Conservatoire National des Arts et Métiers
Informatique - Cycle A - 1997/1998
Graphes et Algorithmes

Graphes et Algorithmes

Résumé de cours

Anne Dicky

Sommaire

Organisation du cours

Programme officiel

Présentation du cours :

Les problèmes combinatoires : généralités, difficultés.

Théorie des graphes et algorithmes de base :

Introduction : vocabulaire et concepts de base (connexité, forte connexité, mise en ordre); nombre cyclomatique, arbres et arborescences.

Représentations des graphes : matricielles (adjacence, incidence); listes (successeurs, prédécesseurs). Les graphes en tant qu'outil de modélisation : exemples en Informatique et en Recherche Opérationnelle.

Parcours des graphes en largeur; en profondeur; application : tri topologique d'un graphe sans circuit; détermination des composantes connexes.

Fermeture transitive; déterminations, méthode matricielle ($I + M^{n-1}$), algorithme de ROY-WARSHALL; parcours en profondeur (cas d'un graphe sans circuit). Évaluation du nombre d'opérations; initiation à la complexité dans ce cas polynomial.

Algorithmes d'optimisation dans les graphes valués :

Chemins optimaux dans un graphe valué : algorithmes de FORD, de DIJKSTRA.

Application : ordonnancement de projets (méthodes M.P.M. et P.E.R.T.)

Flots maximaux dans un réseau de transport : l'algorithme de FORD-FULKERSON (exemples, preuve, complexité). Notion de graphe d'écart et reformulation de cet algorithme.

Flot maximal à coût minimal : algorithme de ROY (en E.D.)

Programmes de transport (heuristiques et notion de "regret"; algorithme du *stepping-stone*).

Arbres couvrants de poids extrémal : algorithmes de KRUSKAL, SOLLIN.

Problèmes d'affectation : la méthode hongroise (lien avec les flots maximaux).

Recherches arborescentes : en profondeur d'abord (problème des reines sur l'échiquier); *Branch and Bound* : résolution du problème du voyageur de commerce (*T.S.P.*) par l'algorithme de LITTLE.

Programmation linéaire :

Définition, historique; panorama des applications industrielles, performances et rentabilité.

Approche géométrique : caractérisation géométrique du cheminement vers le sommet optimum. Caractérisation algébrique d'un sommet; méthode algébrique du simplexe. Méthode des tableaux.

Bibliographie

[1] Claude Berge, *Graphes et hypergraphes*, Dunod, 1970 (nombreuses rééditions). Ouvrage de référence sur la théorie des graphes, dépassant largement le cadre du programme.

[2] Gondran et Minoux, *Graphes et algorithmes*, Eyrolles, 1978. Ouvrage dense, d'un niveau élevé (écoles d'ingénieurs), contenant l'ensemble du programme (et bien davantage).

[3] Robert Faure, *Précis de recherche opérationnelle*, Dunod, 1979. Assez complet, mais touffu et embrouillé.

[4] Gérard Lévy, *Algorithmique combinatoire*, Dunod, 1994. Très bonne description des algorithmes.

[5] Groupe ROSEAUX, *Exercices résolus de recherche opérationnelle, t. I : Graphes, t.III : Programmation Linéaire*, Masson, 1978. Publiés par des enseignants du CNAM, exemples et problèmes couvrant le programme.

Renseignements pratiques

Comment joindre l'enseignante

Par courrier : Anne Dicky
LaBRI - Université Bordeaux 1
351, cours de la Libération
33405 Talence Cedex
Par téléphone : 05 56 84 60 87 (bureau) ou 05 57 95 89 51 (personnel)
Par courrier électronique : `dicky@labri.u-bordeaux.fr`

Examens et notes

Un contrôle de connaissances aura lieu en février (dans le cadre du cours), l'examen final ayant lieu en juin. La note transmise sera

$$\max\left(\frac{\textit{note de contrôle} \quad + 2 \times \quad \textit{note d'examen}}{3},\ \textit{note d'examen}\right)$$

Un examen de rattrapage aura lieu en septembre (la note de contrôle de février ne comptant pas pour cette session).
Sur demande des auditeurs, un examen spécial de rappel sera organisé en juin 99.

Déroulement des cours

En règle générale, il sera distribué à chaque séance une feuille d'exercices (à préparer à domicile), ainsi que le corrigé des exercices de la séance précédente.

La première heure de cours sera consacrée à l'explication et au corrigé des exercices de la séance précédente; la deuxième heure, au cours proprement dit.

Programme prévisionnel

Séance	*Date*	*Sujet*
1	7/10	Présentation; exemples de problèmes
2	14/10	Graphes : vocabulaire, représentations;
3	21/10	clôture transitive, algorithme de Warshall
4	28/10	Parcours d'arborescences
5	4/11	Parcours de graphes, applications :
6	18/11	accessibilité, connexité, tri topologique
7	25/11	Problèmes de plus court chemin :
8	2/12	algorithmes de Floyd, de Ford, de Dijkstra
9	9/12	Problèmes de plus long chemin : algorithme de Bellman
10	16/12	Ordonnancement : méthodes MPM, PERT
11	6/1	Arbres recouvrants de poids minimal : algorithmes de Kruskal, de Sollin
12	13/1	Flot maximal :
13	20/1	algorithme de Ford-Fulkerson
14	27/1	Problèmes d'affectation :
15	3/2	algorithme hongrois
16	10/2	Contrôle de connaissances
17	24/2	Flot maximal à coût minimal :
18	3/3	algorithme de Roy
19	10/3	Programmes de transport :
20	17/3	méthode des potentiels,
21	24/3	algorithme du stepping-stone
22	31/3	Programmation linéaire : résolution graphique,
23	21/4	algorithme du simplexe,
24	28/4	méthode des tableaux
25	5/5	Recherches arborescentes

Les modifications éventuelles à ce calendrier seront annoncées au moins 3 semaines à l'avance, et rappelées lors des séances suivantes.

Partie I
Notions élémentaires sur les graphes

1 Exemples de problèmes formalisables par des graphes

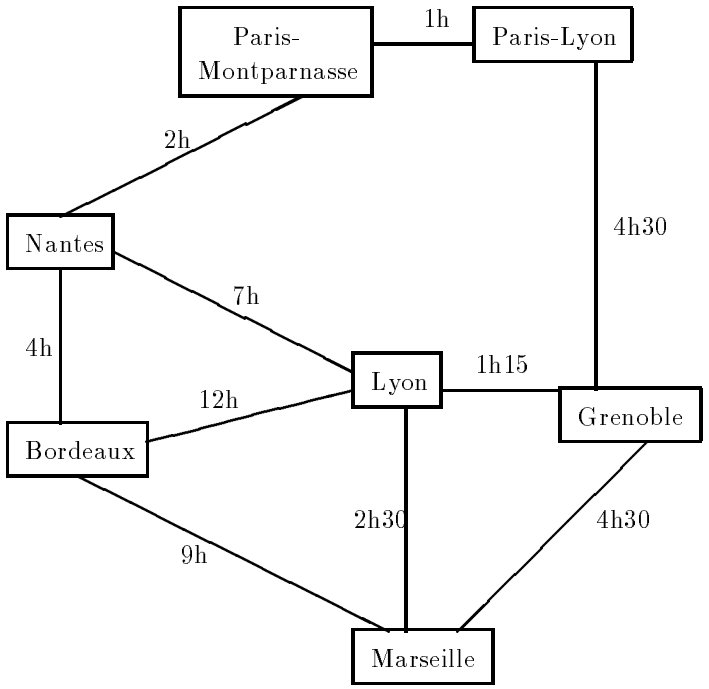
1.1 Choix d'un itinéraire

Sachant qu'une manifestation d'étudiants bloque la gare de Poitiers, et connaissant la durée des trajets suivants :

Bordeaux → Nantes	4 h
Bordeaux → Marseille	9 h
Bordeaux → Lyon	12 h
Nantes → Paris-Montparnasse	2 h
Nantes → Lyon	7 h
Paris-Montparnasse → Paris-Lyon	1 h (en autobus)
Paris-Lyon → Grenoble	4 h 30
Marseille → Lyon	2 h 30
Marseille → Grenoble	4 h 30
Lyon → Grenoble	1 h 15

comment faire pour aller le plus rapidement possible de Bordeaux à Grenoble ?

Les données du problème sont faciles à représenter par un graphe dont les arêtes sont étiquetées par les durées des trajets :

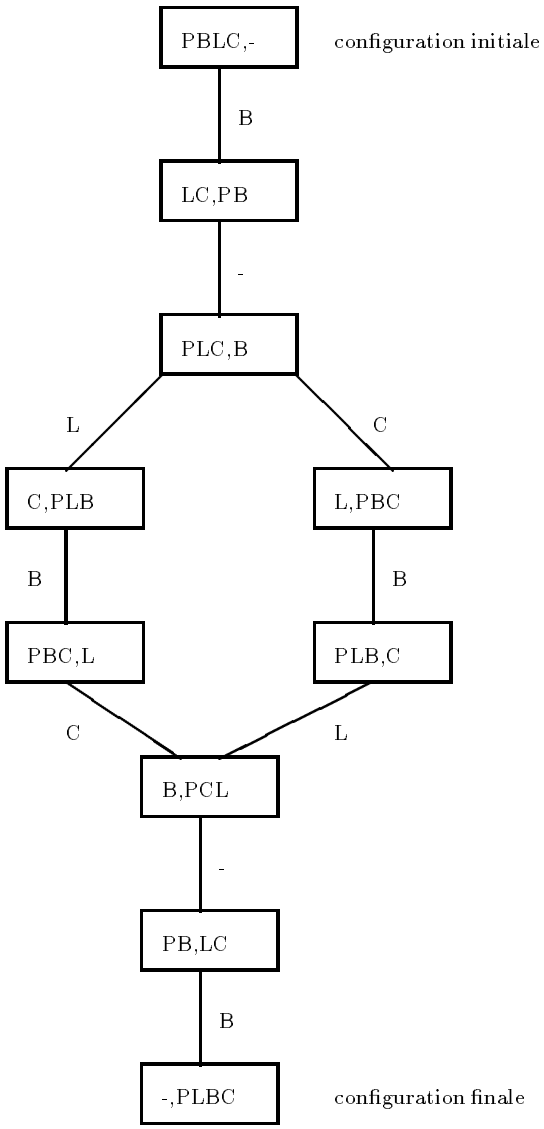


Il s'agit de déterminer, dans ce graphe, le plus court chemin (ou l'un des plus courts chemins, s'il existe plusieurs solutions) entre Bordeaux et Grenoble.

1.2 Le loup, le bouc et le chou

Un passeur se trouve sur le bord d'un fleuve. Il doit faire passer de l'autre côté un loup, un bouc et un chou, mais ne peut transporter qu'un seul client à la fois. Peut-il y arriver, sachant qu'il ne peut laisser seuls ensemble ni le loup et le bouc, ni le bouc et le chou ? Si oui, comment procéder de la façon la plus rapide ?

Le problème peut être formalisé par un graphe dont les sommets sont les configurations acceptables du jeu, et dont les arêtes sont les traversées possibles du passeur (chaque traversée pouvant évidemment être faite dans les deux sens):



Il s'agit de déterminer s'il existe dans le graphe un chemin allant de la configuration initiale (tous les acteurs sur la rive gauche du fleuve) à la configuration finale (tous les acteurs sur la rive droite); et, dans ce cas, on demande un chemin de longueur minimale.

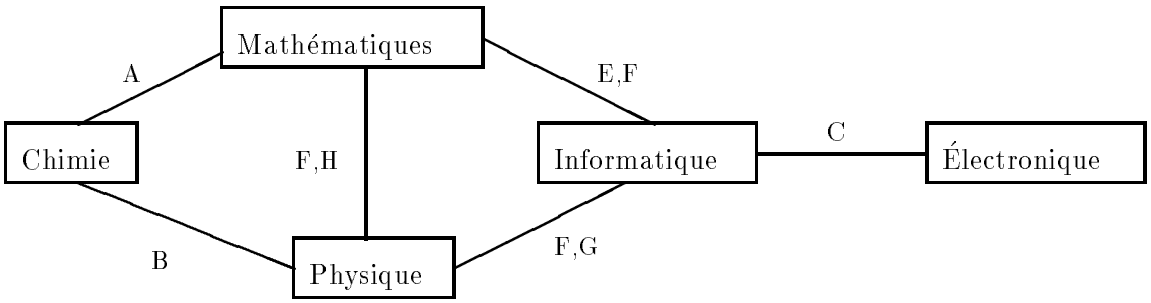
1.3 Organisation d'une session d'examens

Des étudiants A, B, C, D, E et F doivent passer des examens dans différentes disciplines, chaque examen occupant une demi-journée :

Chimie :	*étudiants A et B*
Électronique :	*étudiants C et D*
Informatique :	*étudiants C, E, F et G*
Mathématiques :	*étudiants A, E, F et H*
Physique :	*étudiants B, F, G et H*

On cherche à organiser la session d'examens la plus courte possible.

On peut représenter chacune des disciplines par un sommet, et relier par des arêtes les sommets correspondant aux examens incompatibles (ayant des étudiants en commun) :



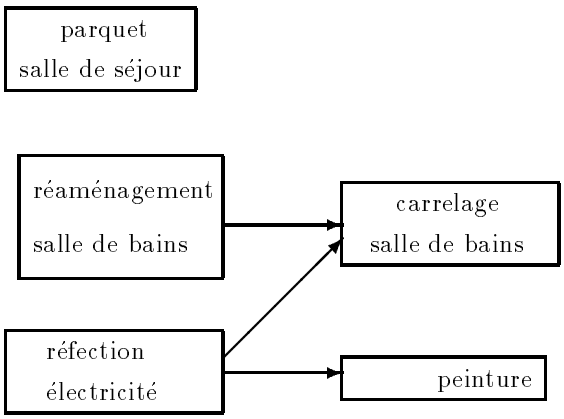
Il s'agit alors de colorier chacun des sommets du graphe en utilisant le moins de couleurs possible, des sommets voisins (reliés par une arête) étant nécessairement de couleurs différentes.

1.4 Planification de travaux

Pour rénover une maison, il est prévu de refaire l'installation électrique (3 jours), de réaménager (5 jours) et de carreler (2 jours) la salle de bains, de refaire le parquet de la salle de séjour (6 jours) et de repeindre les chambres (3 jours), la peinture et le carrelage ne devant être faits qu'après réfection de l'installation électrique.

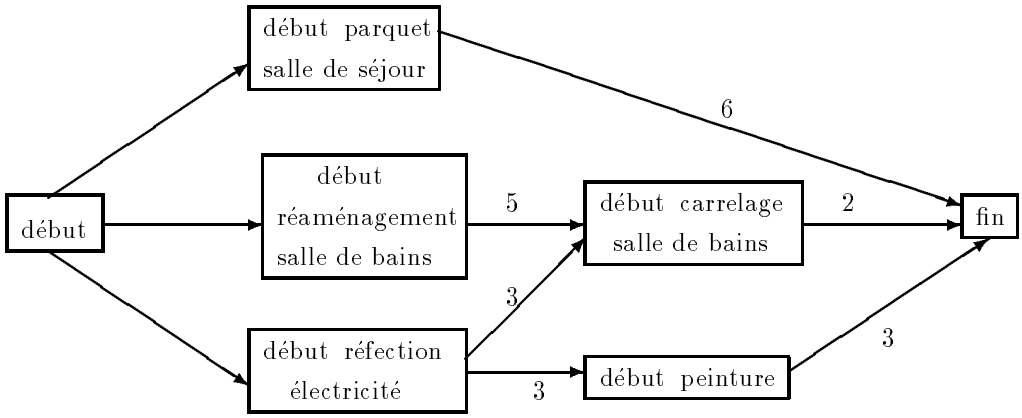
1. *Si le propriétaire décide de tout faire lui-même, dans quel ordre doit-il procéder ?*
2. *Si la rénovation est faite par une entreprise et que chacune des tâches est accomplie par un employé différent, quelle est la durée minimale des travaux ?*

1. On peut représenter les différentes tâches à effectuer par les sommets d'un graphe dont les arcs correspondent aux contraintes de précédence :



   Il s'agit de numéroter les sommets du graphe de façon à respecter les relations de précédence (les successeurs d'un sommet doivent avoir des numéros supérieurs à celui de ce sommet).

2. On peut représenter les différentes étapes de la rénovation sur un graphe dont les arcs sont étiquetés par la durée minimale séparant deux étapes :



   Il s'agit de déterminer la durée du plus long chemin du début à la fin des travaux.

2 Généralités sur les graphes

2.1 Graphes orientés et non orientés

Un **graphe orienté** (*directed graph*) est caractérisé par

- un ensemble S de **sommets** (*vertices*; un sommet : *a vertex*)
- un ensemble A d'**arcs** (*edges*), chaque arc ayant une **origine** (*source*) et un **but** (*target*); un arc a d'origine s, de but t, est généralement noté $s \xrightarrow{a} t$. On dit alors que t est un **successeur** de s, et s un **prédécesseur** de t.

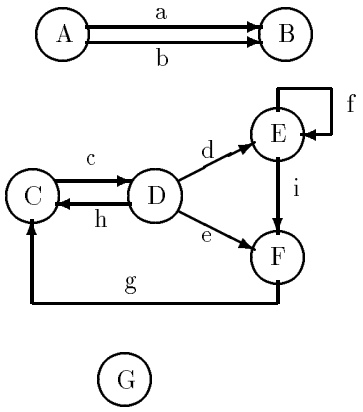


Figure 1: Un exemple de graphe orienté (noté $\mathcal{G}_0$ par la suite)

Si l'on ne s'intéresse pas au sens des arcs, on parlera de graphe **non orienté** dont les sommets sont reliés par des **arêtes**, chaque arête ayant deux **extrémités** (éventuellement confondues).

Tout graphe orienté admet un graphe non orienté **sous-jacent** (ayant pour ensemble d'arêtes l'ensemble de ses arcs).

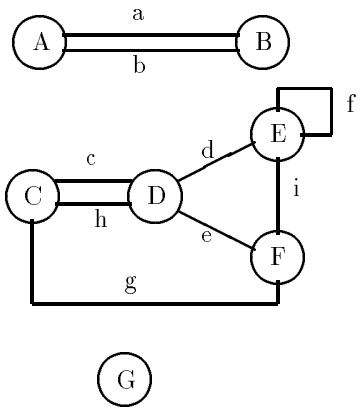


Figure 2: Graphe $\mathcal{G}_1$ (non orienté) sous-jacent à $\mathcal{G}_0$

On peut restreindre un graphe (orienté ou non) à une partie de ses arcs ou arêtes (**graphe partiel**), ou à une partie de ses sommets (**sous-graphe** ou **graphe induit** : on ne considère que les arcs ou arêtes ayant leurs extrémités dans l'ensemble de sommets auquel on se restreint).

