \neq r$ faire $\pi(x) \leftarrow +\infty$
- 3 pour j variant de 2 à n faire

  $y \leftarrow de - num(j)$

  $\pi(y) \leftarrow min_{x\in A,(x,y)\in U}[\pi(x) + p(x,y)]$

  si le minimum ci-dessus est réalisé grâce au sommet x, alors $pere(y) \leftarrow x$.

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

On retrouve une application intéressante de cet algorithme quand on détermine les chemins critiques dans l'analyse d'un *diagramme de PERT* (Program Evaluation and Review Technique). Supposons que pour réaliser un travail complexe on doive exécuter un certain nombre de tâches, dont les périodes d'exécution ne sont pas indépendants. On peut effectuer les tâches en parallèle, mais certaines d'entre elles doivent être achevées avant que d'autre puissent commencer. Le problème est alors de *savoir quand on peut et duand on doit commencer les tâches pour achever l'ensemble du travail au plus tôt.*

On peut modéliser ce problème en utilisant un graphe orienté. Les sommets de ce graphe représentent les tâches. Un arc de a vers b indique que la tâche a doit être achevée avant que la tâche b commence. On porte sur l'arc (a, b) la durée de la tâche a. Enfin deux sommets *début* et *fin* modélisent le début et la fin du chantier. Il y a un arc entre *début* et toute tâche de durée nulle, demême qu'il y en a un de toute tâche vers *fin*, portant la durée de cette tache. Étant donné un chemin de longueur maximum de *début* à *fin*, comme toutes les tâches qui figurent sur ce chemin doivent être achevées pour que le travail soit terminé, la durée minimum du chantier ne peut être inférieure à la longueur de ce chemin. Réciproquement, si on commence toute tâche dès que possible, la durée totale ne sera supérieure à cette longueur. Donc nous devons chercher un chemin de longueur maximum dans un graphe sans circuit (l'existence d'un circuit implique que le travail est irréalisable). Résoudre ce problème revient à résoudre le problème de la recherche d'un chemin de longueur minimum dans le graphe obtenu en remplaçant toutes les longueurs par leurs opposées.

5.3 Plus courts chemins de tout sommet à tout sommet: l'algorithme de Floyd-Warshall

Soit $X = \{x_1, \ldots, x_n\}$ l'ensemble de sommets de G et soit $P = (p_{ij})$ la matrice definit de la façon suivante: $p_{ij} = p(x_i, x_j)$ si $(x_i, x_j) \in U$, $p_{ii} = 0$ et $p_{ij} = \infty$ si $i \neq j$ et $(x_i, x_j) \notin U$. Soit $d_{ij}^{(k)}$ le poids minimum d'un chemin d'au plis k arcs du sommet x_i au sommet x_j. Pour $k = 0$ il existe un plus court chemin sans arc de x_i vers x_j ssi $i = j$. Donc, $d_{ij}^{(0)} = p_{i,j}$ pour tous i et j. Pour $k \geq 1$, on calcule $d_{ij}^{(k)}$ comme étant le minimum entre $d_{ij}^{(k-1)}$ et le poids minimal d'un chemin d'au plus k arcs de x_i à x_j en passant par x_k. On définit donc récursivement

$$d_{ij}^{k} = min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$$

si $k \geq 1$. La matrice $D^{(n)} = (d_{ij}^{(n)})$ donne le résultat.

FLOYD-WARSHALL(G)

- 1 $D^{(0)} \leftarrow P$
- 2 pour $k \leftarrow 1$ à n faire
- 3 pour $i \leftarrow 1$ à n faire
- 4 pour $j \leftarrow 1$ à n faire
- 5 $d_{ij}^{(k)} \leftarrow min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$
- 6 retourner $D^{(n)}$.

Le temps d'exécution de l'algorithme de Floyd-Warshall est de $O(n^3)$.

Pour construire les plus courts chemins, pendant que l'algorithme de Floyd-Warshall calcule les matrices $D^{(k)}$ il est possible de calculer les matrices de liaison $\Pi^{(k)}$, où $\pi_{ij}^{(k)}$ est defini comme le prédécesseur du sommet x_j sur un plus court chemin partant du sommet x_i, et dont tous les sommets intermédiaires sont dans $\{1, \ldots, k\}$. On peut donner une formulation récursive de $\pi_{ij}^{(k)}$.