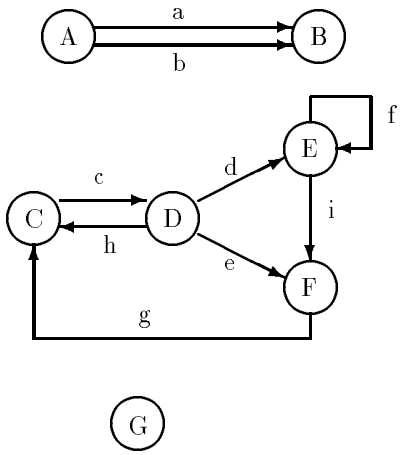
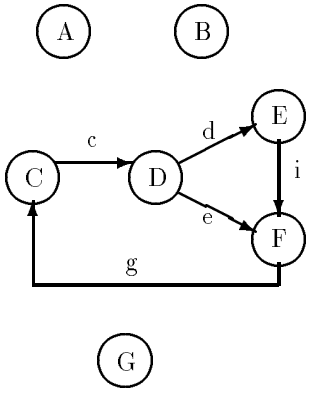
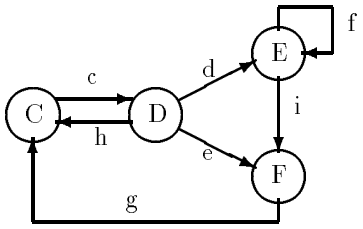


Figure 3: Graphe $\mathcal{G}_0$



$\mathcal{G}_2$: $\mathcal{G}_0$ restreint aux arcs c,d,e,g,i



$\mathcal{G}_3$: $\mathcal{G}_0$ restreint aux somments C,D,E,F

Figure 4: Graphe partiel $\mathcal{G}_2$ et sous-graphe $\mathcal{G}_3$

Un graphe est dit **simple** s'il existe au plus un arc (une arête, dans le cas non orienté) d'origine et de but donnés; dans ce cas, l'ensemble A peut être défini comme une partie de $S \times S$ (graphe d'une relation binaire), un arc (s,t) étant généralement noté $s \longrightarrow t$.

Par exemple, $\mathcal{G}_0$ n'est pas un graphe simple (les arcs a et b ont même origine et même extrémité), mais les graphes restreints $\mathcal{G}_2$ et $\mathcal{G}_3$ sont des graphes simples.

Un graphe orienté simple est dit **antisymétrique** si, pour aucun arc $s \rightarrow t$, le graphe ne comporte d'arc $t \rightarrow s$.

Par exemple, $\mathcal{G}_2$ est un graphe antisymétrique, mais $\mathcal{G}_3$ ne l'est pas (il comporte des arcs $C \rightarrow D$ et $D \rightarrow C$; en outre, un graphe antisymétrique ne peut comporter d'arc dont l'origine et le but sont confondus, comme $E \rightarrow E$).

2.2 Chemins, circuits et cycles

Un **chemin** d'un graphe orienté est une suite d'arcs, l'origine d'un arc étant le but de l'arc précédent : $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \ldots s_{n-1} \xrightarrow{a_n} s_n$ désigne un chemin d'**origine** s_0, de **but** s_n, de longueur n.

Un chemin est dit **simple** s'il ne passe pas deux fois par le même arc.

Un **circuit**, ou **cycle**, est un chemin dont l'origine et le but sont confondus; un circuit est dit **élémentaire** s'il ne passe pas deux fois par le même sommet (les sommets sont deux à deux distincts, sauf l'origine et le but). Une **boucle** est un circuit composé d'un seul arc.

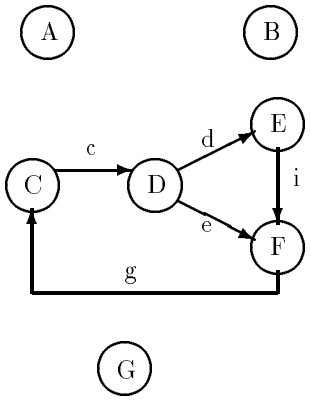


Figure 5: Le graphe $\mathcal{G}_2$

Exemples : $C \xrightarrow{c} D \xrightarrow{d} E \xrightarrow{i} F$ est un chemin simple de $\mathcal{G}_2$; $C \xrightarrow{c} D \xrightarrow{e} F \xrightarrow{g} C$ en est un circuit élémentaire.

Un **dag** (pour *directed acyclic graph*) est un graphe orienté sans circuit.

On parlera également de **chemin**, de **cycle**[1] ou de **boucle** d'un graphe non orienté (ce sont les chemins, circuits ou boucles des graphes orientés admettant ce graphe pour graphe non orienté sous-jacent).

Étant donné un graphe orienté antisymétrique, tout cycle élémentaire du graphe non orienté sous-jacent, muni d'un sens de parcours, peut s'écrire de façon unique comme somme ou différence d'arcs.

Exemples : le cycle $CDFC$ de $\mathcal{G}_2$ peut s'écrire $c + e + g$; le cycle $DFED$ peut s'écrire $e - i - d$.

Un cycle quelconque peut s'écrire comme une somme de cycles élémentaires (il suffit d'entamer un nouveau cycle élémentaire chaque fois que l'on rencontre un sommet déjà vu).

Une **base de cycles** est un ensemble de cycles élémentaires tels que tout cycle puisse s'écrire, de façon unique, comme somme ou différence de cycles de la base; on admettra que toutes les bases de cycles ont le même nombre d'éléments, qui constitue le **nombre cyclomatique** du graphe.

Exemple : le nombre cyclomatique $\mathcal{G}_2$ est 2 (les cycles $DEFD$ et $CDFC$ constituent une base).

[1] On réserve souvent le terme **circuit** aux graphes orientés, et le terme **cycle** aux graphes non orientés; mais la distinction n'est pas normalisée.

2.3 Clôture transitive d'un graphe

La **clôture transitive** (ou **fermeture transitive** d'un graphe simple (orienté ou non) G est le graphe dont les sommets sont ceux de G, et les arcs (ou arêtes) sont les $s \longrightarrow t$ tels qu'il existe dans G un chemin du sommet s au sommet t.

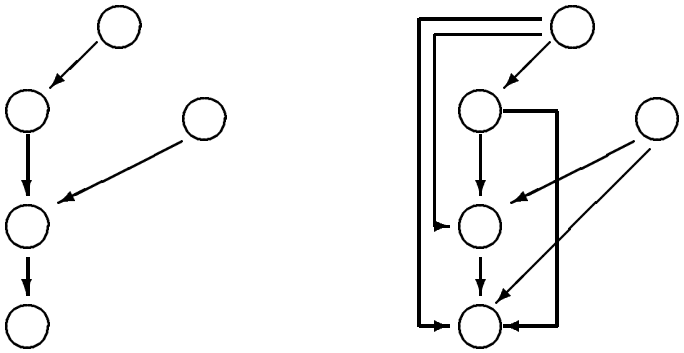


Figure 6: Un graphe simple et sa clôture transitive

2.4 Connexité, forte connexité

On dit qu'un graphe (non orienté) est **connexe** s'il existe un chemin entre deux sommets quelconques de ce graphe; la **composante connexe** d'un sommet s est l'ensemble des sommets qui lui sont reliés.

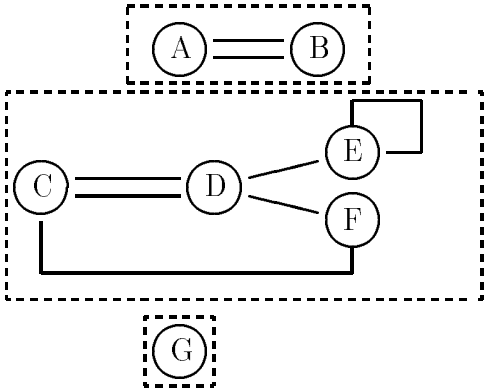


Figure 7: Les 3 composantes connexes de $\mathcal{G}_1$

On dit qu'un graphe orienté est **fortement connexe** si pour tous sommets s et t, il existe un chemin allant de s à t; la **composante fortement connexe** d'un sommet s est l'ensemble des sommets t tels qu'il existe un chemin de s à t et un chemin de t à s.

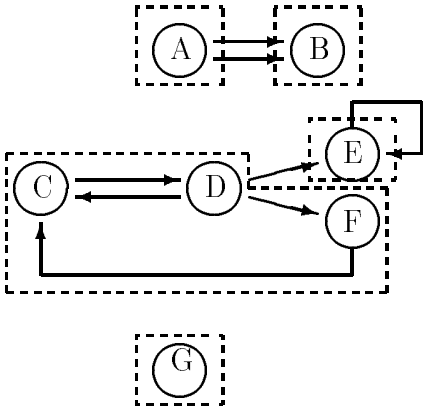


Figure 8: Les 5 composantes fortement connexes de $\mathcal{G}_0$

2.5 Arbres et arborescences

Un **arbre** est un graphe (non orienté) connexe sans cycle simple; dans un tel graphe, deux sommets quelconques sont reliés par un et un seul chemin (le nombre de sommets et le nombre d'arcs sont liés par la relation $|A| = |S| - 1$).

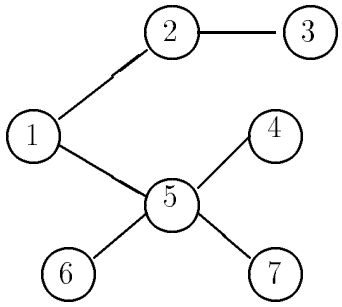


Figure 9: Arbre

Une **arborescence** est un graphe orienté possédant un sommet privilégié, la **racine**, de façon qu'il existe un et un seul chemin de la racine à tout autre sommet.

On peut orienter les arêtes d'un arbre à partir de n'importe quel sommet de façon à obtenir une arborescence :

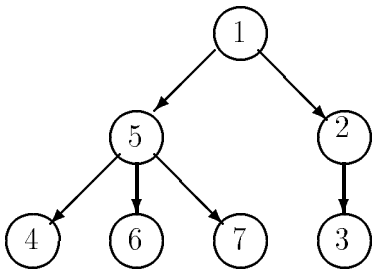


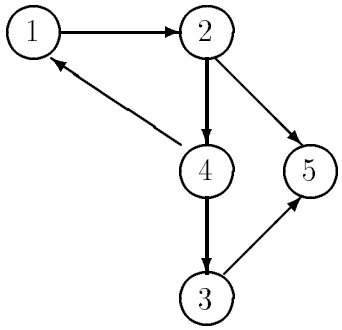
Figure 10: Arborescence obtenue en prenant pour racine le sommet 1

En pratique, le mot *arbre* désigne aussi bien une arborescence qu'un arbre au sens strict (non orienté).

3 Représentation des graphes en informatique

3.1 Structure associée à la représentation graphique

Un graphe peut être représenté par une liste de sommets, chacun étant caractérisé par son nom, une étiquette éventuelle, et la liste des arcs qui ont ce sommet pour origine (eux-mêmes caractérisés par leur but, et une étiquette éventuelle):



sommet 1 : 1→2
sommet 2 : 2→4,2→5
sommet 3 : 3→5
sommet 4 : 4→1,4→3
sommet 5 :

Figure 11: Représentation par listes de listes d'arcs

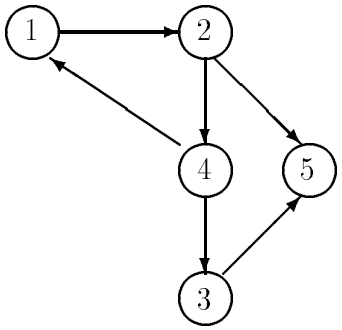
Avantages : simplicité de la mise à jour, facilité de parcours;

Inconvénients : redondance de la représentation pour les graphes non orientés; temps d'accès aux données (adressage); le parcours ne s'effectue que dans le sens des arcs pour les graphes orientés. (Ce dernier inconvénient peut être supprimé en ajoutant, pour chaque sommet, la liste des arcs qui ont ce sommet pour but, au prix d'un encombrement de la mémoire.)

3.2 Représentations matricielles

Matrice d'adjacence (graphes simples)

Un graphe simple non étiqueté à n sommets numérotés peut être représenté par une matrice carrée (n, n) d'entiers : l'élément $M[i, j]$ vaut 1 s'il existe un arc allant du sommet i au sommet j, 0 sinon :



	1	2	3	4	5
1	0	1	0	0	0
2	0	0	0	1	1
3	0	0	0	0	1
4	1	0	1	0	0
5	0	0	0	0	0

Figure 12: Représentation par matrice d'adjacence

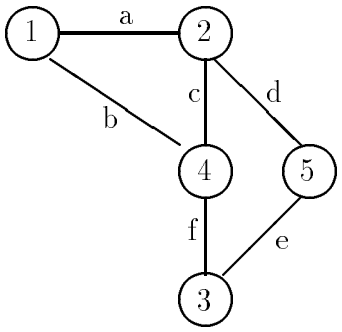
Dans le cas où les arcs sont étiquetés, la matrice sera constituée des étiquettes, à condition de pouvoir caractériser l'absence d'arc par une valeur particulière d'étiquette.

Avantages : rapidité des recherches, compacité de la représentation, simplicité des algorithmes de calcul;

Inconvénients : représentation ne convenant qu'aux graphes simples; redondance des informations pour les graphes non orientés; stockage inutile de cas inintéressants (les zéros de la matrice), à examiner quand on parcourt le graphe (pour la complexité des algorithmes, $|A|$ est à remplacer par $|S|^2$).

Matrice d'incidence

Un graphe non orienté à n sommets numérotés et p arcs numérotés peut être représenté par une matrice (n, p) d'entiers : l'élément $M[i, j]$ vaut 1 si le sommet i est une extrémité de l'arc j, 0 sinon :



	a	b	c	d	e	f
1	1	1	0	0	0	0
2	1	0	1	1	0	0
3	0	1	0	0	1	1
4	0	0	1	0	0	1
5	0	0	0	1	1	0

Figure 13: Représentation par matrice d'incidence

Dans le cas où les arcs sont étiquetés, la matrice sera constituée des étiquettes, à condition de pouvoir caractériser l'absence d'arc par une valeur particulière d'étiquette.

Avantages : rapidité des recherches, compacité de la représentation, informations non redondantes pour les graphes non orientés;

Inconvénients : stockage et examen inutile de zéros; les calculs de matrices classiques ne s'appliquent pas.

4 Calculs matriciels

4.1 Produit de matrices

Étant données des matrices A et B de dimension (n, n), leur produit P est une matrice de même dimension définie par

$$P[i,j] = \sum_{k=1}^{k=n} A[i,k] \times B[k,j]$$

La complexité de l'algorithme de calcul est de l'ordre de n^3 (3 boucles imbriquées dans lesquelles on passe n fois).

4.2 Chemins de longueur donnée

Si M est la matrice d'adjacence d'un graphe G, M^p (puissance au sens du produit de matrices) vérifie

$$M^p[i,j] = 1 \Leftrightarrow \text{il existe un chemin de } p \text{ arcs allant du sommet } i \text{ au sommet } j$$

La complexité d'un algorithme naïf de calcul de M^p est de l'ordre de $p \times |S|^3$ ($p-1$ multiplications de matrices). On peut la réduire à $\log(p) \times |S|^3$ en utilisant l'algorithme récursif

```
SI p = 1 ALORS
     M^p := M
SINON
     J := M^⌊p/2⌋ ;
     K := J × J ;
     SI p est pair ALORS M^p := K
     SINON M^p := K × M
     FIN SI
FIN SI
```

4.3 Clôture transitive

S'il existe un chemin entre deux sommets de G, il existe un chemin élémentaire entre ces sommets, de longueur $< n$: la matrice de la clôture transitive est donc la somme logique $M + M^2 + \ldots + M^{|S|-1}$.

4.4 Algorithme de Warshall

On peut calculer la matrice T de la clôture transitive de façon plus rapide :

```
T := M ;
POUR i allant de 1 à n
     POUR j allant de 1 à n
          SI T[j,i] = 1 ALORS
               POUR k allant de 1 à n
                    SI T[i,k] = 1 ALORS T[j,k] := 1
               FIN POUR
          FIN SI
     FIN POUR
FIN POUR
```

Le principe de l'algorithme est le suivant : pour chaque sommet i, on examine tous les sommets j et k : s'il existe un chemin de j à i et un chemin de i à k, on en déduit qu'il existe un chemin de j à k.

Preuve de l'algorithme : par récurrence sur i, on montre qu'avant le i-ième passage dans la boucle `POUR` extérieure, chacun des $T[j,k]$ est à 1 si et seulement s'il existe un chemin du sommet j au sommet k dont les sommets intermédiaires sont d'indice inférieur à i : la démonstration repose sur le fait que si l'on peut aller de j à k en passant par i, on peut y aller en ne passant qu'*une fois* par i.

Complexité : de l'ordre de $|S|^3$. Inconvénient : si la matrice est "creuse" (contient beaucoup de zéros, c'est-à-dire que le graphe contient peu d'arcs), on fait beaucoup de calculs inutiles.

5 Parcours des graphes

5.1 Parcours des arborescences en largeur

Le parcours **en largeur** (*breadth-first search*) d'une arborescence consiste à visiter les nœuds dans l'ordre "de gauche à droite et de haut en bas" :

```
parcourir une arborescence en largeur :
     enfiler la racine;
     TANT QUE la file n'est pas vide
          soit n le premier nœud de la file;
          { traitement de n }
          defiler;
          enfiler tous les fils de n
     FIN TANT QUE
```

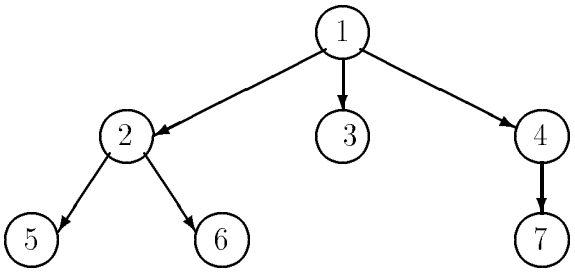


Figure 14: Ordre de parcours en largeur : 1, 2, 3, 4, 5, 6, 7

5.2 Parcours des arborescences en profondeur

Le parcours récursif **en profondeur** (*depth-first search*) d'une arborescence consiste à visiter la racine, la visite d'un nœud étant terminée lorsqu'on a visité tous ses fils :

```
parcourir une arborescence en profondeur :
     visiter la racine

visiter un nœud n :
     { pré-traitement de n }
     visiter tous les fils de n
     { post-traitement de n }
```

L'ordre de visite est "de haut en bas et de gauche à droite" (ordre préfixe).
L'ordre de post-visite est "de bas en haut et de gauche à droite" (ordre postfixe).

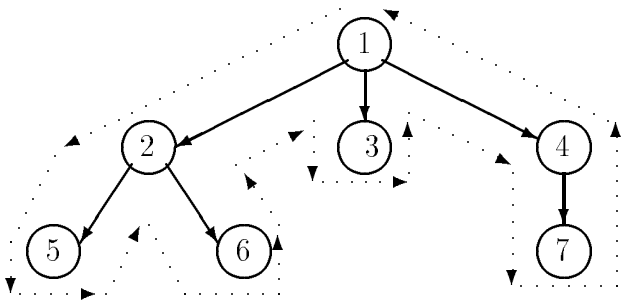


Figure 15: Ordre de visite : 1, 2, 5, 6, 3, 4, 7; de post-visite : 5, 6, 2, 3, 7, 4, 1

On peut effectuer un parcours en profondeur non récursif en marquant les nœuds en cours de traitement, et en utilisant une pile pour garder la trace des nœuds en attente :

```
parcourir une arborescence en profondeur :
empiler la racine;
TANT QUE la pile n'est pas vide
     soit s le sommet de pile;
     SI s n'a pas été marqué ALORS
          { pré-traitement de s }
          marquer s;
          SI s a des fils ALORS
               empiler le fils aîné de s
          FIN SI
     SINON
          { post-traitement de s }
          dépiler;
          SI s a des frères plus jeunes ALORS
               empiler le frère cadet de s
          FIN SI
     FIN SI
FIN TANT QUE
```

5.3 Parcours de graphes

Les algorithmes de parcours des arborescences s'adaptent aux graphes, en considérant que l'on parcourt chacune des arborescences associées aux sommets du graphe; l'arborescence associée à un sommet est celle des descendants de ce sommet que l'on n'a pas encore visités.

Parcours en largeur

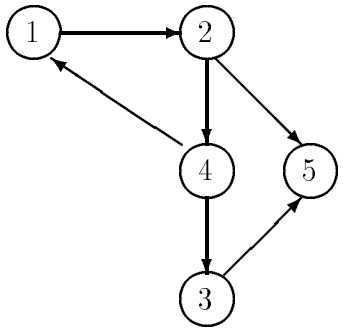


Figure 16: Ordre de parcours : 1, 2, 4, 5, 3

```
parcourir un graphe en largeur :
     visiter tous les sommets

visiter un sommet s :
     SI s n'a pas encore été enfilé ALORS
          enfiler s ;
          TANT QUE la file n'est pas vide
               soit n le premier sommet de la file;
               { traitement de n}
               défiler;
               enfiler tous les successeurs non encore enfilés de n
          FIN TANT QUE
     FIN SI
```

Preuve du programme : chaque sommet est visité, donc enfilé au moins une fois; or, on ne peut enfiler que des sommets non encore enfilés; par conséquent, chaque sommet est enfilé exactement 1 fois.

À chaque passage dans la boucle `TANT QUE`, on défile un sommet différent, donc chaque procédure `visiter` se termine. À la sortie de `visiter`, la file est vide; donc chaque sommet aura été défilé, et par conséquent traité, exactement 1 fois.

Complexité : Chaque sommet est traité 1 fois, et chaque arc est examiné 1 fois (lors du traitement de son origine). La complexité est donc de l'ordre de $\max(|A|, |S|)$.

Parcours en profondeur (récursif)

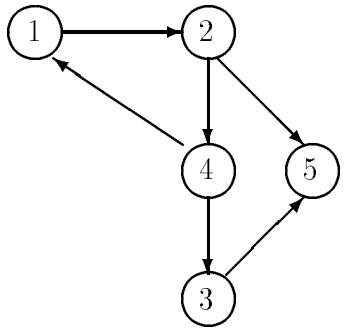


Figure 17: Ordre de visite : 1, 2, 4, 3, 5; de post-visite : 1, 5, 3, 4, 2

```
parcourir un graphe en profondeur :
     visiter tous les sommets

visiter un sommet s :
     SI s n'a pas encore été visité ALORS
          { pré-traitement de s}
          visiter tous les successeurs de s
               { post-traitement de s}
     FIN SI
```

Preuve du programme : chaque sommet est visité exactement 1 fois.

Complexité : chaque sommet est visité 1 fois, chaque arc d'origine est examiné 1 fois. La complexité est donc de l'ordre de $\max(|A|, |S|)$.

5.4 Applications des parcours de graphes

Accessibilité

Pour connaître les sommets accessibles depuis un sommet donné d'un graphe (orienté ou non), il suffit de faire un parcours en profondeur à partir de ce sommet, en marquant les sommets visités :

```
marquage des accessibles depuis un sommet s :
     visiter s

visiter un sommet t :
     SI t n'a pas encore été visité ALORS
          marquer t;
          visiter tous les successeurs de t
     FIN SI
```

Composantes connexes d'un graphe non orienté

La composante connexe d'un sommet s est l'ensemble des sommets accessibles depuis s. On peut colorier les composantes connexes en adaptant l'un des algorithmes de parcours : par exemple (parcours en profondeur)

```
colorier les composantes connexes :
     POUR tout sommet s
          SI s n'a pas encore été visité ALORS
               choisir une nouvelle couleur;
               visiter s
          FIN SI
     FIN POUR

visiter un sommet s :
     SI s n'a pas encore été visité ALORS
          colorier s;
          visiter tous les successeurs de s
     FIN SI
```

Graphe orienté sans circuit

Un graphe orienté comporte un circuit si et seulement si, lors du parcours des sommets accessibles depuis un sommet, on retombe sur ce sommet. Pour savoir si un graphe est sans circuit, il suffit donc d'adapter l'un des algorithme de parcours, en maintenant une liste des sommets critiques (en cours de visite):

```
graphe sans circuit :
     POUR tout sommet s
          SI s n'a pas encore été visité ALORS
               visiter s
          FIN SI
     FIN POUR ;
     répondre ``oui''

visiter un sommet s :
     SI s est dans la liste critique ALORS
          répondre ``non'';
          sortir
     SINON
          ajouter s à la liste critique;
          visiter tous les successeurs de s;
          enlever s de la liste critique
     FIN SI
```

Tri topologique d'un graphe orienté sans circuit

Le tri topologique d'un graphe orienté sans circuit est une numérotation des sommets dans laquelle les descendants d'un sommet de numéro k sont nécessairement de numéro supérieur à k.

L'ordre inverse de l'ordre de post-visite, dans le parcours en profondeur du graphe, permet son tri topologique : en effet, lorsqu'on a terminé la visite d'un sommet s, on a terminé la visite de tous ses descendants. Par conséquent, si t vient avant s dans l'ordre de post-visite, s n'est pas un descendant de t.

```
numéroter les n sommets du graphe :
     k := n;
     visiter tous les sommets

visiter un sommet s :
     SI s n'a pas encore été visité ALORS
          visiter tous les successeurs de s;
          affecter à s le numéro k ;
          k := k − 1
     FIN SI
```

Partie II
Problèmes classiques sur les graphes

6 Algorithmes de plus court chemin

On considère des graphes simples *valués*, c'est-à-dire dont chaque arc est muni d'un **poids** (nombre réel). Un chemin a pour poids la somme des poids des arcs qui le constituent.

Le problème des plus courts chemins consiste à déterminer le poids minimal d'un chemin d'un sommet à un autre, en supposant les poids **positifs**.

6.1 Algorithme de Floyd

On représente un graphe orienté valué à n sommets par une matrice carrée (n, n) de nombres : $M[i, j]$ a pour valeur le poids de l'arc $i \longrightarrow j$ ($+\infty$ si cet arc n'existe pas).

Une variante de l'algorithme de Warshall permet de calculer la matrice D telle que $D[i, j]$ soit le poids minimal d'un chemin de i à j :

```
D := M ;
POUR i allant de 1 à n
     POUR j allant de 1 à n
          SI D[j,i] < +∞ ALORS
               POUR k allant de 1 à n
                    SI D[j,k] > D[j,i] + D[i,k] ALORS
                         D[j,k] := D[j,i] + D[i,k]
                    FIN SI
               FIN POUR
          FIN SI
     FIN POUR
FIN POUR
```

Cet algorithme résout le problème des plus courtes distances de tout sommet du graphe à tout autre sommet, mais ne donne pas le chemin correspondant.

Preuve de l'algorithme : on montre par récurrence sur i qu'au début du i-ième passage dans la boucle `POUR` extérieure, chacun des $D[j, k]$ représente le poids minimal des chemins de j à k dont les sommets intermédiaires sont d'indice strictement inférieur à i.

Complexité : $|S|^3$.

Les algorithmes qui suivent permettent de déterminer les plus courts chemins d'un sommet particulier s du graphe à tout autre sommet. On notera $p(a \longrightarrow b)$ le poids d'un arc $a \longrightarrow b$.

6.2 Algorithme de Ford

```
POUR tout sommet t
     d(s,t) := +∞
FINPOUR;
d(s,s) := 0;
RÉPÉTER
     POUR tout sommet t
          SI d(s,t) < +∞ ALORS
               POUR tout successeur u de t
                    SI d(s,u) > d(s,t) + p(t ⟶ u) ALORS
                         d(s,u) := d(s,t) + p(t ⟶ u)
                    FIN SI
               FIN POUR
          FIN SI
     FIN POUR
TANT QU'on modifie quelque chose
```

Preuve de l'algorithme : on montre inductivement qu'au i-ième passage dans la boucle RÉPÉTER, chacun des $d(s,u)$ a pour valeur le poids du plus court chemin de s à u comprenant moins de i sommets intermédiaires. Il s'ensuit que l'algorithme termine (on ne peut passer plus de $|S|-1$ fois dans la boucle RÉPÉTER) et qu'il calcule bien les poids minimaux cherchés.

Complexité : $|S| \times |A|$.

6.3 Algorithme de Dijkstra

```
POUR tout sommet t
     d(s,t) := +∞
FINPOUR;
d(s,s) := 0;
TANT QU'il reste des sommets non fixés
     choisir un sommet t non fixé tel que d(s,t) soit minimale;
     fixer t;
     POUR tout successeur u de t
          SI d(s,u) > d(s,t) + p(t ⟶ u) ALORS
               d(s,u) := d(s,t) + p(t ⟶ u)
          FIN SI
     FIN POUR
FIN TANT QUE
```

Preuve de l'algorithme : l'algorithme se termine, puisqu'à chaque passage dans la boucle `TANT QUE` on fixe un et un seul sommet.

On montre par récurrence qu'à tout stade de l'algorithme, pour tout sommet fixé t, $d(s,t)$ est le poids minimal d'un chemin de s à t. Supposons en effet cette propriété vérifiée avant que l'on ne fixe le sommet t; soit $s = s_0 \to s_1 \to \ldots \to s_{n-1} \to s_n = t$ un chemin quelconque de s à t; soit k le plus petit entier tel que $s_0, s_1, \ldots, s_k$ soient des sommets fixés; la façon dont t est choisi implique que la distance $d(s,t)$ est inférieure ou égale à $d(s,s_k) + d(s_k, s_k+1)$, donc, a fortiori, à la longueur du chemin $s \to s_1 \to \ldots \to s_{n-1} \to t$, et par conséquent $d(s,t)$ est bien le poids minimal d'un chemin de s à t.

Complexité : on passe exactement $|S|$ fois dans la boucle `TANT QUE`, et on examine 1 fois chacun des arcs (au moment où l'on fixe son origine). La détermination du sommet non fixé à distance minimale de s peut se faire en $\log|S|$ opérations élémentaires : la complexité est donc $\max(|S|\log|S|, |A|)$. Cet algorithme est donc plus efficace que les algorithmes de Ford et de Floyd.

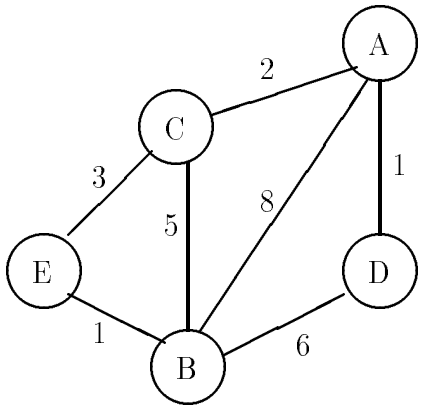
Une variante de l'algorithme de Dijkstra permet de déterminer, pour chaque sommet t accessible depuis s, un chemin de poids minimal de s à t : il suffit d'associer un arc $a(t)$ à chaque sommet t distinct de s, en mettant à jour les arcs $a(t)$ en même temps que les distances :

```
...
          SI d(s,u) > d(s,t) + p(t ⟶ u) ALORS
               d(s,u) := d(s,t) + p(t ⟶ u);
               a(u) := t ⟶ u
          FIN SI
...
```

Les arcs $a(t)$ définissent une arborescence de racine s, correspondant à des chemins de poids minimal d'origine s.

Exemple d'application de l'algorithme de Dijkstra

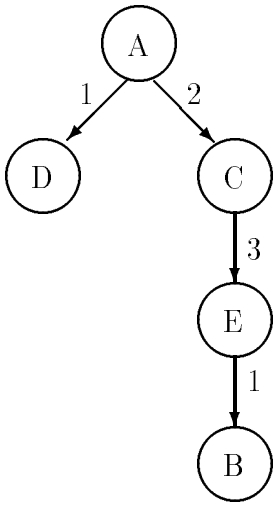
Soit à chercher les plus courts chemins depuis le sommet A dans le graphe suivant :



Le tableau suivant donne les distances depuis A et les arcs marqués au cours des différentes étapes de l'algorithme :

sommets	A	B	C	D	E
initialement	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$
on fixe A		8 (AB)	2 (AC)	1 (AD)	$+\infty$
on fixe D		7 (DB)	2		$+\infty$
on fixe C		7			5 (CE)
on fixe E		6 (EB)			
valeurs finales	0	6 (EB)	2 (AC)	1 (AD)	5 (CE)

On obtient ainsi l'arborescence



7 Algorithmes de plus long chemin

Le problème consiste à déterminer le poids du plus *long* chemin acyclique reliant un sommet à un autre du graphe.

Il n'est pas possible d'adapter directement les algorithmes de plus court chemin, compte tenu de l'existence possible de cycles dans le graphe.

Dans le cas général, on peut parcourir l'arbre des chemins acycliques partant d'un sommet s, au moyen d'un algorithme de parcours en profondeur utilisant une liste :

parcourir le graphe :

```
POUR tout sommet t différent de s
    d(s,t) := +∞
FINPOUR;
d(s,s) := 0;
visiter s
```

visiter un sommet t :

```
ajouter t à la liste;
POUR tout successeur u de t qui n'est pas dans la liste
    SI d(s,u) = +∞ ou d(s,u) < d(s,t) + p(t ⟶ u) ALORS
        d(s,u) := d(s,t) + p(t ⟶ u)
    FIN SI;
    visiter u
FIN POUR;
enlever t de la liste
```

Preuve de l'algorithme : on montre récursivement que lorsqu'on visite un sommet t qui n'est pas dans la liste, la liste comporte la suite des sommets d'un chemin acyclique allant de s à t; d'autre part, les états de la liste sont distincts à chacune des entrées dans la procédure `visiter`. Par conséquent l'algorithme termine (il ne peut y avoir qu'un nombre fini d'états de la liste, correspondant aux chemins acycliques depuis le sommet s).

Pour tout sommet t, à tout instant, si $d(s,t) < +\infty$, $d(s,t)$ représente le poids d'un chemin acyclique allant de s à t. Comme tous les chemins sont examinés, on obtiendra bien le poids maximal.

Complexité : dans le pire des cas (graphe complet), le nombre d'entrées dans la procédure `visiter` est le nombre de chemins acycliques d'origine s, c'est-à-dire

$$\sum_{k=1}^{k=|S|} (|S|-1)(|S|-2)\dots(|S|-k+1)$$

(chaque terme de la somme est le nombre de chemins acycliques d'origine s comportant k sommets). À chacun de ces appels, on examine les $|S|$ successeurs du sommet que l'on visite. La complexité est donc de l'ordre de $|S|!$

Dans le cas particulier d'un graphe acyclique, on peut adapter efficacement les algorithmes de plus court chemin à la recherche des poids maximaux.

La modification des algorithmes de Ford ou de Floyd consiste simplement à changer le sens de l'inégalité; la modification de l'algorithme de Dijkstra est un peu plus compliquée (algorithme de Bellmann).

7.1 Algorithme de Ford modifié

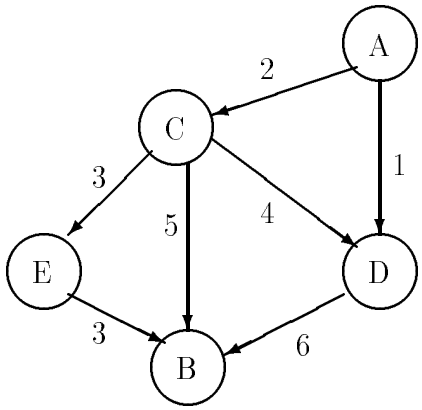
```
POUR tout sommet t
     d(s,t) := +∞
FINPOUR;
d(s,s) := 0;
RÉPÉTER
     POUR tout sommet t
          SI d(s,t) < +∞ ALORS
               POUR tout successeur u de t
                    SI d(s,u) < d(s,t) + p(t ⟶ u) ALORS
                         d(s,u) := d(s,t) + p(t ⟶ u)
                    FIN SI
               FIN POUR
          FIN SI
     FIN POUR
TANT QU'on modifie quelque chose
```

7.2 Algorithme de Bellmann

```
POUR tout sommet t
     d(s,t) := +∞
FINPOUR;
d(s,s) := 0;
fixer s;
TANT QU'il reste des sommets non fixés
     choisir un sommet t non fixé dont tous les prédécesseurs sont fixés;
     POUR tout successeur u de t
          SI d(s,u) > d(s,t) + p(t ⟶ u) ALORS
               d(s,u) := d(s,t) + p(t ⟶ u)
          FIN SI
     FIN POUR;
     fixer t
FIN TANT QUE
```

Exemple d'application de l'algorithme de Bellman

Soit à chercher les plus longues distances depuis le sommet A dans le graphe suivant :



Le tableau suivant donne les plus longues distances depuis A calculées au cours des différentes étapes de l'algorithme :

sommets	A	B	C	D	E
initialement	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$
on fixe A		$+\infty$	2	1	$+\infty$
on fixe C		7		6	5
on fixe D		7			5
on fixe E		8			
valeurs finales	0	8	2	6	5

8 Problèmes d'ordonnancement

Un problème d'ordonnancement *simple* est caractérisé par un ensemble de *tâches* à exécuter, chacune ayant une durée déterminée, et des *contraintes de précédence* entre les tâches : c'est-à-dire qu'une tâche, pour être exécutée, requiert qu'une ou plusieurs autres tâches aient été accomplies. Le problème consiste à planifier l'exécution des tâches.

Dans la pratique, les problèmes d'ordonnancement sont plus complexes : les durées d'exécution peuvent être imprécises (données sous forme d'intervalles, avec éventuellement une loi de probabilité); d'autres contraintes temporelles peuvent surgir (contraintes absolues : telle tâche doit impérativement être commencée après telle date, ou terminée avant telle date; contraintes disjonctives : telle et telle tâches ne doivent pas être exécutées simultanément...) On se limitera à l'étude des problèmes d'ordonnancement simples.

Quelques définitions :

- une tâche est dite **critique** si son exécution ne peut être différée ou allongée sans retarder la fin des travaux;
- la **marge libre** d'une tâche est le temps maximal dont cette tâche peut être différée ou allongée sans retarder la fin des travaux, indépendamment des autres tâches;
- la **marge totale** d'une tâche est le temps maximal dont cette tâche peut être différée ou allongée sans retarder la fin des travaux (ce qui peut éventuellement contraindre une autre tâche non critique).

On reprendra l'exemple de la planification de travaux donné en page 8 :

> *Pour rénover une maison, il est prévu de refaire l'installation électrique (3 jours), de réaménager (5 jours) et de carreler (2 jours) la salle de bains, de refaire le parquet de la salle de séjour (6 jours) et de repeindre les chambres (3 jours), la peinture et le carrelage ne devant être faits qu'après réfection de l'installation électrique.*
>
> *Quelle est la durée minimale des travaux, et comment doit-on les ordonner ?*

8.1 La méthode des potentiels/tâches (MPM)

Dans la méthode MPM (*Method of Project Management*), on représente chaque tâche par un et un seul sommet d'un graphe dont les arcs sont les contraintes de précédence; chaque arc est valué par la durée de la tâche source.

On ajoute deux tâches fictives `début` et `fin`, et les arcs `début` $\longrightarrow s$ (de valeur 0) pour tout sommet s sans prédécesseur et $s \longrightarrow$ `fin` pour tout sommet s sans successeur (valué par la durée de la tâche s).