Pour $k = 0$ soit $\pi_{ij}^{(0)} = nil$ si $i = j$ ou $p_{ij} = \infty$ et $\pi_{ij}^{(0)} = i$ si $i \neq j$ et $p_{ij} < \infty$. Formellement, pour $k \geq 1$,

$$\pi_{ij}^{(k)} := \pi_{ij}^{(k-1)} \text{ si } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$$

et

$$\pi_{ij}^{(k)} := \pi_{kj}^{(k-1)} \text{ si } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}.$$

6 Flot maximum dans un réseau

La recherche d'un flot de valeur maximum dans un réseau a des applications directes, par exemple en termes de trafic dans un réseau de circulation ou d'un réseau téléinformatique mais il en a bien d'autres: problèmes d'affectation, problèmes de transport, ou encore la détermination de la connectivité d'un graphe, paramètre très lié à la résistance aux pannes dans un réseau.

Dans ce chapitre, on considère un *réseau* $N = (X, U, c, s, t)$, i.e., un graphe orienté $G = (X, U)$ pour lequel on a défini une *capacité* $c : U \to \mathbb{R}^+ \cup \{0\}$ des arcs, et choisi deux sommets privilégiés s, comme *source*, et t comme *puits* ou *destination*.

On appelle *flot* dans le réseau N une application $f : U \to \mathbb{R}^+ \cup \{0\}$ qui vérifie les propriétés suivantes:

(i) pour tout arc u, $f(u) \leq c(u)$,
(ii) pour tout sommet x, $x \neq s, t$, il y a conservation du flot en x, ce qui signifie que le flot total entrant en x est égal au flot sortant de x, i.e.,

$$\sum_{y \in \Gamma^-(x)} f(y, x) = \sum_{z \in \Gamma^+(x)} f(x, z) \quad \forall x \in X - \{s, t\}.$$

On définit la *valeur du flot* f, que on note $val(f)$ le flot total quittant s moins le flot total entrant en s (valeur à la source) où le flot total entrant en t moins le flot total quittant t (valeur au puits).

Soit S un ensemble de sommets contenant s et ne contenant pas t. On note $\overline{S} := X \setminus S$ le complémentaire de S dans X. Soit $(S, \overline{S})$ l'ensemble des arcs dont l'origine est dand S et l'extrémité dans $\overline{S}$. L'ensemble $(S, \overline{S})$ s'appelle une (s,t)-*coupe* (où une *coupe* séparant la source de destination). On définit la *capacité* d'une (s,t)-coupe $(S, \overline{S})$ par:

$$c(S, \overline{S}) = \sum_{u \in (S, \overline{S})} c(u).$$

Lemme 6.1. *Si f est un flot et $(S, \overline{S})$ une (s,t)-coupe, la quantité*

$$f(S, \overline{S}) - f(\overline{S}, S)$$

est indépendante du choix de S, et définit sans ambiguité la valeur du flot.

Preuve. Pour tout sommet x de S autre que s on a, d'après le principe de conservation du flot,

$$\sum_{u \text{d'origine } x} f(u) - \sum_{u \text{ d'extrémité } x} f(u) = 0.$$

En utilisant la definition de la valeur du flot calculée à la source, on a

$$\sum_{u \text{ d'origine } s} f(u) - \sum_{u \text{ d'extrémité } s} f(u) = val(f).$$

Faisant la somme de toutes ces égalites , on vois certains $f(u)$ apparaître deux fois: ce sont les arcs u dont les deux extrémités appartiennent à S et telles $f(u)$ apparaissent une fois avec le signe "+" et une fois avec "-". Les arcs de $(S, \overline{S})$ apparaissent une seule fois avec le signe "+" et ceux de $(\overline{S}, S)$ apparaissent une seule fois avec le signe "-". Le résultat de la sommation indiquée donne donc

$$f(S, \overline{S}) - f(\overline{S}, S) = val(f).$$

Pour tout arc u de cette coupe nous avons $f(u) \leq c(u)$, donc $f(S, \overline{S}) \leq c(S, \overline{S})$ pour tout flot f. Puisque $f(u) \geq 0$ pour tout arc u de $(\overline{S}, S)$, on deduit une partie du résultat suivant:

Proposition 6.1 (Ford-Fulkerson) *Soit f un flot et $(S,\overline{S})$ une (s,t)-coupe d'un réseau N. Alors*

$$val(f) \leq c(S,\overline{S}),$$

i.e., la valeur d'un flot est plus petite où egale à la valeur d'un coupe. Il existe un flot f^* (un flot de valeur maximum) et une (s,t)-coupe $(S^*,\overline{S^*})$ (une coupe de capacité minimum) dont

$$val(f^*) = c(S^*,\overline{S^*}).$$

Si les capacités sont entières, il existe un flot maximum f^* tel que tous les $f^*(u), u \in U$ soient entiers.

La preuve de deuxième affirmation est algorithmique et sera prouvé ensuite.