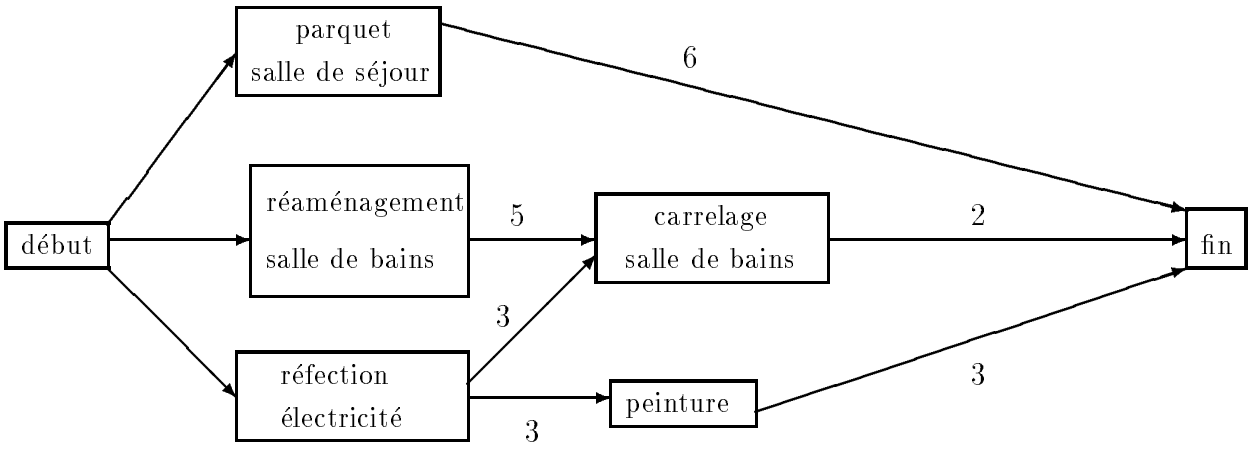


Figure 18: Modélisation MPM

Il va de soi que le problème n'a de solution que si le graphe obtenu est acyclique.

On peut alors calculer la **date de début au plus tôt** de chaque tâche (en supposant que le début des travaux ait lieu au temps 0), comme le poids du plus *long* chemin allant de `début` à cette tâche; la durée minimale des travaux est le poids maximal d'un chemin allant de `début` à `fin` (un tel chemin est dit **critique**, et tous ses sommets correspondent à des tâches critiques).

De même, on peut calculer la **date de début au plus tard** de chaque tâche, qui est la différence entre la durée minimale du processus et la durée minimale des travaux restant à accomplir (poids du plus long chemin allant de cette tâche à `fin`).

La marge totale d'une tâche est la différence entre ces deux dates.

Dans l'exemple proposé, la durée minimale des travaux est de 7 jours, et les contraintes tâche par tâche sont les suivantes :

Tâches	*Début au plus tôt*	*Début au plus tard*	*Marge totale*	*Marge libre*
Parquet salle de séjour	0	1	1	1
Réaménagement salle de bains	0	0	0	0
Carrelage salle de bains	5	5	0	0
Réfection électricité	0	1	1	0
Peinture	3	4	1	0

8.2 La méthode potentiels/étapes (PERT)

Dans la méthode PERT (*Project Evaluation and Review Technique*), on représente chaque tâche par un et un seul arc, valué par la durée de la tâche, les sommets du graphe correspondant aux étapes de la réalisation du projet. On aura ainsi, pour chaque tâche, un sommet `début-de-tâche` et un sommet `fin-de-tâche`.

Pour tenir compte des contraintes de précédence, on ajoute des tâches fictives de durée nulle, reliant la fin de la tâche contraignante au début de la tâche contrainte (ainsi qu'un état `début` au début de chaque tâche non contrainte, et la fin de chaque tâche non contraignante à un état `fin`):

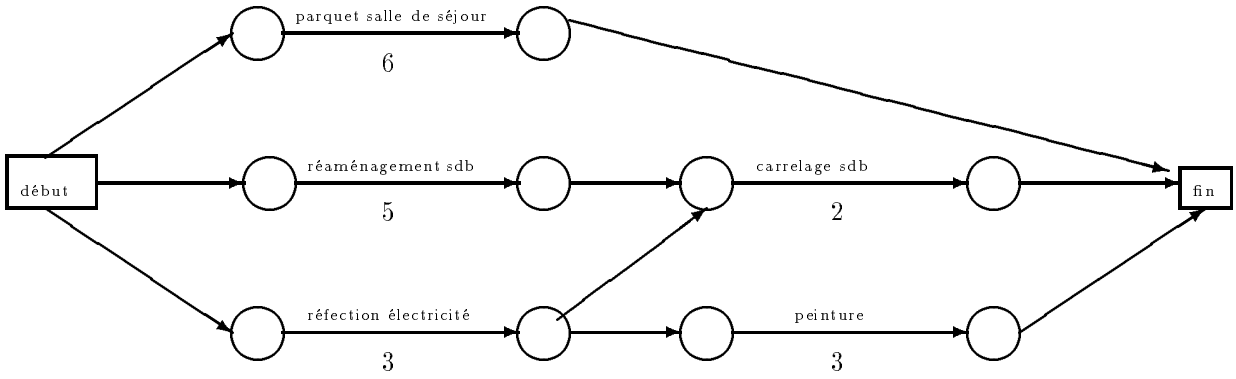


Figure 19: Modélisation PERT (les tâches fictives ne sont pas étiquetées)

Le problème est que le graphe ainsi obtenu peut être très grand, et comporter des informations inutiles; on le réduira, de façon non déterministe, en appliquant les règles suivantes :

- tout chemin $s_1 \longrightarrow s_2 \xrightarrow{T} s_3$ peut être remplacé par un arc $s_1 \xrightarrow{T} s_3$, le sommet s_2 étant supprimé, à condition qu'il ne soit le but d'aucun autre arc;
- tout chemin $s_1 \xrightarrow{T} s_2 \longrightarrow s_3$ peut être remplacé par un arc $s_1 \xrightarrow{T} s_3$, le sommet s_2 étant supprimé, à condition qu'il ne soit l'origine d'aucun autre arc.

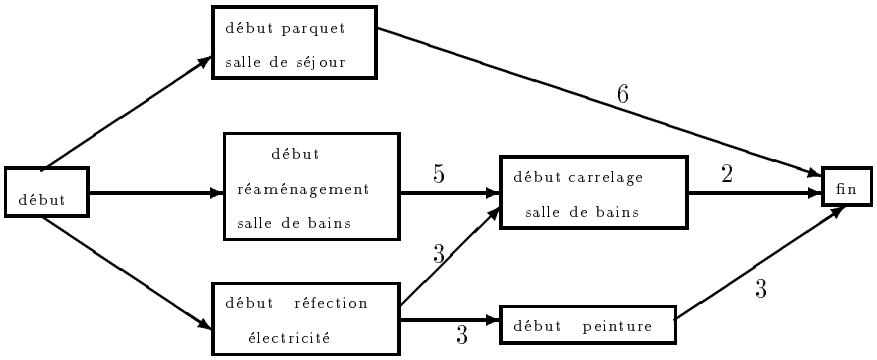


Figure 20: Modélisation PERT après réduction

Comme dans la méthode MPM, le problème d'ordonnancement n'a de solution que si le graphe obtenu est acyclique.

On calculera la **date de début au plus tôt** de chaque étape (en supposant que le début des travaux ait lieu au temps 0), comme le poids du plus *long* chemin allant de `début` à cette étape; la durée minimale des travaux est le poids maximal d'un chemin (**chemin critique**) allant de `début` à `fin`. Toute tâche située sur ce chemin est une tâche critique.

De même, on calculera la **date de fin au plus tard** de chaque étape, qui est la différence entre la durée minimale du processus et la durée minimale des travaux restant à accomplir (poids du plus long chemin allant de cette étape à `fin`).

La marge totale d'une tâche $s \longrightarrow t$ est la différence entre la date de fin au plus tard de t, et la somme de la date de début au plus tôt de s et de la durée de la tâche.

Tâches	*Début au plus tôt*	*Fin au plus tard*	*Marge totale*	*Marge libre*
Parquet salle de séjour	0	7	1	1
Réaménagement salle de bains	0	5	0	0
Carrelage salle de bains	5	7	0	0
Réfection électricité	0	4	1	0
Peinture	3	7	1	0

Une solution de problème d'ordonnancement est souvent représentée par un **diagramme de Gantt** :

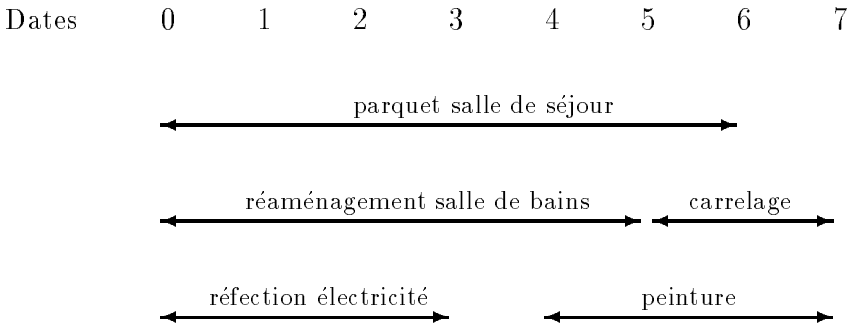


Figure 21: Diagramme de Gantt

9 Arbres recouvrants de poids minimal

Exemple de problème

Un réseau local comporte des machines A, B, C, D, et E qui doivent pouvoir communiquer entre elles (certaines machines servant éventuellement de relais). Les liaisons envisagées sont représentées par le graphe suivant (les arêtes sont étiquetées par la distance entre les machines):

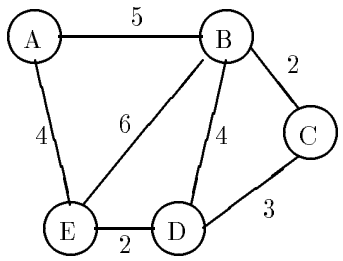


Figure 22: Réseau d'interconnexion

Comment câbler le réseau à moindre coût ?

Il s'agit d'enlever des arêtes au graphe de façon qu'il reste connexe, et que la somme des valuations des arêtes soit minimale.

Il est clair que le graphe partiel cherché est sans cycle (si on enlève une arête à un cycle d'un graphe connexe, le graphe reste connexe), c'est donc un arbre : le problème se ramène donc au

Problème de l'arbre recouvrant de poids minimal: *étant donné un graphe non orienté connexe $\mathcal{G}$ dont les arêtes sont valuées par des nombres positifs (poids), déterminer un arbre qui soit un graphe partiel de $\mathcal{G}$ (arbre* **recouvrant** *$\mathcal{G}$) de poids total minimum.*

9.1 Algorithme de Kruskal

L'algorithme de Kruskal suppose que les n arêtes $a_1, a_2, \ldots a_n$ du graphe $\mathcal{G}$ sont numérotées par ordre croissant de poids :

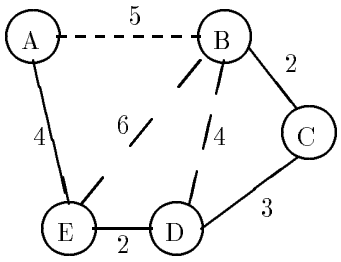
```
soit A le graphe partiel de G sans arêtes;
POUR i allant de 1 à n
     SI l'arête a_i ne crée pas de cycle dans A ALORS
          ajouter a_i à A
     FIN SI
FIN POUR
```

Preuve de l'algorithme : $\mathcal{A}$ peut s'obtenir en enlevant successivement à $\mathcal{G}$ des arêtes faisant partie d'un cycle, donc $\mathcal{A}$ est connexe; comme $\mathcal{A}$ ne comporte pas de cycle, c'est un arbre.

Soit $\mathcal{B}$ un arbre recouvrant de $\mathcal{G}$: à chaque arête a de $\mathcal{A}$, on peut associer une et une seule arête b de $\mathcal{B}$ reliant les deux composantes connexes de $\mathcal{A}$ privé de l'arête a. Mais l'arête b n'a pu être éliminée par l'algorithme de Kruskal que si elle était de poids supérieur ou égal à a (puisqu'elle ne crée pas de cycle dans $\mathcal{A}$ privé de a): le passage de $\mathcal{A}$ à $\mathcal{B}$ ne peut qu'augmenter le poids total, et par conséquent $\mathcal{A}$ est de poids minimal.

(À noter que si toutes les arêtes sont de poids différents, la solution est unique; il ne peut y avoir plusieurs solutions que si, parmi des arêtes de même poids, un choix d'élimination se propose.)

Complexité : la boucle `POUR` de l'algorithme est exécutée $|A|$ fois; le test permettant de savoir si une arête crée ou non un cycle dans $\mathcal{A}$ peut se faire en moins de $|S|$ opérations (le nombre d'arêtes de l'arbre $\mathcal{A}$ étant $|S|-1$). La complexité est donc de l'ordre de $|A| \times |S|$ (le tri des arêtes, qui peut se faire en $|A| \log |A|$ opérations, est négligeable).



Poids	Arêtes	
2	B-C	
2	D-E	
3	C-D	
4	B-D	éliminée
4	A-E	
5	A-B	éliminée
6	B-E	éliminée

Figure 23: Application de l'algorithme de Kruskal

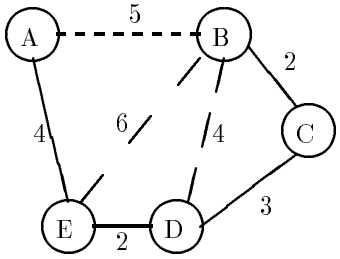
9.2 Algorithme de Sollin

```
soit A le graphe partiel de G sans arêtes;
TANT QUE A compte plus d'une composante connexe
     choisir une composante connexe C de A ;
     ajouter à A une arête de poids minimal
     reliant C à une autre composante connexe de A
FIN TANT QUE
```

Preuve de l'algorithme : à chaque passage dans la boucle, deux composantes connexes distinctes de A sont reliées, ce qui ne crée pas de cycle; l'algorithme termine (diminution d'une composante connexe à chaque passage dans la boucle), et à la fin $\mathcal{A}$ est un arbre (graphe sans cycle à une seule composante connexe).

Soit $\mathcal{B}$ un arbre recouvrant de $\mathcal{G}$: à chaque arête a de $\mathcal{A}$, on peut associer une et une seule arête b de $\mathcal{B}$ reliant les deux composantes connexes de $\mathcal{A}$ privé de l'arête a. Par construction de $\mathcal{A}$, a est de poids minimal parmi les arêtes de $\mathcal{G}$ reliant ces deux composantes, donc de poids inférieur ou égal à celui de b : cette propriété étant vraie pour toutes les arêtes, le poids total de $\mathcal{A}$ est inférieur ou égal à celui de $\mathcal{B}$.

Complexité : la boucle `TANT QUE` de l'algorithme est exécutée $|S|-1$ fois (à chaque fois, on ajoute 1 arête à $\mathcal{A}$); le maintien de la liste des composantes connexes est de l'ordre de $|S|$ (il suffit de maintenir une liste de couleurs de sommets, les sommets d'une même composante ayant la même couleur); la recherche des arêtes de poids minimal partant d'une composante connexe donnée est de l'ordre de $|A|$. La complexité est donc de l'ordre de $|S| \times \max(|S|, |A|)$.



Composantes connexes	Composante choisie	Arête choisie
A,B,C,D,E	A	A-E
AE,B,C,D	B	B-C
AE,BC,D	D	D-E
AED,BC	n'importe	C-D

Figure 24: Étapes successives de l'application de l'algorithme de Sollin

10 Recherches arborescentes

On considère ici des problèmes que l'on ne sait pas résoudre par un algorithme polynomial : d'une façon générale, il s'agit de déterminer un élément "optimal" (suivant un critère donné) dans un ensemble E fini, mais trop grand pour qu'on puisse, en pratique, examiner tous ses éléments.

Pour résoudre un tel problème, on peut utiliser le principe "diviser pour régner" : on divise l'ensemble en sous-ensembles plus petits, on résout le problème dans chacun de ces sous-ensembles, et on fait la synthèse (en comparant les éléments optimaux obtenus).

Employé récursivement, ce procédé correspond à la construction d'une arborescence dont les nœuds sont les sous-ensembles, et les feuilles, les éléments de E : l'intérêt est de pouvoir élaguer l'arborescence, en éliminant d'emblée certains nœuds, au lieu d'examiner tous les éléments de E.

Les algorithmes de **séparation et évaluation** (*Branch and Bound*) construisent l'arborescence en évaluant *a priori* les chances de trouver la solution optimale dans tel ou tel sous-ensemble.

Cette évaluation empirique donne un ordre de parcours de l'arborescence (on examine d'abord les sous-ensembles qui ont le plus de chances de contenir la solution), ce qui peut permettre d'arriver rapidement à la solution cherchée dans de nombreux cas (amélioration *en moyenne* de l'algorithme de parcours).

On peut aussi l'utiliser comme **heuristique** pour déterminer rapidement une solution approchée (c'est-à-dire une solution "acceptable", même si elle n'est pas optimale), en examinant seulement, à chaque branchement, le sous-ensemble pour lequel l'évaluation est la plus favorable.

10.1 Le problème des Reines sur l'échiquier

Comment peut-on placer n Reines sur un échiquier de dimensions $n \times n$ de façon qu'aucune ne soit en prise ?

On peut considérer le problème comme la recherche d'une solution maximale (n Reines sur l'échiquier) dans l'ensemble E des façons de placer des Reines sur les colonnes de l'échiquier sans qu'aucune ne soit en prise.

On peut alors séparer E en sous-ensembles correspondant à une même position de la Reine sur la première colonne; chacun de ces sous-ensembles peut être divisé suivant la position de la Reine de la deuxième colonne, etc.

La construction de l'arborescence en profondeur d'abord permet de trouver une solution, s'il en existe. Ce qui correspond à l'algorithme suivant, où chaque nœud de l'arborescence est étiqueté par un échiquier comportant des Reines, des cases rayées (en prise), et des cases libres :

chercher une solution :
```
créer la racine avec l'échiquier vide;
p := 0;
visiter la racine
```

visiter un nœud s **:**
```
SI p = n ALORS
     succès (fin)
SINON
     p := p + 1 ;
     POUR i allant de 1 à n
          SI la case (p, i) de l'échiquier est libre ALORS
               créer un fils t de s avec le même échiquier;
               placer une Reine sur la case (p, i);
               rayer la colonne p, la ligne i
               et les diagonales passant par (p, i);
               visiter t
          FIN SI
     FIN POUR
FIN SI;
échec
```

Lorsqu'on applique l'algorithme à la main, on peut regrouper les nœuds de l'arborescence correspondant à des positions symétriques :

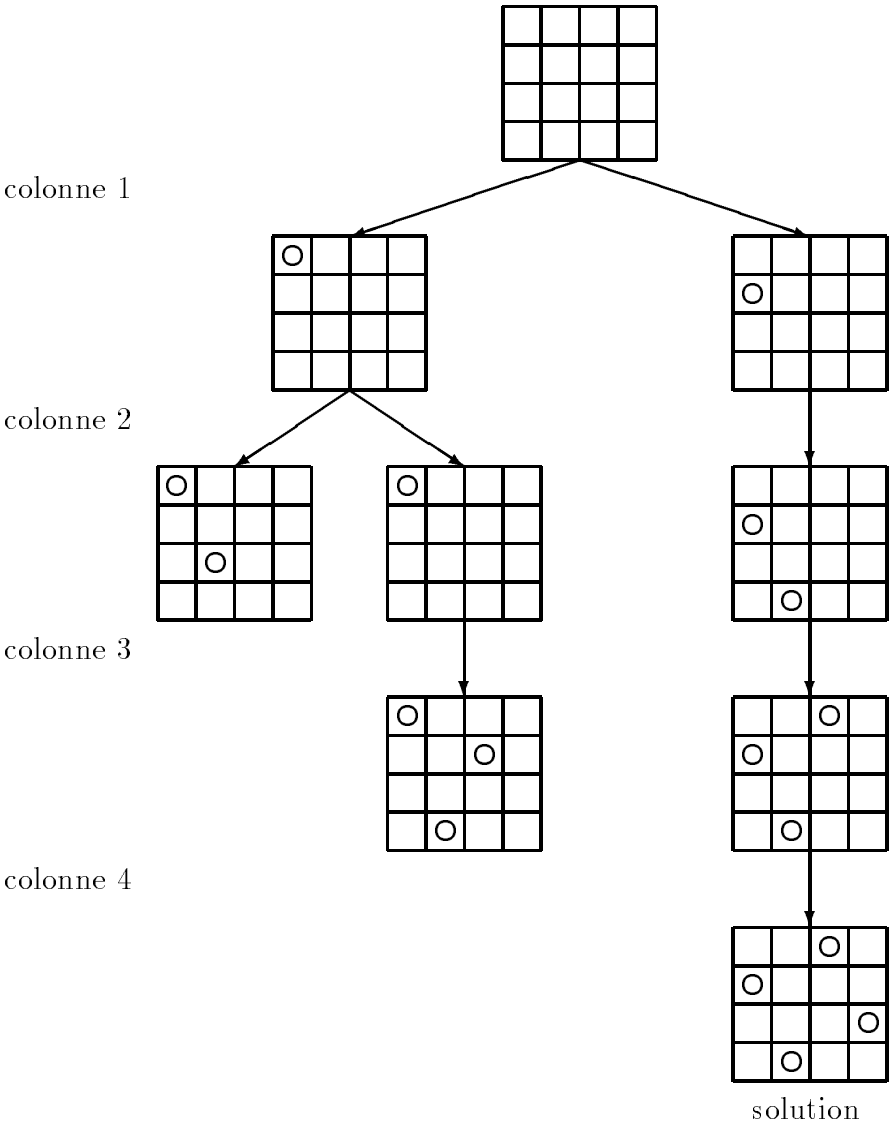
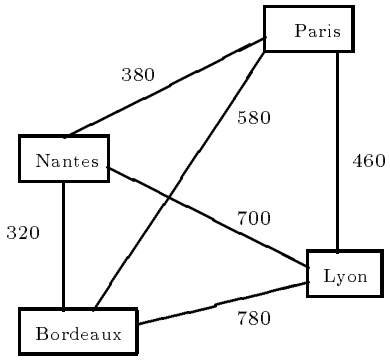


Figure 25: Arborescence des placements de Reines sur un échiquier 4×4

10.2 Le problème du voyageur de commerce

Un voyageur de commerce habitant Bordeaux doit effectuer une tournée passant par Lyon, Nantes et Paris :



Dans quel ordre devrait-il effectuer ses déplacements de façon à minimiser le trajet parcouru ?

C'est le problème classique du voyageur de commerce (*TSP : Traveling Salesman Problem*): il s'agit de déterminer dans le graphe un circuit élémentaire passant par chaque sommet (circuit **hamiltonien**), de façon que la somme des distances d'un sommet à un autre soit minimale, les distances étant données par le tableau

	Bordeaux	Lyon	Nantes	Paris
Bordeaux		780	320	580
Lyon	780		700	460
Nantes	320	700		380
Paris	580	460	380	

Le problème pourrait être résolu en examinant tous les ordonnancements possibles des sommets au départ de Bordeaux; si l'on part d'un graphe à n sommets, cela donne $n!$ possibilités, ce qui conduit à un algorithme irréalisable, en pratique.

L'algorithme de Little

L'algorithme de Little est du type "séparation et évaluation" : on effectue un parcours en profondeur de l'arborescence binaire des choix possibles, en attribuant à chaque nœud n une *évaluation par défaut* $e(n)$, et une *valeur de regret* à chaque choix auquel on renonce (évaluation par défaut du prix à payer pour renoncer à ce choix).

On remarque tout d'abord que l'on ne change pas le problème en soustrayant un nombre quelconque à une ligne ou une colonne de la matrice des distances entre les villes (cela revient à soustraire cette distance à tous les circuits hamiltoniens). On appellera **réduction** de la matrice l'opération consistant à soustraire de chaque ligne le plus petit élément, puis à soustraire le plus petit élément de chaque colonne de la matrice ainsi obtenue.

L'évaluation par défaut associée à la racine de l'arborescence est la somme des nombres soustraits lors de cette réduction.

Chaque nœud de l'arborescence correspond à un choix (fait ou abandonné).

L'évaluation par défaut associée à un nœud de type "choix fait" est la somme de l'évaluation par défaut associée au nœud père, et d'une évaluation par défaut du chemin restant à parcourir (obtenue par réduction de la matrice des liaisons encore possibles; il faut prendre soin d'éliminer les liaisons créant des circuits parasites, par exemple Nantes-Bordeaux si l'on a déjà choisi d'emprunter Bordeaux-Lyon et Lyon-Nantes).

Le regret associé à un choix abandonné est calculé de la façon suivante : si on renonce à une liaison $s \to t$, il faudra sortir de s et entrer dans t par d'autres liaisons; la valeur de regret associée à $s \to t$ est la somme du plus petit coût des liaisons $s \to s'$ et du plus petit coût des liaisons $t' \to t$ encore disponibles.

L'évaluation par défaut associée à un nœud de type "choix abandonné" est la somme du regret et de l'évaluation par défaut du nœud père.

On parcourt l'arborescence en profondeur, en ignorant les nœuds dont l'évaluation par défaut dépasse la valeur d'un circuit hamiltonien déjà trouvé.

algorithme de Little :

```
maxi := +∞ ;
s(racine) := ville de départ;
calculer e(racine);
visiter la racine
```

visiter un nœud n :

```
SI e(n) ≤ maxi ALORS
     SI s(n) ≠ ville de départ ALORS
          POUR toutes les liaisons s(n) → v possibles
               calculer le regret de s(n) → v
          FIN POUR ;
          soit s(n) → v une liaison de regret maximal r;
          créer les fils v et v̄ de n;
          s(v) := v; calculer e(v);
          visiter v;
          s(v̄) := s(n); e(v̄) := e(n) + r;
          visiter v̄
     SINON
          maxi := max(maxi, e(n))
     FIN SI
FIN SI
```

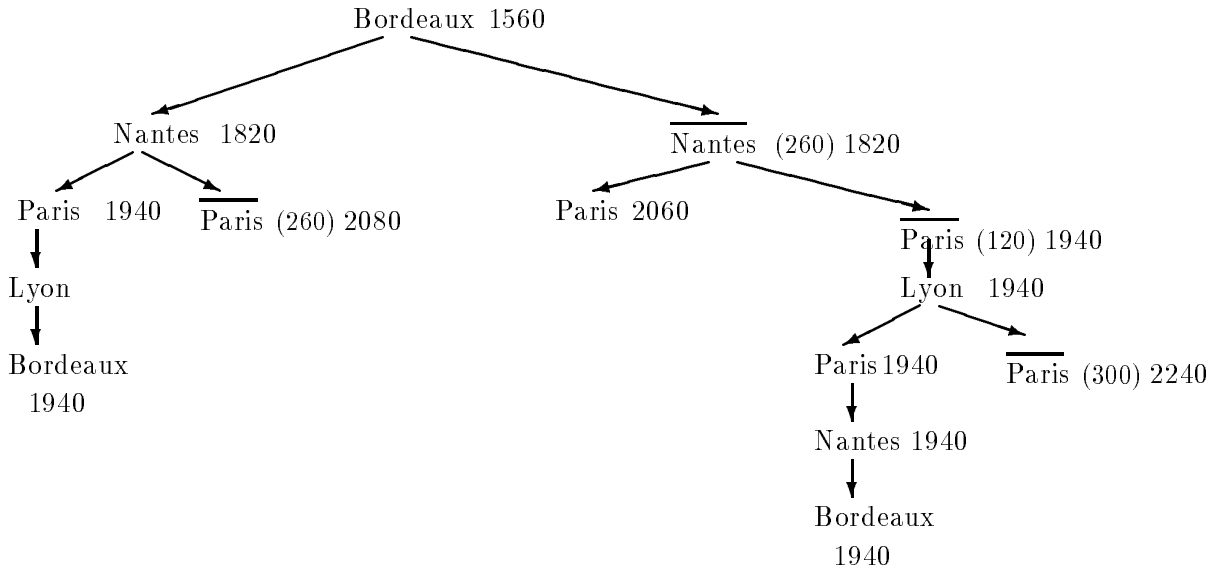


Figure 26: Application de l'algorithme de Little

Partie III
Problèmes de flots

11 Généralités sur les flots

11.1 Définition

Un **flot** dans un graphe est une valuation des arcs respectant la **loi de conservation des flux** (dite aussi **loi de Kirchhoff**): pour tout sommet s, la somme des valuations des arcs d'origine s (flux sortant de s) est égale à la somme des valuations des arcs de but s (flux entrant).

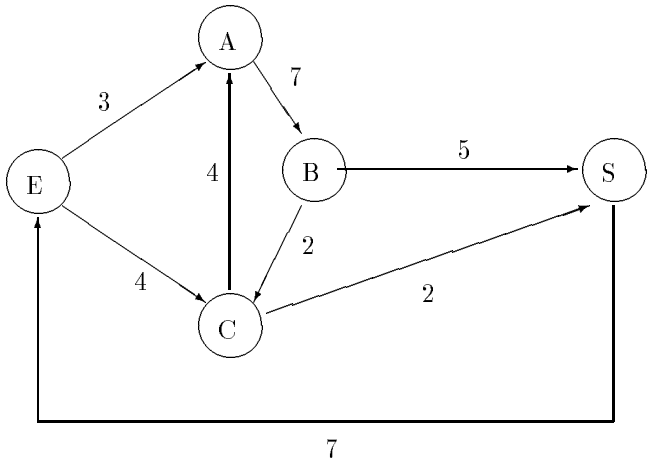


Figure 27: Exemple de flot

Les flots sont un modèle des réseaux de communication (circuits électriques, réseau de machines, réseau de transport...)

11.2 Décomposition d'un flot à valeurs positives

Théorème de décomposition : *Tout flot non nul, à valeurs positives ou nulles, peut s'écrire comme une combinaison linéaire à coefficients positifs de flots dont chacun vaut 1 sur les arcs d'un circuit du graphe, et 0 sur tout autre arc.*

Preuve : par récurrence sur le nombre d'arcs sur lesquels le flot n'est pas nul.

Soit f un flot non nul à valeurs positives ou nulles sur un graphe G, $s_0 \to s_1$ un arc de G sur lequel f n'est pas nul. Le flot entrant dans le sommet s_1 est alors strictement positif, donc le flot sortant aussi, et il existe un arc $s_1 \to s_2$ sur lequel f n'est pas nul. En répétant le raisonnement, on peut construire une suite $s_0, s_1, s_2, \ldots, s_n, \ldots$ de sommets de façon que le flot f soit strictement positif sur chacun des arcs $s_n \to s_{n+1}$.

Comme le nombre de sommets du graphe est fini, le chemin $s_0 \to s_1 \to \ldots \to s_n \to \ldots$ comporte nécessairement un circuit; soit r la plus petite valeur de f sur le circuit : alors f peut s'écrire $rc_1 + f_1$, où c_1 est le flot valant 1 sur le circuit et 0 ailleurs, et où $f_1 = f - rc_1$ est un flot à valeurs positives ou nulles, qui est nul sur strictement plus d'arcs que f.

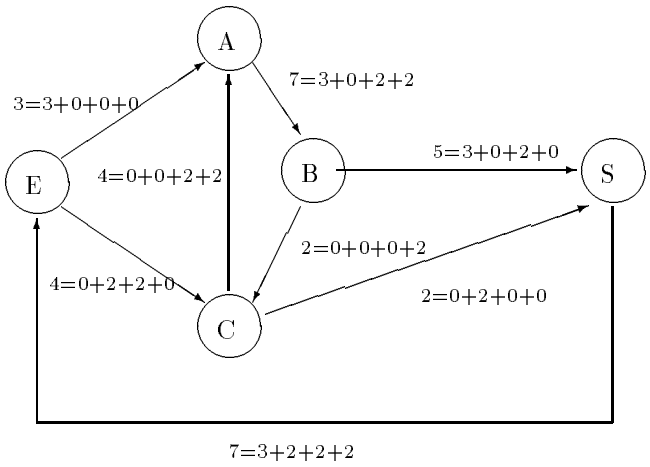


Figure 28: $f = 3c_1 + 2c_2 + 2c_3 + 2c_4$

11.3 Flots sur un graphe borné

On se donne un graphe **borné**, c'est-à-dire un graphe orienté simple antisymétrique dont les arcs sont étiquetés par des *intervalles* à bornes entières (éventuellement infinies); on notera $\min(s \to t)$ et $\max(s \to t)$ les bornes inférieure et supérieure de l'intervalle.

On suppose que le graphe comporte un sommet `entrée`, un sommet `sortie`, et un arc `sortie` $\longrightarrow$ `entrée` appelé **arc retour**, étiqueté $[-\infty, +\infty]$ (en général, cet arc n'est pas représenté dans le traitement du problème).

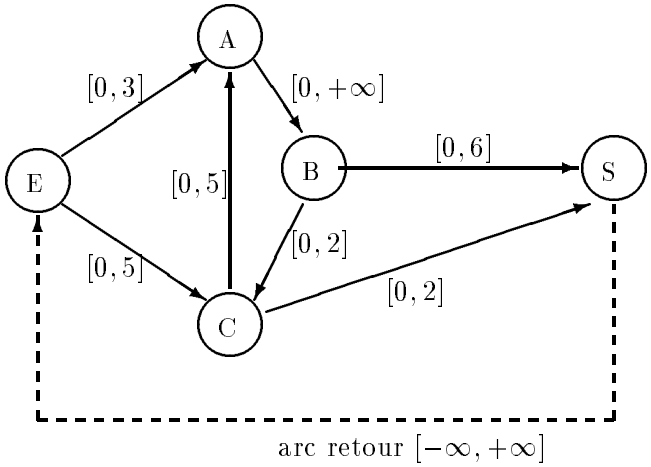


Figure 29: Graphe borné

Le problème du **flot compatible** consiste à trouver un flot pour lequel la valuation de tout arc est dans l'intervalle défini par son étiquette. On n'étudiera pas ce problème dans le cours (en pratique, dans les problèmes étudiés, la borne inférieure des intervalles sera 0, et par conséquent le flot nul sera un flot compatible).

Le problème du **flot maximal**, que l'on étudiera ici, consiste à trouver un flot compatible affectant la plus grande valeur possible à l'arc retour.

On dira qu'un arc est **saturé** par un flot si sa valuation est la borne supérieure de l'intervalle défini par son étiquette, et **antisaturé** si sa valuation est la borne inférieure de cet intervalle.

Dans le cas où tous les intervalles sont de la forme $[0, m]$, on dira que m est la **capacité** de l'arc (qui sera simplement étiqueté $[m]$). Un arc saturé est alors un arc dont le flot est sa capacité; un arc antisaturé, un arc sur lequel le flot est nul.

On dira qu'un flot est **complet** si tout chemin de l'entrée à la sortie passe par au moins un arc saturé.

Si un flot n'est pas complet, il n'est pas maximal : pour l'améliorer, il suffit de considérer un chemin de l'entrée à la sortie dont aucun arc n'est saturé, et d'augmenter les valuations de tous les arcs de façon compatible avec les étiquettes (cette technique permet d'obtenir un flot complet par améliorations successives de n'importe quel flot compatible).

Un flot maximal est donc complet. Mais un flot complet n'est pas nécessairement maximal :

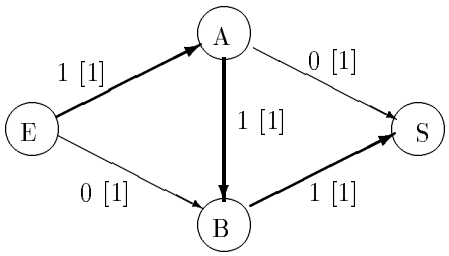


Figure 30: Exemple de flot complet non maximal

11.4 Graphe d'écart associé à un flot compatible

À un flot compatible f sur un graphe G, on associe le **graphe d'écart** G_f dont les sommets sont ceux du graphe G, mais dont les arcs sont à double sens : à tout arc $a = s \longrightarrow s'$ de G, on associe dans le graphe G_f

- si $f(a) < \max(a)$ (arc non saturé), un arc $s \longrightarrow s'$ de capacité $\max(a) - f(a)$;
- si $f(a) > \min(a)$ (arc non antisaturé), un arc $s' \longrightarrow s$ de capacité $f(a)$.

On ne représente pas les arcs du graphe d'écart dont la capacité est nulle.

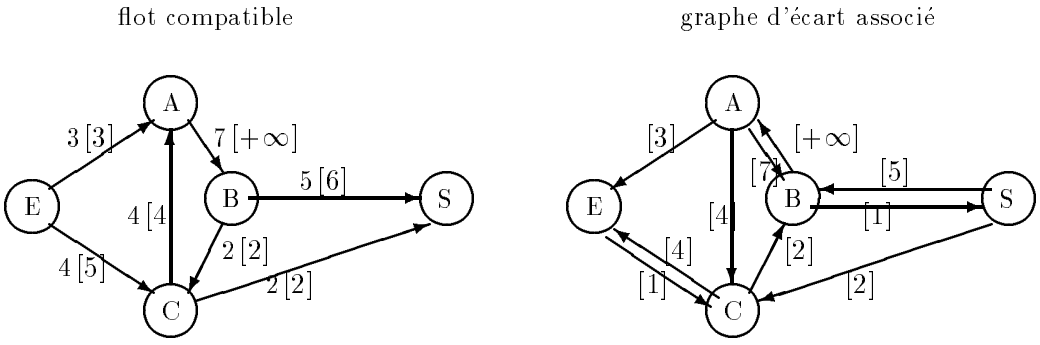


Figure 31: Graphe d'écart associé à un flot compatible

L'ajout d'une quantité d (positive ou négative) du flot sur l'arc a correspond, dans le graphe d'écart, à augmenter de d la capacité de $s' \longrightarrow s$, et à retrancher de d la capacité de $s \longrightarrow s'$; la capacité d'un arc du graphe d'écart correspond donc aux possibilités de modification du flot (augmentation dans le sens $s \longrightarrow s'$, diminution dans le sens contraire).

12 Algorithme de Ford-Fulkerson pour optimiser un flot

On suppose donné un flot compatible associant à tout arc $s \to t$ une valuation $f(s \to t)$.

L'algorithme de Ford-Fulkerson consiste à marquer les sommets de façon à déterminer si le flot est maximal, et, sinon, à indiquer une façon de l'augmenter.