6.1 Algorithme de Ford et Fulkerson

L'algorithme de Ford et Fulkerson détermine un flot de valeur maximum ainsi qu'une coupe de capacité minimum. Il repose sur le principe suivant: partant d'un flot f dans le réseau (par exemple le flot nul), on cherche s'il existe une "chaîne augmentante" pour ce flot. S'il n'en existe pas, le flot f est maximum; s'il en existe une, on augmente le flot le long des arcs de cette chaîne et on recommence avec le nouveau flot obtenu.

Supposons établi un flot f. Une chaîne $P(f)$ de s à t est dite *augmentante* pour le flot f s'il existe un $\delta > 0$ tel que l'on a $c(u) - f(u) \geq \alpha$ pour les arcs u de cette chaîne à l'endroit (c'est-à-dire les arcs dont on rencontre l'origine avant l'extrémité lorsqu'on parcourt $P(f)$ de s vers t) et $f(u) \geq \delta$ pour les arcs de $P(f)$ à l'envers. En désignant par $P^+(f)$ les arcs à l'endroit et par $P^-(f)$ les arcs à l'envers de $P(f)$, on prendra alors

$$\delta := \min \{ \min{}_{u \in P^+(f)}[c(u) - f(u)], \min{}_{u \in P^-(f)} f(u)\}.$$

Ayant ainsi déterminé une chaîne augmentante $P(f)$ pour le flot f et la valeur de δ correspondante, on augmente $f(u)$ de δ pour les arcs $u \in P^+(f)$ et on le diminue de δ pour les arcs de $P^-(f)$: la valeur du flot a augmenté de δ.

Lemme 6.2. *Un flot f est un flot de valeur maximum ssi il n'existe pas de chaîne augmentante de s à t.*

Preuve. Supposons qu'il n'existe pas de chaîne augmentante de s à t. Appelons S^* l'ensemble des sommets x pour lesquels il existe un chaîne augmentante de s à x. Le sommet s est dans S et, par hypothèse, t n'est pas dans S^*. Donc $(S^*,\overline{S^*})$ est une (s,t)-coupe.

Soit (x,y) un arc de G dont $x \in S^*$ et $y \in \overline{S^*}$. Il existe une chaîne augmentante de s à x mais cette chaîne ne peut pas être prolongée en une chaîne augmentante de s à y, donc $f(x,y) = c(x,y)$ et $f(y,x) = 0$. Donc f est un flot maximum, d'après la première affirmation de Proposition 6.1.

Aussi $val(f) = c(S,\overline{S^*})$ donc $(S^*,\overline{S^*})$ est une coupe minimum, est la deuxième affirmation de Proposition 6.1 est établi.

Chaque itération de l'algorithme de Ford et Fulkerson est constituée de deux phases. La première est constituée d'un algorithme de marquage destiné à trouver une chaîne augmentante s'il en existe une. La seconde exploite la chaîne augmentante pour améliorer le flot courant. On appelle "sommet marqué" un sommet tel que on a trouvé une chaîne augmentante de s à x pour le flot courrant f. Examiner un sommet marqué x c'est regarder si l'on peut prolonger la chaîne augmentante de s à x jusqu'à un nouveau sommet y voisin de x non encore marqué. Pour cela, on attribue à chaque sommet marqué y une marque à trois places. Dans la première place dans la marque de y nous gardons le nom du sommet x qui nous a permis de marquer y. La seconde place nous conservons un signe "+" où "-". Le signe "+" indique que (x,y) est un arc à l'endroit par rapport à la chaîne augmentante; le signe "-" indique que, lorsque y à été marque l'arc utilisé était (y, x), c'est-à-dire un arc à l'envers par rapport à la chaîne augmentante. Le troisième renseignement $\delta(y)$ indique de combien on peut espérer augmenter le flot à l'aide de cette chaîne, jusqu'à y: $\delta(y)$ est le minimum de $\delta(x)$ et de $c(x, y) - f(x, y)$ si le signe de y est "+" , ou bien, si le signe de y est "-", le minimum de $f(y, x)$ et de $\delta(x)$.

Algorithme de marquage(N, f)

1. **Initialisation:**

   marquer s par $(s, +, \infty)$

   considérer tout sommet $x \neq s$ comme non marqué

   considérer qu'aucun sommet n'est examiné

2. tant que t est non marqué et qu'il reste un sommet marqué non examiné, faire

2.1. soit x un sommet marqué non examiné et soit δ la valeur absolue du second paramètre de la marque de x

2.2. pour tout successeur y de que n'est pas marqué, faire

  si $c(x, y) > f(x, y)$, alors

   $\delta(y) \leftarrow$ min $\{\delta(x), c(x, y) - f(x, y)\}$

   marquer y par $(x, +, \delta(x))$

2.3. pour tout prédécesseur z de x qui n'est pas marqué, faire

  si $f(z, x) > 0$, alors

   $\delta(z) \leftarrow$ min $\{\delta(x), f(z, x)\}$

   marquer z par $(x, -, \delta(z))$

2.4. considérer x comme examiné

Algorithme Ford-Fulkerson(N)