12.1 Algorithme de marquage

```
marquer + le sommet entrée;
RÉPÉTER
     POUR tout sommet marqué s non encore traité
          POUR tout arc s ⟶ t
               SI t n'est pas marqué et s → t n'est pas saturé ALORS
                    marquer t par (+, s)
               FIN SI
          FIN POUR ;
          POUR tout arc s ⟶ t
               SI t n'est pas marqué et s → t n'est pas antisaturé ALORS
                    marquer t par (−, s)
               FIN SI
          FIN POUR
     FIN POUR
TANT QU'on marque de nouveaux sommets;
SI le sommet sortie n'est pas marqué ALORS
     le flot est maximal
SINON
     augmenter le flot
FIN SI
```

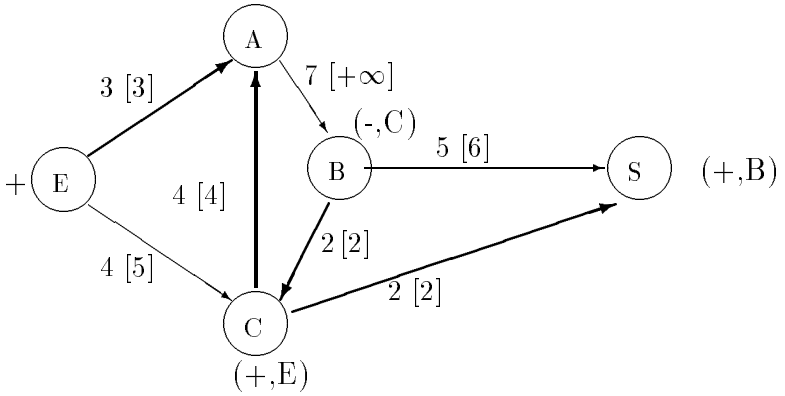


Figure 32: Marquage de Ford-Fulkerson

Complexité : il est clair que l'algorithme se termine, puisqu'on marque au plus $|S|$ sommets (on passe donc au plus $|S|$ fois par la boucle RÉPÉTER); d'autre part, on examinera au plus 2 fois chaque arc (en traitant son origine et son extrémité). La complexité est donc au pire de l'ordre de $\max(|S|, |A|)$.

12.2 Algorithme d'optimisation

calculer une augmentation possible du flot :

$a := +\infty$;
$t :=$ sortie;
TANT QUE $t \neq$ entrée
    SI t est marqué $(+, s)$
        $a := \min(a, \max(s \to t) - f(s \to t))$
    FIN SI;
    SI t est marqué $(-, s)$
        $a := \min(a, f(t \to s) - \min(t \to s))$
    FIN SI
FIN TANT QUE;

augmenter le flot :

$t :=$ sortie;
TANT QUE $t \neq$ entrée
    SI t est marqué $(+, s)$
        augmenter de a le flot sur l'arc $s \longrightarrow t$;
    FIN SI;
    SI t est marqué $(-, s)$
        diminuer de a le flot sur l'arc $t \longrightarrow s$;
    FIN SI
FIN TANT QUE

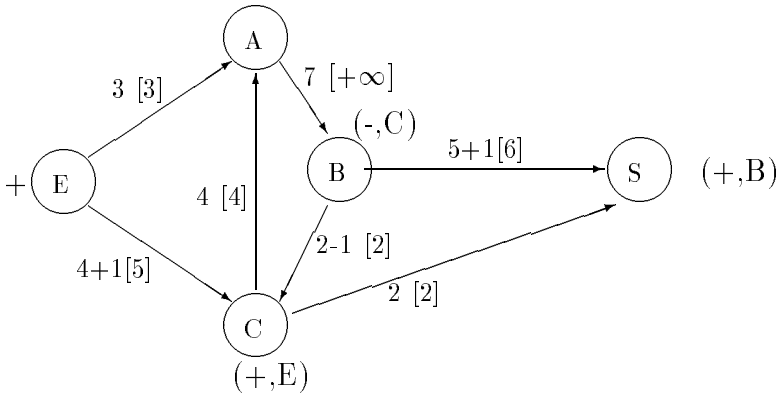


Figure 33: Augmentation du flot sur un chemin marqué de l'entrée à la sortie

Complexité : on traite deux fois de suite un chemin comportant au plus $|S|$ sommets; la complexité est donc au pire de l'ordre de $|S|$.

12.3 Preuve de l'algorithme de Ford-Fulkerson

Si la sortie est marquée à la fin de l'algorithme, il est clair que l'algorithme d'optimisation augmente le flot de la valeur a : en effet, on a respecté la loi de conservation des flux (on fait varier de la même façon le flux entrant et le flux sortant de chaque sommet) et la compatibilité du flot (l'augmentation étant le minimum des variations possibles sur les arcs du chemin).

Réciproquement, on montre que si la sortie n'est pas marquée à la fin de l'algorithme, le flot f est maximal.

En effet, soit M l'ensemble des sommets marqués, et soit N l'ensemble des sommets non marqués; la loi de conservation des flux est globalement vérifiée par l'ensemble M, c'est-à-dire que pour tout flot compatible c

$$\sum_{s \to t, s \notin M, t \in M} c(s \to t) = \sum_{s \to t, s \in M, t \notin M} c(s \to t)$$

soit encore (puisque le sommet `entrée` est dans M et le sommet `sortie` est dans N)

$$c(\texttt{arc retour}) = \sum_{s \to t, s \in M, t \in N} c(s \to t) - \sum_{\substack{s \to t, s \in N, t \in M \\ s \to t \neq \text{arc retour}}} c(s \to t)$$

et par conséquent

$$c(\texttt{arc retour}) \leq \sum_{s \to t, s \in M, t \in N} \max(s \to t) - \sum_{\substack{s \to t, s \in N, t \in M \\ s \to t \neq \text{arc retour}}} \min(s \to t)$$

Or, par définition de l'algorithme de marquage, les arcs de M vers N sont saturés, et les arcs de N vers M sont antisaturés, par le flot f : on a ainsi

$$f(\texttt{arc retour}) = \sum_{s \to t, s \in M, t \in N} \max(s \to t) - \sum_{\substack{s \to t, s \in N, t \in M \\ s \to t \neq \text{arc retour}}} \min(s \to t)$$

et par conséquent f est maximal parmi les flots compatibles.

Toute partition des sommets en deux ensembles M et N tels que l'entrée soit dans M, la sortie dans N, que tout arc de M vers N soit saturé, et que tout arc de N vers M soit antisaturé, est appelée **coupe minimale** du graphe. La démonstration précédente prouve qu'il suffit d'exhiber une coupe minimale pour montrer qu'un flot est maximal :

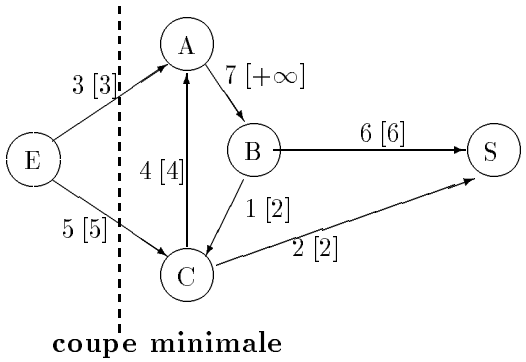
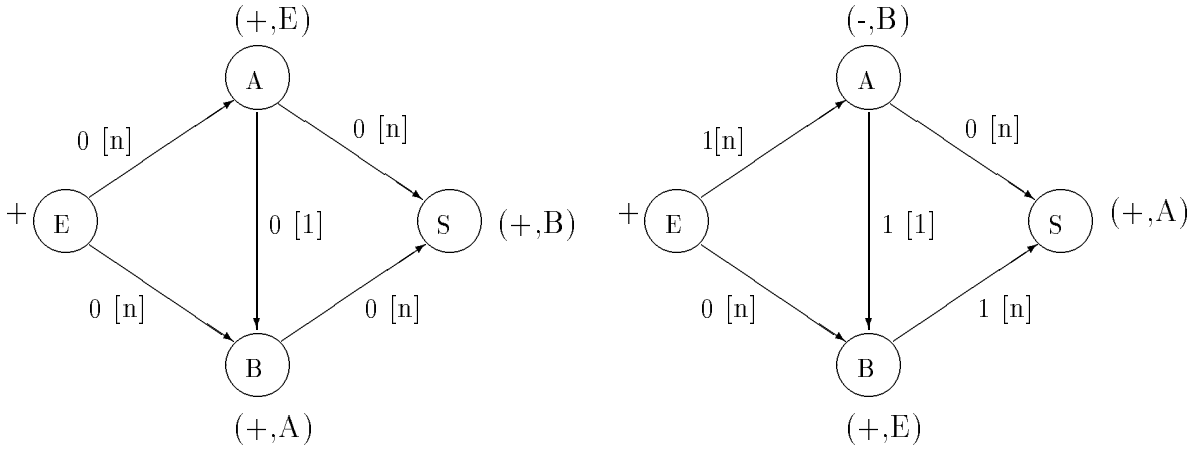


Figure 34: Flot maximal

12.4 Complexité de la méthode de Ford-Fulkerson

Pour résoudre le problème du flot maximal, il suffit d'appliquer l'algorithme d'optimisation de Ford-Fulkerson à n'importe quel flot compatible jusqu'à ce que le flot obtenu soit maximal.

Appliquée sans discernement, cette méthode n'est pas bonne, car le nombre d'optimisations nécessaires peut dépendre des bornes des intervalles. Ainsi, dans l'exemple suivant, il faut $2n$ optimisations pour atteindre le flot maximal évident ($2n$) à partir du flot nul, en saturant et en antisaturant alternativement l'arc $A \longrightarrow B$:



La méthode de Ford-Fulkerson peut cependant être améliorée si l'on impose d'augmenter le flot sur le chemin le plus court possible (en nombre d'arcs): l'ordre de complexité est alors un polynôme en $|S|$ et $|A|$, et ne dépend plus des capacités[2].

[2]Il existe des algorithmes polynomiaux plus efficaces pour optimiser un flot compatible, voir par exemple ROSEAUX : *Exercices résolus de recherche opérationnelle, t. I : Graphes.*

12.5 Interprétation dans le graphe d'écart

Soit f un flot compatible sur un graphe G, G_f le graphe d'écart associé.

Le marquage de Ford-Fulkerson correspond au marquage des sommets accessibles depuis le sommet d'entrée dans G_f (on cherche à trouver un chemin de l'entrée à la sortie):

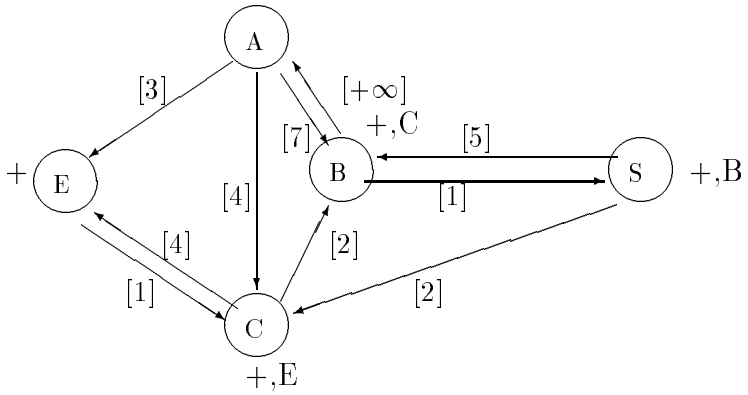


Figure 35: Marquage de Ford-Fulkerson dans le graphe d'écart

L'optimisation de Ford-Fulkerson correspond à une modification du graphe d'écart : on diminue le plus possible les capacités des arcs du chemin marqué, en augmentant d'autant les arcs réciproques. On obtient ainsi le graphe d'écart associé au flot augmenté :

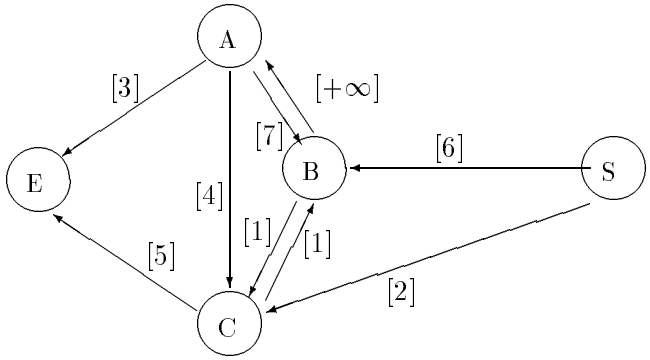


Figure 36: Optimisation de Ford-Fulkerson dans le graphe d'écart

Il est clair qu'à partir du graphe d'écart modifié, on peut retrouver le flot modifié; par conséquent, il est possible d'appliquer la méthode de Ford-Fulkerson par modifications successives du graphe d'écart associé à un flot compatible, ce qui permet d'utiliser les algorithmes classiques de parcours de graphes.

13 Affectation de coût minimal

Exemple de problème

Un édile municipal a décidé de favoriser trois entrepreneurs A, B et C susceptibles de fournir des produits x, y et z, à des prix donnés par le tableau suivant :

	Produit x	*Produit y*	*Produit z*
A	*1 MF*	*2 MF*	*3 MF*
B	*1 MF*	*4 MF*	*4 MF*
C	*1 MF*	*3 MF*	*5 MF*

Comment acheter les produits au moindre coût ?

On cherche à associer un produit différent à chaque entrepreneur de façon à minimiser la somme des coûts : il s'agit donc de choisir 3 éléments dans la matrice

1	2	3
1	4	4
1	3	5

de façon que chaque ligne et chaque colonne contienne un élément choisi, et que la somme des éléments choisis soit la plus petite possible.

D'une manière générale, un **problème d'affectation de coût minimal** est donné par un ensemble $X = \{x_1, \ldots, x_n\}$, d'un ensemble $Y = \{y_1, \ldots, y_n\}$, et une matrice M, de dimensions $n \times n$, des coûts d'affectation (si une affectation n'est pas possible, on considérera qu'elle est de coût $+\infty$); on cherche une bijection f de X sur Y telle que la somme des coûts des affectations

$$C_M(f) = \sum_{x \in X} M[x, f(x)] = \sum_{y \in Y} M[f^{-1}(y), y]$$

soit minimale.

13.1 Analyse du problème

On a nécessairement

$$C_M(f) \geq \sum_{1\leq i\leq n} \min_{1\leq j\leq n} M[i,j] \text{ et de même } C_M(f) \geq \sum_{1\leq j\leq n} \min_{1\leq i\leq n} M[i,j]$$

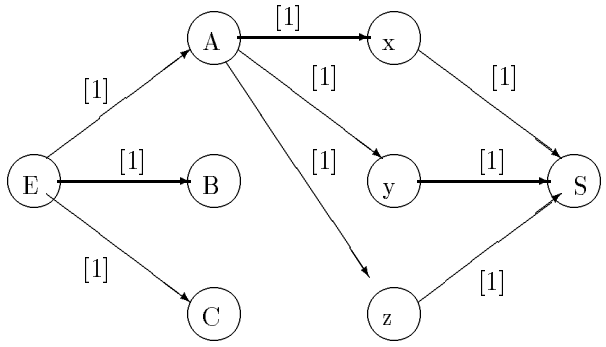
Le problème de savoir si la borne inférieure des coûts est accessible, c'est-à-dire s'il existe une affectation f telle que

$$C_M(f) = \max(\sum_{1\leq i\leq n} \min_{1\leq j\leq n} M[i,j], \sum_{1\leq j\leq n} \min_{1\leq i\leq n} M[i,j])$$

est un problème simple de flot maximum.

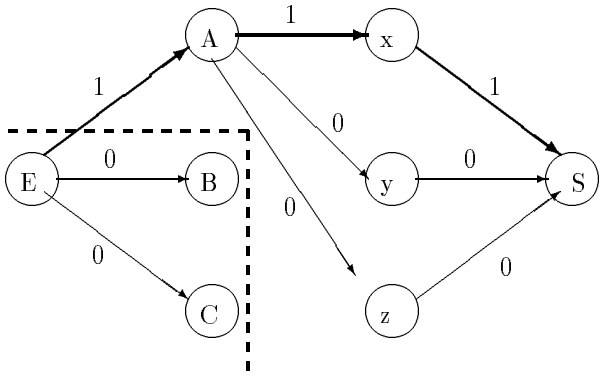
En effet, on considère un graphe dont les sommets sont les éléments de X et Y, plus 2 sommets `entrée` et `sortie`; chaque sommet de X est l'extrémité d'un arc provenant de l'entrée, chaque sommet de Y est l'origine d'un arc vers la sortie; les autres arcs du graphe sont les $x_i \rightarrow y_j$ tels que $M[i,j]$ soit minimal parmi les $M[i,k]$ (si la somme des éléments minimaux des lignes est supérieure à la somme des éléments minimaux des colonnes; dans le cas contraire, on choisira les $x_i \rightarrow y_j$ tels que $M[i,j]$ soit minimal parmi les $M[k,j]$).

	minimaux
1 2 3	1
1 4 4	1
1 3 5	1
minimaux 1 2 3	



Tous les arcs ayant pour capacité 1, il s'agit de savoir si le flot maximal traversant l'arc retour a pour valeur n. Si c'est le cas, la bijection cherchée est donnée par les arcs $x_i \rightarrow y_j$ de poids 1 dans un flot maximal.

Si ce n'est pas le cas, il est nécessaire d'ajouter des arcs au graphe. Pour choisir les arcs supplémentaires, une technique consiste à appliquer l'algorithme de marquage de Ford-Fulkerson à un flot maximal : on partitionne ainsi chacun des ensembles X et Y en sommets marqués (X_M, Y_M) et sommets non marqués (X_N, Y_N).



Pour tout flot compatible, le flot sur l'arc retour est la différence entre la somme des flots des arcs allant d'un sommet marqué vers un sommet non marqué, et la somme des flots des arcs allant d'un sommet non marqué vers un sommet marqué.

Or, le théorème de Ford-Fulkerson garantit que les arcs allant de l'entrée vers un x_i non marqué sont saturés, que les arcs allant d'un y_j marqué vers la sortie sont aussi saturés, et que les arcs allant d'un x_i non marqué vers un y_j marqué sont antisaturés. La seule façon d'augmenter le flot maximal consiste donc à ajouter des arcs allant d'un x_i marqué vers un y_j non marqué; pour minimiser le coût de l'affectation, on choisira les arcs dont le coût est le plus petit possible.

La **méthode hongroise** (algorithme de Kuhn) pour résoudre un problème d'affectation repose sur le fait qu'on ne change pas le problème en soustrayant d'une ligne (ou d'une colonne) un nombre r quelconque. En effet, si M' est la matrice ainsi transformée, on aura, pour toute affectation f, $C_M(f) = C_{M'}(f) + r$; les affectations de coût minimal seront ainsi les mêmes pour M et pour M'.

On appellera **réduction** de M l'opération consistant à soustraire de chaque ligne le plus petit élément, puis à soustraire le plus petit élément de chaque colonne de la matrice ainsi obtenue (on commencera par les colonnes si la somme des minimaux des colonnes est inférieure à la somme des minimaux des lignes).

S'il est possible de privilégier, dans la matrice réduite (dont tous les éléments sont positifs ou nuls), n zéros de façon qu'il y ait exactement un zéro privilégié par ligne et un zéro privilégié par colonne, le problème est résolu (le coût de l'affectation correspondante étant 0); sinon, on introduit des zéros supplémentaires par une technique correspondant à l'algorithme de Ford-Fulkerson.

13.2 Algorithme hongrois

```
réduire M;
TANT QU'on ne peut affecter n zéros, 1 par ligne et 1 par colonne
     affecter le plus possible de zéros
     (au plus 1 par ligne et par colonne);
     marquer toutes les lignes sans zéro affecté;
     TANT QU'on peut marquer quelque chose
          marquer les colonnes ayant un zéro non affecté dans une ligne marquée;
          marquer les lignes ayant un zéro affecté dans une colonne marquée
     FIN TANT QUE;
     soit r le plus petit nombre à colonne non marquée et ligne marquée;
     soustraire r de chaque ligne marquée;
     ajouter r à chaque colonne marquée;
     réduire M
FIN TANT QUE
```

Exemple d'application : on réduit la matrice

1	2	3
1	4	4
1	3	5

en

0	0	0
0	2	1
0	1	2

(en soustrayant les minimaux des colonnes), et on affecte le zéro du coin Nord-Ouest :

X

[0]	0	0	
0	2	1	X
0	1	2	X

Le plus petit élément d'une ligne marquée et d'une colonne non marquée est 1, on modifie le problème en

1	0	0
0	1	0
0	0	1

Il y a deux solutions possibles de coût minimal 7 :

1	[0]	0
0	1	[0]
[0]	0	1

$(A-y, B-z, C-x)$ et

1	0	[0]
[0]	1	0
0	[0]	1

$(A-z, B-x, C-y)$

Preuve de l'algorithme hongrois

La procédure de marquage des lignes et des colonnes est exactement l'algorithme de marquage de Ford-Fulkerson dans le graphe associé, appliqué à un flot maximal (correspondant à une affectation maximale de zéros de la matrice).