1. Définir un flot f, éventuellement le flot nul

2. répéter

2.1. appliquer l'algorithme de marquage

2.2. si t est marquer, alors

  soit P la chaîne augmentante de s à t

  soit $\delta \leftarrow \delta(t)$

  pour tout arc (x, y) de P parcouru à l'endroit, faire

   $f(x, y) \leftarrow f(x, y) + \delta$

  pour tout arc (x, y) de P parcouru à l'envers, faire

   $f(x, y) \leftarrow f(x, y) - \delta$

3. jusqu'à ce que t ne soit plus marqué.

À la sortie de cette boucle, le flot est de valeur maximum. On peut exiber des exemples des réseaux avec des capacités éelles où l'algorithme de Ford et Fulkerson ne converge pas en temps fini. Dès que capacités sont entièreres, le flot maximum est obtenu en un nombre fini d'itérations. Pour certains réseaux ce nombre est borné par la valeur de flot (donc des capacités des arcs), donc en géneral, l'algorithme de Ford et Fulkerson n'est pas un algorithme polynomial. Nous allons présenter l'algoritme de Dinic pour le problème de flot maximum dont sa complexité s'exprime en fonction des la dimension (n et m) du graphe.

6.2 Algorithme de Dinic-Edmonds-Karp (DEK)

Soit $N = (G, c, s, t)$ un réseau avec un graphe G antisymétrique et soit f un flot de N. À partir de N et f nous allons créer un réseau auxiliaire $N(f) = (X, U(f), ac, s, t)$ appelé *réseau d'écart*, où la notion de chaîne augmentante pour f est remplacée par la notion plus simple de chemin augmentant pour le flot nul. La construction de $N(f)$ est la suivante:

–à tout arc (x, y) non saturé de N (i.e. $f(x, y) < c(x, y)$), on associe un arc (x, y) de capacité $ac(x, y) = c(x, y) - f(x, y)$

–à tout arc (x, y) de N portant un flux non nul, on associe un arc (y, x) de capacité $ac(y, x) = f(x, y)$.

Un chemin dans $N(f)$ est trivialement en bijection avec un chaîne augmentante pour le flot dans N. L'idée de l'algoritme de Dinic-Edmonds-Karp est d'utiliser comme chaînes augmentantes seulement des (s, t)-chemins de $N(f)$ avec un nombre minimum d'arcs (donc de plus courts chemins).

L'algorithme DEK procède par étapes. Pour définir une étape donnée appelons d la distance (en nombre d'arcs) de s à t dans $N(f)$ au début de cette étape. L'algorithme va chercher des chaînes augmentantes de s à t de longueur d pour les utiliser pour augmenter le flot, jusqu'à ce qu'il n'existe plus de telles chaînes. Pour cela, on construit à partir de $N(f)$ ce qu'on appele le *réseau de couches* $AN(f)$:

- $L_0 = \{s\}$ et $L_d = \{t\}$
- pour $i = 1, \ldots, d-1$ soit L_i l'ensemble des sommets de $N(f)$ que sont à distance (en nombre d'arcs) i de s.

Les arcs du réseau de couches sont les arcs de $N(f)$ ayant leur origine dans L_i et leur extrémité dans L_{i+1} pour $i = 0, \ldots d-1$. Pour construire le réseau $AN(f)$ on utilise le parcours en largeur de $N(f)$ en mémorisant seulement les arcs vers les nouveaux sommets et les sommets au distance $< d$ de s.

Dans un réseau de couches donné, on répète les opérations suivantes:

1. Chercher un chemin de s à t dans le réseau $AN(f)$. On procède pour cela par un parcours à partir de s. Pendant cette recherche, lorsque, utilisant un arc (x, y), on atteint un sommet y de demi-degré extérieur nul, on supprime l'arc (x, y) du $AN(f)$.

2. Si on trouve un chemin de s à t, on augmente le flot dans G en utilisant la chaîne augmentante correspondante. On actualise le réseau de couches en supprimant les arcs (x, y) du réseau de couches provenant d'un arc (x, y) de N qui devient saturé et les arcs (y, x) du réseau de couches provenant d'un arc (x, y) de N sur lequel le flux est devenu nul.

3. Si on ne trouve pas de tel chemin, l'étape est terminée.

On passe alors à l'étape suivante après avoir mis à jour le réseau d'écart $N(f)$ et le réseau de couches $AN(f)$ correspondant. On peut montrer que d'une étape à l'autre de l'algorithme de Dinic, la valeur de la distance d entre la source s et la destination t croît strictement. L'algorithme est terminé lorsqu'il n'existe plus de chemin de s à t dans le réseau d'écart.

Considérons une étape de l'algorithme DEK; entre deux suppresions d'arcs, le nombre d'arcs traversés par les parcours successifs vaut au plus $d \leq n$. Le graphe de couches ayant au départ au plus m arcs, on déduit qu'une étape a une complexité en $O(nm)$. Le nombre d'étapes ne peut pas être supérieur à n, puisque la distance d est croissante d'une étape à l'autre et ne peut dépasser n. L'algorithme DEK a donc une complexité de $O(n^2m)$ (une autre version de cet algorithme est en $O(n^3)$.

6.3 Flot maximum et la programmation linéaire

Le problème du flot maximum est un problème de programmation linéaire où les variables sont les flux circulant sur les arcs du réseau. Le problème de la coupe de capacité minimum est aussi un problème de programmation lineaire. Pour montrer que les deux problèmes sont duaux l'un de l'autre on procède de la façon suivante.

Soit $N = (G = (X, U), c, s, t)$ le réseau et soit A la matrice d'incidence du graphe G. Un flot f de valeur v est défini par les contraintes

$$Af = \begin{cases} +v & \text{en ligne } s \\ -v & \text{en ligne} t \\ 0 & \text{le reste} \end{cases}$$

$$f \leq c$$

$$f \geq 0$$

Le problème est de maximiser v avec ces contraintes. Pour réécrire le problème, on va utiliser le vecteur $g \in \mathbb{R}^n$ définit par

$$g(x) = \begin{cases} -1 & \text{si } x = s \\ +1 & \text{si } x = t \\ 0 & \text{sinon.} \end{cases}$$

Le programme linéaire devient

$$\max v$$

$$Af + gv = 0$$

$$f \leq c$$

$$f \geq 0.$$

Pour donner la formulation du problème dual, on introduira les variables $\pi(x)$ pour les premières n contraintes de conservation de flot et les variables $\gamma(x, y)$ pour les contraintes de capacité. On obtient donc

$$\min \sum_{(x,y)\in U} \gamma(x, y)c(x, y)$$

$$\pi(x) - \pi(y) + \gamma(x, y) \geq 0 \text{ pour chaque arc } (x, y)$$

$$-\pi(s) + \pi(t) \geq 1$$

$$\pi(x) \neq 0$$

$$\gamma(x, y) \geq 0.$$

Lemme 6.3. *Chaque (s,t)-coupe $(S,\overline{S})$ définit une solution réalisable du problème dual de la façon suivante:*

$$\gamma(x,y) = \begin{cases} 1 & si\ x \in S\ \ et\ y \in \overline{S} \\ 0 & sinon, \end{cases}$$

$$\pi(x) = \begin{cases} 0 & si\ x \in S \\ 1 & si\ x \in \overline{S}. \end{cases}$$

Preuve. Il y 4 cases simples à considérer (1) $x, y \in S$ (2) $x, y \in \overline{S}$ (3) $x \in S$ et $y \in \overline{S}$ (4) $x \in \overline{S}$ et $y \in S$. L'inégalité est stricte seulement si $x \in \overline{S}$ et $y \in S$. D'autre part, $\pi(s) = 0$ et $\pi(t) = 1$ puisque $s \in S$ et $t \in \overline{S}$. Donc $-\pi(s) + \pi(t) = 1$ qui vérifie la deuxième inégalité. Finallement, le coût de cette solution duale est de

$$\sum_{(x,y)\in U} \gamma(x,y)c(x,y) = \sum\{c(x,y) : (x,y) \in U, x \in S, y \in \overline{S}\} = c(S,\overline{S}).$$

La théorème de dualité de la programmation linéaire et la Lemme 6.3 nous fournissent une autre preuve du théorème de Ford et Fulkerson (Proposition 6.1).

6.4 Flot de coût minimum et l'algorithme de Busacker et Gowen

Le but d'un problème de flot de coût minimum est de déterminer, dans un réseau où les arcs sont valués non seulement par une fonction capacité c, mais aussi une fonction de coût q, un flot d'une valeur donné v et de coût minimum parmi les flots de valeur v (le coût d'un flot f étant la somme, étendue à tous les arcs u du réseau, des produits $f(u)q(u)$. La formulation sous forme de programme linéaire est la suivante:

$$\min \sum_{u\in U} q(u)f(u)$$

$$Af = \begin{cases} +v & \text{en ligne } s \\ -v & \text{en ligne} t \\ 0 & \text{le reste} \end{cases}$$

$$f \leq c$$

$$f \geq 0$$

L'algorithme de Busacker et Gowen détermine un flot de coût minimum de la même façon que l'algoritmes de Ford et Fulkerson où celui de Dinic. On augmente à chaque phase le flot dans le réseau en utilisant un chaîne augmentante de coût minimum de s à t dans le réseau d'écart, le coût d'une chaîne étant obtenu en additionnant les coûts des arcs qui sont dans le sens direct puis en soustrayant les coûts des arcs inverses. Pour cela on utilise le réseau d'écart $N(f)$ que l'on value de la façon suivante:

–un arc (x,y) du $N(f)$ issu d'un arc (x,y) non saturé du réseau N est valué par le coût $q(x,y)$ que possède cet arc;
–un arc (y,x) du $N(f)$ issu d'un arc (x,y) du réseau N portant un flux non nul est valué par l'opposé $-q(x,y)$ du coût que possède cet arc.
Notons R le graphe valué ainsi obtenu. Rechercher dans le réseau d'origine une chaîne augmentante de coût minimum revient exactement à rechercher un plus court chemin de s à t dans R. On détermine dans R un plus court chemin de s à t (pour cela on peut utiliser l'algorithme de Floyd-Warshall) et on augmente le flot dans le réseau d'origine en utilisant la chaîne augmentante correspondant au chemin que l'on vient de découvrir.

7 Applications de la théorie des flots

7.1 Forte arc-connectivité d'un graphe orienté

Nous allons commencer par l'étude de la forte arc-connectivité d'un graphe orienté $G = (X, U)$. Soient a et b deux sommets de G. On considère le réseau $R_{ab} = (G, c \equiv 1, a, b)$ avec une capacité égale à 1 sur tous les arcs de G. Soit $P(a, b)$ le nombre maximum de chemins d'origine a et d'extrémité b, deux à deux arc-disjoints et soit $N(a, b)$ représente le nombre minimum d'arcs qu'il faut supprimer pour qu'il n'existe plus de chemin de a vers b. Il est trivial que $P(a, b) \leq N(a, b)$.

Proposition 7.1. *Si f_{ab} est un flot de valeur maximum $val(f_{ab})$ dans le réseau R_{ab}, alors $val(f_{ab}) = P(a, b)$. De plus, pour toute paire de sommets distincts a et b on a $P(a, b) = N(a, b)$*(théorème de Menger).

Preuve. Considérons un flot maximum à valeurs entières (puisque les capacités sont entièrs). Les arcs de R_{ab} qui portent un flux égal à 1 forment des chemins de a vers b deux à deux arc-disjoints, donc $val(f_{ab}) \leq P(a, b)$.

Considérons maintenant une coupe minimum $(S, \overline{S})$. D'après la Proposition 6.1, cette coupe est de capacité $val(f_{ab})$, donc elle contient autant d'arcs. La destruction des arcs de cette coupe détruit tout chemin de a à b, donc $N(a, b) \leq val(f_{ab}) \leq P(a, b)$. Mais comme l'iégalité $P(a, b) \leq N(a, b)$ a été déjà constatée, on obtient $P(a, b) = N(a, b)$.

Pour déterminer la forte arc-connectivité de G, if faut choisir le minimum de $N(a, b)$ pris pour tous les $n(n-1)$ couples de sommets a, b. Grâce au lemme suivant, il suffit d'envisager n couples.

Lemme 7.1. (lemme de Zorn) *Supposons définie une numérotation des sommets de G de 0 à $n-1$. La forte arc-connectivité de G est le minimum de $N(x_i, x_{i+1})$ quand i varie de 0 à $n-1$, en posant $x_n = x_0$.*

Preuve. Par réduction à l'absurde. Supposons que le minimum $N(x_i, x_{i+1})$, $i = 1, \ldots, n-1$, soit strictement plus grand que la forte arc-connexité k du G, c'est-à-dire aussi plus grande que le cardinal minimum d'une coupe déconnectante. En détruisant les k arcs d'une coupe qu séparе un certain sommet $a =: x_j$ d'un certain sommet $b := x_l$ on ne sépare donc aucun sommet x_i du sommet x_{i+1} pour $i = 0, \ldots, n-1$. Donc il existe un chemin de $a = x_j$ à x_{j+1}, un chemin de x_{j+1} à $x_{j+2}, \ldots$, un chemin de x_{k-1} à $x_k = b$, et cela après la suppression des k arcs de la coupe. En concaténant ces chemins, on obtient un chemin de a à b, une contradiction.

7.2 Détermination de l'arête connectivité d'un graphe non orienté

Soit $G = (X, E)$ un graphe non oriente. G est dit *k-arête-connexe* si la suppression d'au plus $k-1$ arêtes quelconques résulte en un graphe connexe. On appelle *arête-connectivité* de G le plus grand entier k tel que G soit k-arête-connexe.