Retrancher la valeur m aux lignes marquées revient à ajouter au graphe des chemins de coût m allant de l'entrée à la sortie, sur lesquels le flot était nul : la modification du graphe augmente donc d'au moins 1 le flot maximal. Comme ce flot ne peut dépasser n, l'algorithme termine.

Aucune des modifications de la matrice ne change l'ensemble des solutions du problème; la solution fournie à la sortie de l'algorithme (qui est de coût 0 pour le problème modifié) est donc une affectation minimale.

Le coût d'affectation de la solution retenue est exactement la somme des éléments que l'on a soustraits à une ligne ou une colonne de matrice.

Complexité

La complexité d'une réduction de matrice est de l'ordre de n^2, de même que celle d'un marquage des lignes et des colonnes (par analogie avec le marquage de Ford-Fulkerson).

La recherche d'une affectation maximale de zéros peut se faire en n^3 (au plus n applications de l'optimisation de Ford-Fulkerson à partir d'un flot nul).

On passe au plus n fois par la boucle `TANT QUE` extérieure (puisque le flot maximal augmente d'au moins 1 unité à chaque fois). La complexité au pire est donc de l'ordre de n^4.

14 Flot maximal de coût minimal

On se donne un graphe antisymétrique $G = (S, A)$ dont les arcs sont étiquetés par des *capacités* entières positives, et par des *coûts* réels positifs ou nuls. On notera $\max(a)$ la capacité, et $c(a)$ le coût, d'un arc a.

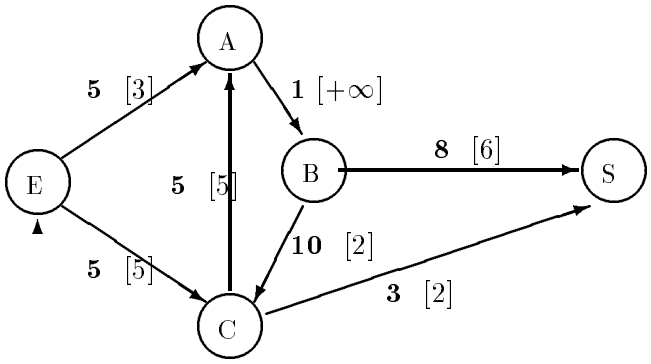


Figure 37: Graphe étiqueté pour un problème de flot maximal à coût minimal

Le problème est de déterminer, parmi les flots maximaux sur ce graphe, un flot f dont le coût total

$$c(f) = \sum_{a \in A} f(a) \times c(a)$$

soit le plus petit possible.

Il s'agit d'un problème pour lequel on ne connaît pas d'algorithme polynomial : on le traitera donc par optimisation, en améliorant progressivement une solution initiale.

Deux méthodes sont possibles :

- augmenter progressivement le flot nul, en choisissant à chaque étape la modification la plus rentable (dont le coût est le plus petit possible pour une augmentation donnée du flot); on arrête le processus lorsqu'on a obtenu un flot maximal;
- déterminer un flot maximal, et en diminuer progressivement le coût.

La première méthode est utilisée par l'**algorithme de Roy**, très proche de l'algorithme de Ford-Fulkerson (il s'agit d'ordonner le marquage des sommets de la façon la plus économique possible).

La seconde méthode est celle qu'utilise l'**algorithme de Bennington**, qui consiste à effectuer des déplacements cycliques de flot de façon à diminuer le coût.

14.1 Graphe d'écart associé à un flot à coûts

À un flot compatible f sur le graphe G, on associera le graphe d'écart G_f, dans lequel le coût d'un arc $s \to s'$ est $c(a)$ si $a = s \to s'$ est un arc de G, et $-c(a)$ si l'arc de G correspondant est $a = s' \to s$:

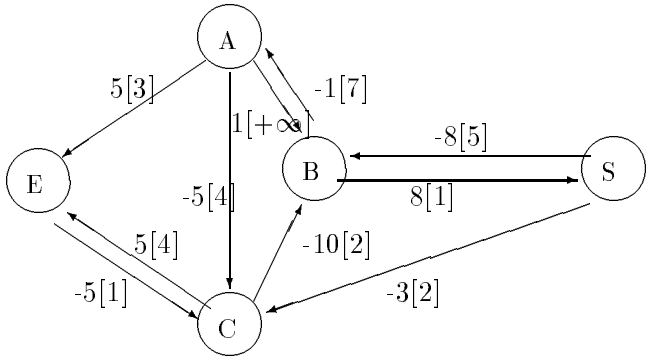


Figure 38: Graphe d'écart associé à un flot compatible à coûts

Critère d'optimisation : *Soit f un flot compatible. Il existe un flot de même valeur que f sur l'arc retour, et de coût strictement plus petit que celui de f, si et seulement s'il existe, dans le graphe d'écart G_f, un circuit dont la somme des coûts des arcs est strictement négative.*

Preuve :

Si : s'il existe un circuit de coût unitaire négatif $-c$ dans G_f, de capacité minimale r, il suffit d'augmenter le flot f de r sur ce circuit pour obtenir un flot de même valeur sur l'arc retour, et dont le coût sera celui de f, diminué de $r \times c$.

Seulement si : soit f' un flot de même valeur que f sur l'arc retour, de coût strictement plus petit que celui de f. Soit d le flot défini sur G_f par

$$d(s \to s') = \begin{cases} f'(s \to s') - f(s \to s') \text{ si } s \to s' \in G, f(s \to s') < f'(s \to s') \\ f(s' \to s) - f'(s' \to s) \text{ si } s' \to s \in G, f(s' \to s) > f'(s' \to s) \end{cases}$$

d est un flot à valeurs positives : d'après le théorème de décomposition, d peut s'écrire $\sum a_i f_i$ où les a_i sont des coefficients positifs, et où chaque c_i est un flot valant 1 sur un circuit de G_f et 0 ailleurs. Le coût de d est alors

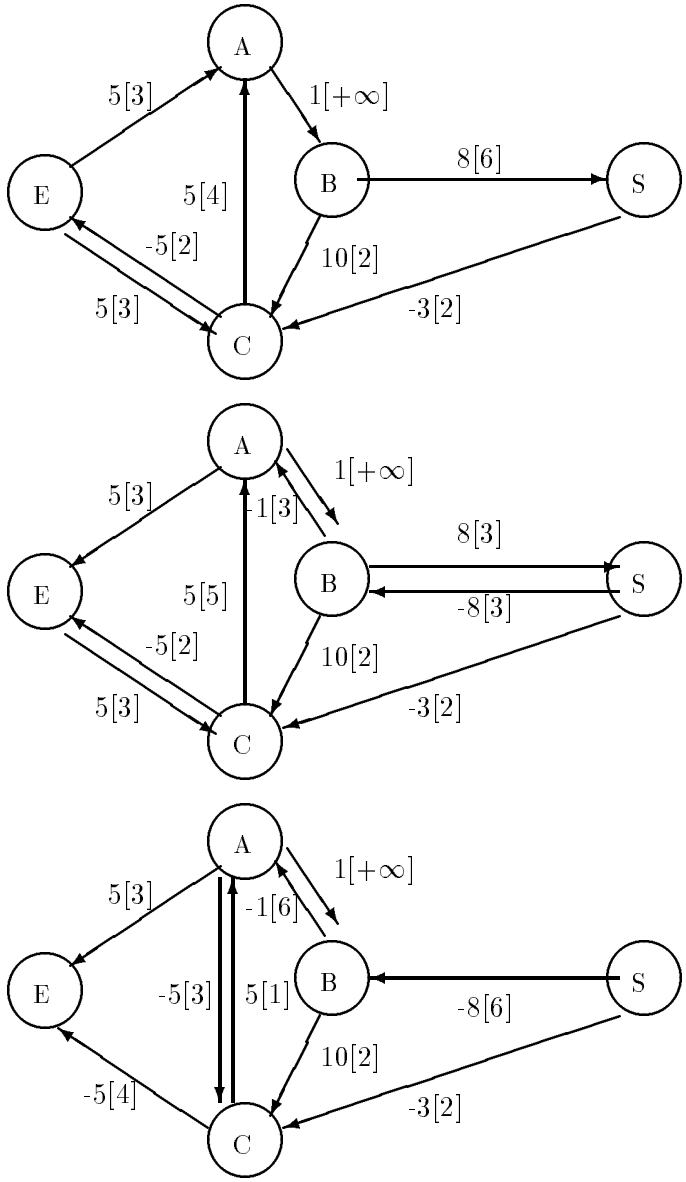
$$c(f') - c(f) = \sum a_i c(f_i)$$

et par conséquent au moins l'un des $c(f_i)$ est négatif.

14.2 Algorithme de Roy

Un flot f sur le graphe G peut être augmenté si et seulement s'il existe un chemin du point d'entrée au point de sortie dans le graphe d'écart G_f (c'est une formulation du théorème de Ford-Fulkerson); si c'est le cas, on peut augmenter le flot de de la plus petite des capacités figurant sur ce chemin (de la façon indiquée par l'algorithme de Ford-Fulkerson).

L'algorithme de Roy consiste à choisir, parmi les chemins de l'entrée à la sortie de G_f, celui dont le coût unitaire (somme des coûts des arcs) est minimal, et à augmenter le flot sur ce chemin; on répète le processus jusqu'à l'obtention d'un flot maximal.



augmentation de 2
sur le chemin ECS (coût 8)

augmentation de 3
sur le chemin EABS (coût 14)

augmentation de 3
sur le chemin ECABS (coût 19)

Figure 39: Étapes successives de l'algorithme de Roy

Preuve de l'algorithme de Roy

L'algorithme termine, puisqu'à chaque étape on augmente le flot, jusqu'à ce que l'on ait atteint le flot maximal (c'est la méthode de Ford-Fulkerson, dans laquelle on force l'ordre des optimisations).

On montre par récurrence qu'à chaque étape, on obtient un flot de coût minimal parmi les flots de même valeur sur l'arc retour : la propriété est évidente pour le flot de départ (de coût 0).

Supposons-la vraie pour un flot f obtenu à une étape : d'après le critère de minimalité, il n'existe pas de circuit de coût négatif dans le graphe d'écart G_f. L'optimisation effectuée consiste à augmenter le flot sur un circuit C de G_f passant par l'arc retour, de coût unitaire minimal parmi ces circuits; on obtient ainsi un flot f'.

Soit C' un circuit de $G_{f'}$: les arcs de C' qui ne sont pas des arcs de G_f sont nécessairement des arcs dont les opposés sont dans G_f, et par conséquent $C + C'$ est une somme de circuits élémentaires de G_f (donc de coût unitaire positif ou nul), dont l'un au moins passe par l'arc retour (donc a un coût unitaire supérieur ou égal à celui de C). Il s'ensuit que le coût unitaire de C', différence entre le coût unitaire de $C + C'$ et le coût unitaire de C, est positif ou nul : d'après le critère de minimalité, le flot f' est minimal parmi les flots de même valeur sur l'arc retour.

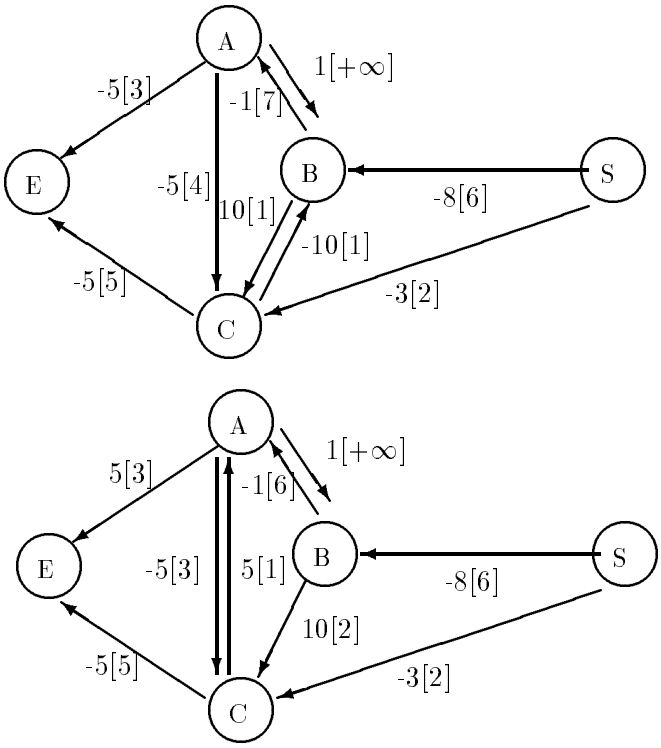
Par conséquent, le flot obtenu à la fin de l'algorithme est bien un flot de coût minimal parmi les flots maximaux.

Complexité

Pour déterminer le chemin de coût minimal sur lequel augmenter le flot, on ne peut pas appliquer un algorithme rapide de plus court chemin (en raison des coûts négatifs de certains arcs): on doit au contraire calculer les chemins acycliques du graphe d'écart, ce qui dans le pire des cas donne une complexité de l'ordre de $|S|!$

14.3 Algorithme de Bennington

L'algorithme de Bennington consiste à choisir, parmi les circuits élémentaires de coût unitaire négatif du graphe G_f, celui dont le coût unitaire est minimal (maximal en valeur absolue), et à augmenter le flot de la plus petite des capacités figurant sur ce circuit; on part d'un flot maximal, et on réitère le procédé jusqu'à obtention d'un flot de coût minimum :



flot maximal de coût 121

on augmente le flot de 1
sur le circuit ACBA de coût -16 :
on obtient un flot de coût 105 (minimal)

Figure 40: Application de l'algorithme de Bennington

Preuve de l'algorithme de Bennington

L'algorithme termine : en effet, il n'existe qu'un nombre fini de circuits élémentaires possibles, et, à chaque étape, on diminue le coût du flot d'une valeur au moins égale au plus petit des coûts unitaires (en valeur absolue) de ces circuits.

Les flots obtenus à chaque étape sont des flots maximaux. Le graphe d'écart obtenu à la fin de l'algorithme ne comporte pas de circuit de coût négatif; par conséquent, d'après le critère de minimalité, le flot obtenu est un flot de coût minimal parmi les flots maximaux.

On n'étudiera pas ici la **complexité** de l'algorithme de Bennington (due à la nécessité de calculer les circuits élémentaires du graphe d'écart).

15 Problème de transport

Exemple de problème

Une organisation terroriste dispose, chez ses militants de Nantes, La Rochelle, Besançon et Strasbourg, de stocks d'explosifs de respectivement 120 kg, 100 kg, 100 kg et 100 kg, pour lesquels elle a reçu des commandes de ses antennes de Bordeaux (80 kg), Paris (190 kg) et Lyon (150 kg).

Les coûts de transport d'un kilo d'explosifs, suivant les liaisons routières considérées, sont donnés par le tableau suivant :

	Bordeaux	Paris	Lyon
Nantes	30 F	20 F	
La Rochelle	40 F	10 F	
Besançon		40 F	80 F
Strasbourg		40 F	80 F

Comment répartir les envois de façon à satisfaire les demandes, tout en minimisant le coût du transport ?

D'une manière générale, un **problème de transport** est la donnée de $m + n$ quantités positives $l_1, l_2, \ldots, l_m, c_1, c_2, \ldots, c_n$ et de $m \times n$ nombres $c(i, j) \in [0, +\infty]$ (coûts unitaires des liaisons, un coût infini correspondant à une liaison inexistante).

On représente généralement les données du problème par la matrice à m lignes et n colonnes des coûts, augmentée d'une colonne (pour indiquer ligne à ligne les quantités l_i) et d'une ligne (pour indiquer les quantités c_j):

30	20	$+\infty$	120
40	10	$+\infty$	100
$+\infty$	40	80	100
$+\infty$	40	80	100
80	190	150	

Figure 41: Données d'un problème de transport

Une **solution** f du problème est une fonction qui à tout i et tout j associe une quantité $f(i,j)$ de façon à satisfaire les $m+n$ équations

$$l_i = \sum_{j=1}^{j=n} f(i,j) \quad \text{et} \quad c_j = \sum_{i=1}^{j=m} f(i,j)$$

On supposera désormais que des solutions existent, c'est-à-dire que la somme des quantités demandées est égale à la somme des quantités disponibles :

$$\sum_{1 \le i \le m} l_i = \sum_{1 \le j \le n} c_j$$

Le problème consiste à trouver une solution au moindre coût.

Le problème de transport équivaut à chercher un flot maximal de coût minimal dans un graphe admettant m sommets L_i, n sommets C_j, un point d'entrée E et un point de sortie S, et dont les arcs sont

- les $E \to L_i$, de capacité l_i et de coût 0;
- les $C_j \to S$, de capacité c_j et de coût 0;
- les $L_i \to C_j$, de capacité $+\infty$ et de coût $c(i,j)$.

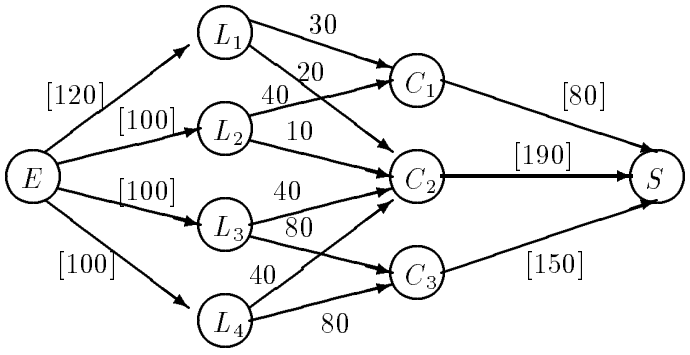


Figure 42: Interprétation d'un problème de transport comme un problème de flot

À toute solution, on peut associer le graphe non orienté dont les sommets sont les L_i et les C_j, et dont les arêtes sont les liaisons utilisées (correspondant aux $f(i,j) \neq 0$).

On dira que la solution est une **solution de base** si le graphe associé est acyclique, et une **solution non dégénérée** si le graphe est connexe. Une solution de base non dégénérée est donc une solution associée à un arbre, c'est-à-dire comportant exactement $n + m - 1$ valeurs non nulles (une solution est une solution de base si elle comporte au plus $n + m - 1$ valeurs non nulles).

On remarquera qu'à toute solution, on peut associer une solution de base de coût équivalent (ou moindre) : en effet, s'il existe un cycle dans le graphe associé, il correspond à deux circuits en sens inverse dans le graphe d'écart associé au flot correspondant; l'un de ces circuits est de coût négatif ou nul, et on peut donc "couper le circuit" en modifiant le flot sur le cycle correspondant, sans augmenter le coût.