Soit $G^* = (X, E^*)$ le *symétrisé* de G, c'est-à-dire chaque arête uv de G donne naissance dans G^* aux deux arcs (u, v) et (v, u).

Étant donnés deux sommets a et b, on considère les quatre paramètres suivants:

- $N(a, b)$ est le nombre minimum d'arêtes à supprimer dans G pour qu'il n'existe plus de chaîne entre a et b;

- $N^*(a, b)$ est le nombre minimum d'arcs à supprimer dans G^* pour qu'il n'existe plus de chemin de a et b dans G^*;

- $P(a, b)$ est le nombre maximum de chaînes deux à deux arête-disjointes de G entre a et b;

- $P^*(a,b)$ est le nombre maximum de chemins deux à deux arc-disjoints de G^* de a vers b.

Proposition 7.2. *Soit $val(f_{ab})$ la valeur maximum d'un flot de a à b dans G^*. Alors*

$$N(a,b) = N^*(a,b) = P(a,b) = P^*(a,b) = val(f_{ab}).$$

Preuve. (i) On a de façon évidente $N(a,b) \geq P(a,b)$.

(ii) Montrons $P(a,b) \geq P^*(a,b)$.

Si les $P^*(a,b)$ chemins de a vers b n'utilisent pas, l'un, un arc (u,v) et l'autre, l'arc (v,u) on peut oublier les orientations pour fabriquer $P^*(a,b)$ chaînes entre a et b. Supposons donc qu'un chemin P_1 et un chemin P_2 partagent une arête uv en utilisant l'un (u,v) et l'autre (v,u). Soit P_1' la partie de P_1 entre a et u et P_1'' la partie de P_1 entre v et b. De façon analogue, soit P_2' la partie de P_2 entre a et v et P_2'' la partie de P_2 entre u et b. On transforme P_1 et P_2 en $P_1' \cup P_2''$ et $P_2' \cup P_1''$. On voit que l'on partage ainsi exactement une arête de moins. Ceci permet de transformer les $P^*(a,b)$ chemins de a vers b, en autant des chaînes deux à deux arête-disjointes.

(iii) Montrons $N^*(a,b) \geq N(a,b)$.

Supprimons de G les arêts qui donnent naissance à $N^*(a,b)$ arcs d'une (a,b)-coupe dans G^*. Il y au plus $N^*(a,b)$ telles arêtes. L'existence d'une chaîne entre a et b dans le graphe obtenu impliquerait l'existence d'une chemin de a à b dans G^* moins les arcs de la coupe. D'où la relation.

Nous avons finalement les inégalités suivantes:

$$N^*(a,b) \geq N(a,b) \geq P(a,b) \geq P^*(a,b).$$

La proposition 7.1 implique qu'on a aussi

$$P^*(a,b) = N^*(a,b) = val(f_{ab}).$$

Toutes les quantités sont donc égales, ce qui achève la démonstration.

7.3 Forte connectivité d'un graphe orienté, connectivité d'un graphe non orienté

Un graphe G non orienté est dit *k-connexe* s'il a au moins $k+1$ sommets et si la suppression d'au plus $k-1$ sommets quelconque résulte en graphe connexe. On appelle *sommet-connectivité* de G le plus grand entier k tel que G soit k-connexe.

Considérons un graphe orienté. On construit alors un nouveau réseau NG de la façon suivante: à tout sommet x de G on associe un arc (x', x'') et à un arc (u,v) de G on associe l'arc (v'', u'). Tous les arcs de NG ont une capacité égale à 1. On prend comme source a'' et comme destination b'. On peut montrer que le nombre minimum de sommets à supprimer pour détruire tout chemin de a vers b dans G est égal à la valeur d'un flot maximum de a'' à b' dans NG.

Si il s'agit de calculer la connectivité d'un graphe non orienté G, on utilise le symétrisé de G pour se ramener au cas précédent. La proposition 6.1. permet d'établir deux autres théorèmes de Menger suivants:

Proposition 7.3. *Un graphe non orienté $G = (X,E)$ est k-connexe ssi entre deux sommets quelconques, il existe k chaînes n'ayant en commun que leurs extrémités.*

Proposition 7.4. *Un graphe orienté $G = (X,E)$ est k-fortement connexe ssi entre deux sommets quelconques, il existe k chemins de l'un vers l'autre n'ayant en commun que leurs extrémités.*

7.4 Couplage maximum dans un graphe biparti

Etant donné un graphe non orienté $G = (X, E)$, un *couplage* est un sous-ensemble d'arêtes M deux à deux sans extrémités commune. Un *couplage maximum* est un couplage avec un nombre maximum d'arêtes. Les flots permettent de trouver un couplage maximum dans un graphe biparti $G = (X_1, X_2; E)$. On définit le réseau $N(G)$ de la manière suivante. Soit la source s et le puits t deux nouveaux sommets n'appartenant pas à $X := X_1 \cup X_2$. Les arcs de $N(G)$ sont donnés par

$$\{(s,x) : x \in X_1\} \cup \{(x,y) : xy \in E \text{ et } x \in X_1, y \in X_2\} \cup \{(y,t) : y \in X_2\}.$$

Pour compléter la construction, on affecte une capacité unitaire à chaque arc de $N(G)$.

Proposition 7.5. *Soit $G = (X_1 \cup X_2; E)$ un graphe biparti et soit $N(G)$ son réseau correspondant. Si M est un couplage de G, il existe un flot entièrs f dans $N(G)$ avec $val(f) = |M|$. Inversement, si f est un flot entier dans $N(G)$, il existe un couplage M de G de cardinal $|M| = val(f)$.*

Preuve. Soit M un couplage de G. On définit f de la manière suivante: si $(x, y) \in M$ avec $x \in X_1$ et $y \in X_2$, alors $f(s, x) = f(x, y) = f(y, t) = 1$. Pour tous les autres arcs (u, v) on définit $f(u, v) = 0$. Il est evident que f est un flot de valeur $|M|$. Pour la réciproque, soit f un flot entièr de $N(G)$ (donc un flot avec des valeurs 0 et 1) et soit

$$M = \{xy : x \in X_1, y \in X_2 \text{ et } f(u,v) = 1\}.$$

Il est facile de se convaincre que M est un couplage de valeur $val(f)$.

7.5 Le problème de transport

Soit $R = (X, U, c, S, T)$ un réseau du transport, donc les sommets représentent des villes, et les arcs des routes à sens unique. Le coût du transport d'une unité de marchandise le long de l'arc (i, j) vaut $c_{ij} \geq 0$. On suppose qu'aux sommets-sources S il y des excédents de marchandise et qu'aux sommets-puits T il y a des demandes de cette marchandise. Il s'agit d'organiser le transport des excédents de façon à minimiser le coût de transport.

Une solution réalisable du problème de transport sera par les nombres x_{ij} d'unités transportées le long des arcs $(i, j) \in U$, donc elle doit vérifier les contraintes suivantes:

- pour toute source i, $\sum_{(i,j)\in U} x_{ij} - \sum_{(j,i)\in U} x_{ji} =$excédent en i
- pour tout puits i, $\sum_{(j,i)\in U} x_{ji} - \sum_{(i,j)\in U} x_{ij} =$demande en i
- pour tout sommet intermédiaire i, $\sum_{(i,j)\in U} x_{ij} - \sum_{(j,i)\in U} x_{ji} = 0$.

Soit $b \in \mathbb{R}^X$ un vecteur dont

$$b_i = \begin{cases} +\text{ excédent en } i \text{ si } i \text{ est une source} \\ -\text{ demande en } i \text{ si } i \text{ est un puits} \\ 0 \qquad \text{sinon,} \end{cases}$$

Soit A la matrice d'incidence du réseau R, c'est-à-dire une matrice ayant autant de lignes que R possède de sommets et autant de colonnes qu'il a d'arcs; dans la colonne correspondante à l'arc (i, j), on met -1 sur la ligne i et $+1$ sur la ligne j et 0 ailleurs.

Le problème de transport s'écrit:

$$\min cx$$

$$\text{avec les contraintes} \begin{cases} Ax = b \\ x \geq 0 \end{cases}$$

On peut ramener le problème de transport à la recherche d'un flot maximum de coût minimum de la façon suivante. On ajoute au réseau R deux sommets s et t, les arcs (s,i) pour toutes les sources $i \in S$ et les arcs (j,t) pour tous les puits $j \in T$. À un arc (s,i) on attribue une capacité égale à l'excédent b_i en i; à un arc (j,t) on attribue une capacité égale à la demande $-b_j$ dans j; à tous les autres arcs on attribue une capacité suffisamment grande (disons, la somme des excédents). On attribue enfin un coût nul aux arcs de la forme (s,i) ou (j,t).

On peut voir facilement que la recherche d'un flot maximum de coût minimum donne bien un solution optimale du problème de transport.

Bibliographie

I. Charon, A. Germa, O. Hudry, Méthodes d'optimisation combinatoire, Masson 1996.

T. Cormen, C. Leiserson, R. Rivest, Introduction à l'algorithmique, Dunod, 1994.

C. Papadimitriou, K. Steiglitz, Combinatorial Optimization. Algorithms and Complexity, Prentice-Hall, 1982.