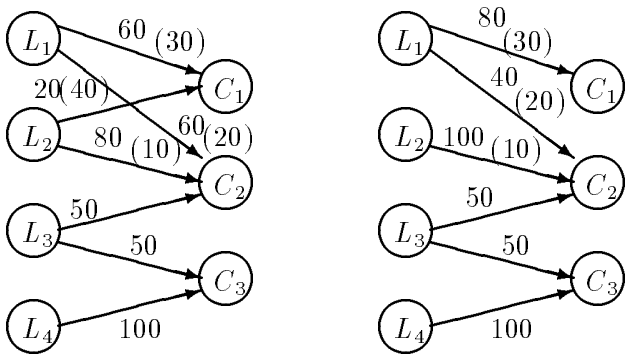


Figure 43: Une solution et la solution de base associée

On peut résoudre le problème de transport en diminuant progressivement le coût d'une solution de base.

15.1 Algorithme du marchepied (*stepping-stone*)

L'algorithme du marchepied (ou **méthode des potentiels**) consiste à améliorer une solution de base non dégénérée, en déterminant celle des liaisons non utilisées qui permettraient de diminuer le plus le coût global. Il s'agit d'une simplification de l'algorithme de Bennington : l'ajout d'une arête à l'arbre associé à la solution crée un et un seul cycle, et on cherche si le circuit associé dans le graphe d'écart est de coût négatif.

Il suffit pour cela d'associer à chaque nœud n de l'arbre un *potentiel* $V(n)$; on fixe arbitrairement à 0 le potentiel d'une des lignes; pour chaque arête de L_i à C_j, les potentiels seront liés par la relation

$$V(C_j) - V(L_i) = c(i,j)$$

ce qui permet de calculer de proche en proche le potentiel de chacun des nœuds.

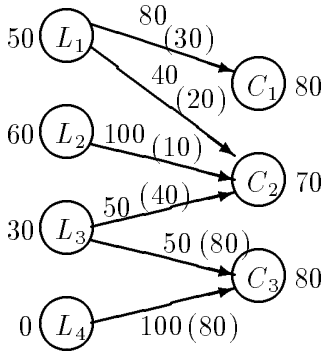


Figure 44: Potentiels associés à une solution de base

On examine ensuite chacune des liaisons non utilisées : si toutes vérifient la relation $c(i,j) \geq V(C_j) - V(L_i)$, la solution est de coût minimal; sinon, on ajoute à l'arbre une arête de L_i à C_j pour laquelle la diminution de tension $c(i,j) + V(L_i) - V(C_j)$ serait maximale, et on modifie le flot sur le circuit créé par cette arête, de façon à couper ce circuit.

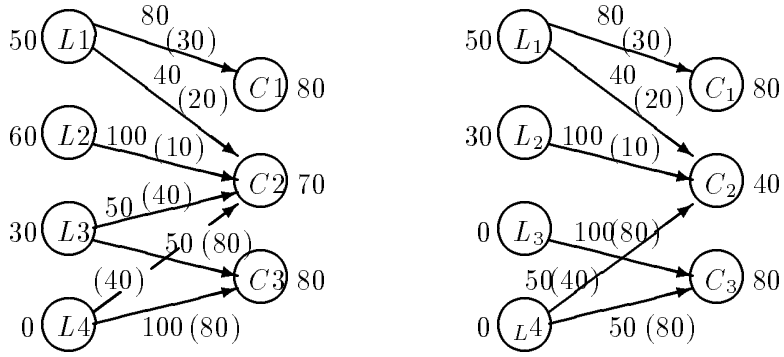


Figure 45: Application de l'algorithme du marchepied

Bien que cet algorithme ne s'applique en théorie qu'aux arbres (solutions de base non dégénérées), il s'adapte facilement aux forêts (graphes acycliques non connexes): il suffit d'ajouter des arêtes L_i — C_j de façon à obtenir un arbre, correspondant à des transports de quantités infinitésimales $f(i,j) = \varepsilon$ que l'on met à zéro dans la solution définitive.

15.2 Obtention d'une solution de base

Algorithmes gloutons

La **méthode du coin Nord-Ouest** consiste à affecter la valeur maximale possible à la liaison L_1 — C_1, puis, dans l'ordre, aux liaisons L_1 — C_2, L_1 — C_3, ... jusqu'à épuisement de la quantité disponible en L_1 (ou aux liaisons L_2 — C_1, L_3 — C_1,..., si la quantité disponible en L_1 était inférieure à la quantité demandée en C_1), et ainsi de suite. D'une façon générale, chaque fois qu'une affectation a épuisé la quantité correspondant à une ligne ou une colonne, on cherche à épuiser le plus rapidement possible la quantité de la colonne ou de la ligne qui reste.

Cet algorithme permet d'obtenir très rapidement une solution (on examine au plus $m+n-1$ liaisons), mais le coût de cette solution peut être très élevé.

80	40		120
	100		100
	50	50	100
		100	100
80	190	150	

Figure 46: Application de la méthode du coin Nord-Ouest

Une version un peu plus sophistiquée consiste à épuiser les lignes et les colonnes en affectant à chaque fois la valeur maximale possible à la liaison de coût minimal de la ligne ou de la colonne que l'on est en train d'épuiser.

Cet algorithme est plus coûteux du fait des comparaisons (de l'ordre de $m \times n$), et, en pratique, n'est pas efficace.

Méthode de Balas-Hammer

Cette méthode consiste à chercher les liaisons critiques (dont le rejet conduirait à une affectation de coût très élevé).

Pour chaque ligne et chaque colonne non encore épuisées, on calcule la différence (**regret**) entre le coût minimal des liaisons encore disponibles et le coût immédiatement supérieur; on affecte alors la plus grande valeur possible à la liaison de coût minimal de la ligne (ou de la colonne) pour laquelle le regret est le plus élevé.

Calcul des regrets :

30	20	$+\infty$	regret 10
40	10	$+\infty$	regret 30
$+\infty$	40	80	regret 40
$+\infty$	40	80	regret 40
regret 10	regret 10	regret 0	

On affecte la valeur la plus grande possible, soit 30, à la liaison L_3C_2 (en effet, on a au plus 80 sur L_4C_3, donc au moins 70 sur L_3C_3). Les affectations $L_3C_3 : 70$, $L_4C_3 : 80$ et $L_4C_2 : 20$ sont obligatoires.

On calcule les regrets sur les liaisons restantes :

30	20	regret 10
40	10	regret 30
regret 10	regret 10	

On affecte la valeur la plus grande possible, soit 100, à la liaison L_2C_2, ce qui conduit à la solution de base

80	40		120
	100		100
	30	70	100
	20	80	100
80	190	150	

Figure 47: Application de la méthode de Balas-Hammer

La complexité est de l'ordre de $m \times n$; cet algorithme est en moyenne plus efficace que les algorithmes gloutons.

Partie IV
Programmation linéaire

16 Position du problème

Exemple

Un agriculteur peut utiliser 2 types d'engrais X et Y, pour fertiliser ses terres : il sait que celles-ci requièrent, par hectare et par an, au moins 60 kg de potasse, 120 kg de calcium et 90 kg de nitrates.

Les deux types d'engrais coûtent le même prix au poids; pour 10 kg ils contiennent, en plus d'un produit neutre :

pour X **:** *1 kg de potasse, 3 kg de calcium et 3 kg de nitrates;*

pour Y **:** *2 kg de potasse, 2 kg de calcium et 1 kg de nitrates.*

Comment l'agriculteur peut-il fertiliser ses cultures au moindre coût ?

Il s'agit d'un problème à 2 inconnues x et y (nombres positifs), représentant les quantités d'engrais de types X et Y à acheter par hectare et par an.

Les contraintes imposées sur la fertilisation d'un hectare peuvent s'écrire

$$\begin{cases} x + 2y \geq 60 \\ 3x + 2y \geq 120 \\ 3x + y \geq 90 \end{cases}$$

Le problème consiste à trouver x et y de façon à minimiser $x+y$ (le coût étant proportionnel à la quantité d'engrais achetée par hectare et par an).

D'une manière générale, un problème de **programmation linéaire** simple consiste à déterminer les valeurs numériques de m variables $x_1, x_2, \ldots, x_m$ satisfaisant n contraintes

$$a_i^1 x_1 + a_i^2 x_2 + \ldots + a_i^m x_m \leq c_i \quad (1 \leq i \leq n)$$

(où les a_i^j sont des constantes, c'est-à-dire que $a_i^1 x_1 + a_i^2 x_2 + \ldots + a_i^m x_m$ est une fonction *linéaire* des variables)
de façon à minimiser (ou à maximiser) une quantité $Z = z_1 x_1 + z_2 x_2 + \ldots z_m x_m$ (donc encore une fonction linéaire des variables).

17 Interprétation géométrique

Dans un espace de dimension m, $a_i^1 x_1 + a_i^2 x_2 + \ldots + a_i^m x_m = c_i$ est une équation d'un **hyperplan** (espace de dimension $m-1$: droite si $m=1$, plan si $m=2$...) séparant l'espace en deux demi-espaces caractérisés par les relations $a_i^1 x_1 + a_i^2 x_2 + \ldots + a_i^m x_m \geq c_i$ et $a_i^1 x_1 + a_i^2 x_2 + \ldots + a_i^m x_m < c_i$.

L'ensemble des solutions $(x_1, x_2, \ldots, x_m)$ vérifiant les contraintes est ainsi l'ensemble des coordonnées des points d'un **polyèdre convexe** (s'il contient deux points, il contient le segment de droite qui les joint).

Les ensembles de points d'équations $z_1 x_1 + z_2 x_2 + \ldots + z_m x_m = Z$ sont des hyperplans tous parallèles à une certaine direction; le problème consiste donc à trouver la valeur extrémale de Z pour laquelle l'hyperplan d'équation $z_1 x_1 + z_2 x_2 + \ldots + z_m x_m = Z$ coupe le polyèdre des contraintes, et les solutions correspondant à cette valeur de Z, c'est-à-dire les points d'intersection de l'hyperplan et du polyèdre.

On peut montrer que, du fait que le polyèdre est convexe, cette intersection est une partie de la surface du polyèdre : soit un sommet, soit (cas dégénéré) une arête ou une face du polyèdre (en dimension $n=3$; ce qui se généralise facilement à n quelconque).

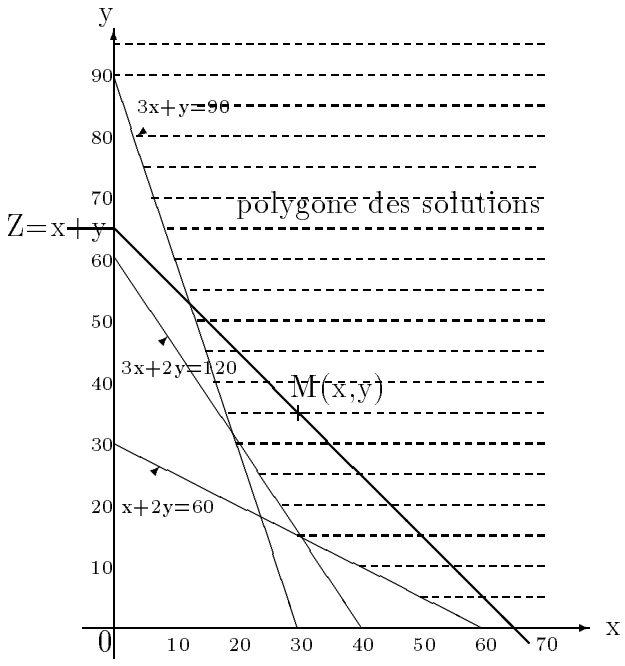


Figure 48: Interprétation géométrique d'un problème de programmation linéaire

18 Résolution graphique

Lorsque le problème ne comporte que 2 équations, il se prête à une résolution graphique : il suffit de chercher la position de la droite $Z = z_1x_1 + z_2x_2$ pour laquelle Z a une valeur extrémale, et le sommet (ou l'arête) du polygone des solutions pour laquelle cette valeur est atteinte.

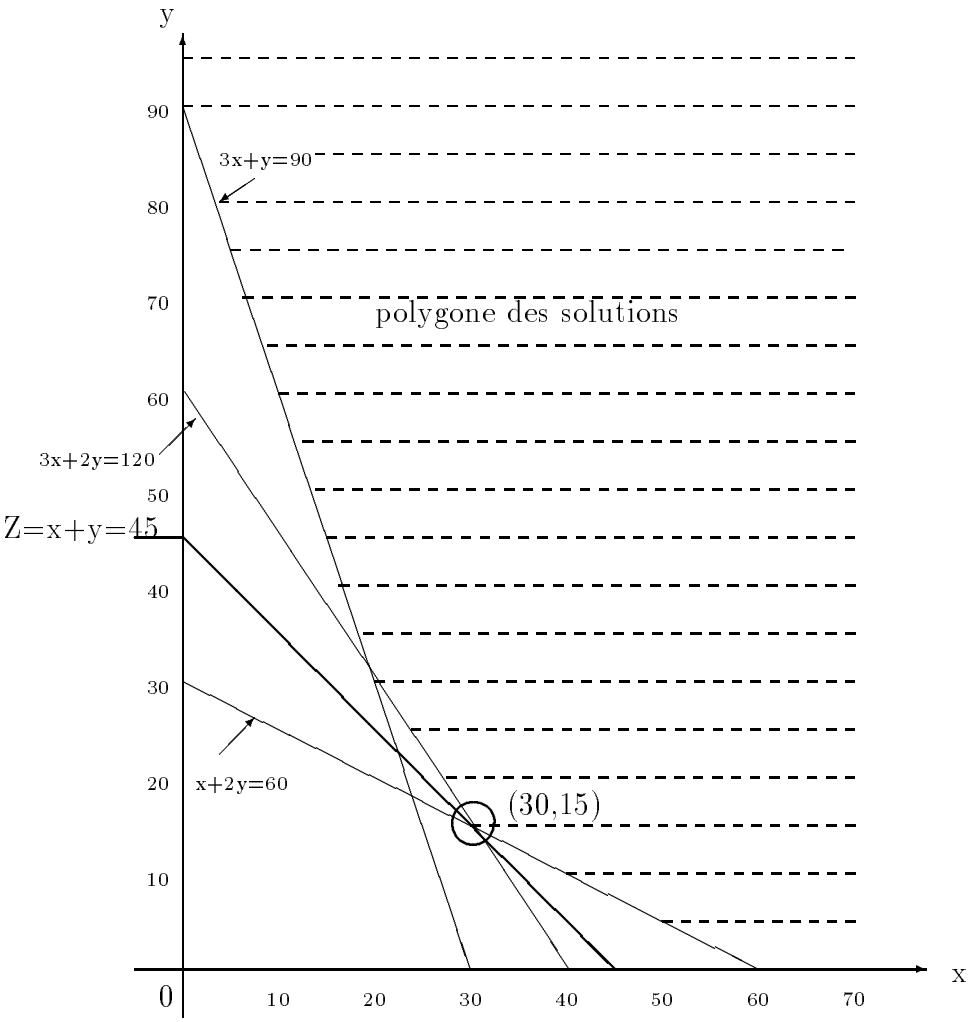


Figure 49: Résolution graphique d'un problème de programmation linéaire

19 Algorithme du simplexe

Le principe de l'algorithme du simplexe consiste à améliorer progressivement une solution de base non dégénérée (correspondant à l'un des sommets du polyèdre), en se déplaçant le long des arêtes.

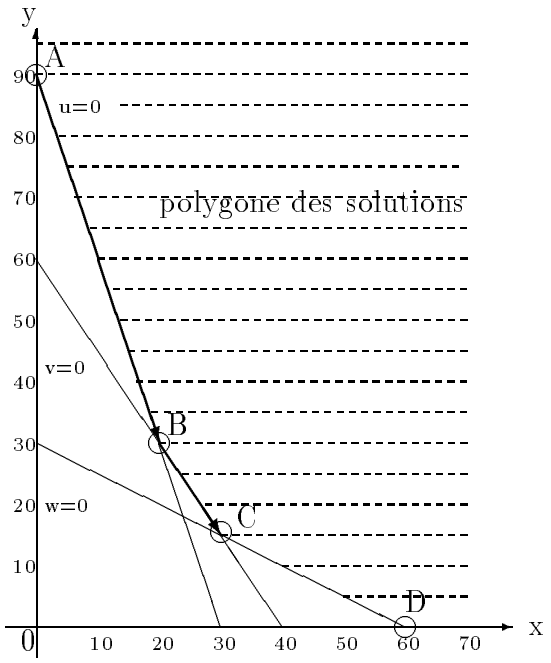


Figure 50: Algorithme du simplexe

Dans l'exemple ci-dessus, on peut partir de la solution correspondant au point A correspondant à $x = 0$, $3x + y = 90$. En posant $u = 3x + y - 90$, on peut exprimer Z sous la forme $Z = u - 2x + 90$, et les contraintes sous la forme

$$\begin{cases} x, u \geq 0 \\ 5x - 2u \leq 120 \\ 3x - 2u \leq 60 \end{cases}$$

On peut faire diminuer Z en gardant u à zéro, et en augmentant x de 20 (quantité maximale respectant les contraintes): on obtient ainsi la solution B, correspondant à $3x - 2u = 60$.

En posant $v = -3x + 2u + 60$, on peut exprimer Z sous la forme $-\frac{1}{3}u + \frac{2}{3}v + 50$, et les contraintes sous la forme $u, v \geq 0$, $\frac{4}{3}u - \frac{5}{3}v \leq 20$. On peut faire diminuer Z en gardant v à zéro, et en augmentant u de la valeur maximale respectant les contraintes, c'est-à-dire de 15 : on obtient ainsi la solution C, correspondant à $\frac{4}{3}u - \frac{5}{3}v = 20$.

En posant $w = -\frac{4}{3}u + \frac{5}{3}v + 20$, on peut exprimer Z sous la forme $\frac{1}{4}v + \frac{1}{4}w + 45$, et les contraintes sous la forme $v, w \geq 0$. On ne peut faire diminuer Z; le point C correspond donc à la solution optimale.

D'une manière générale, soit un problème de programmation linéaire à m variables $x_1, x_2, \ldots, x_m$, que l'on suppose contraintes à vérifier $x_1, x_2, \ldots, x_m \geq 0$ (on peut toujours se ramener à ce cas), avec n inéquations de la forme

$$\sum_{j=1}^{j=m} a_i^j x_j \leq c_i$$

(pour $1 \leq i \leq n$), et une fonction économique à maximiser

$$Z = \sum_{j=1}^{j=m} z_j x_j + c$$

En introduisant n **variables d'écart** $x_{m+1}, x_{m+2}, \ldots, x_{m+n}$, on peut remplacer chaque contrainte

$$\sum_{j=1}^{j=m} a_i^j x_j \leq c_i$$

par l'équation

$$\sum_{j=1}^{j=m} a_i^j x_j + x_{m+i} = c_i$$

et la contrainte $x_{m+i} \geq 0$.

Une solution **réalisable** du problème est la donnée de $m + n$ valeurs $x_1, x_2, \ldots, x_{m+n}$ satisfaisant les contraintes; la solution est dite **de base** si au moins m de ces valeurs sont nulles (ce qui correspond à la surface du polyèdre); une solution de base est dite **non dégénérée** si exactement m des valeurs sont nulles (ce qui correspond aux sommets du polyèdre).

Les variables correspondant aux valeurs non nulles sont dites **variables de base**, les autres **variables hors base**.

L'algorithme du simplexe consiste à améliorer une solution de base non dégénérée en augmentant une et une seule des valeurs associées aux variables hors base (ce qui revient à faire entrer dans la base l'une de ces variables), et en mettant à zéro une variable de base (ce qui revient à la faire sortir de la base). On poursuit cette amélioration jusqu'à obtention de la solution optimale.

20 Méthode des tableaux

Dans la méthode dite "des tableaux réduits", on présente à la fois les données et contraintes du problème, et la solution en cours d'optimisation, sous forme d'un tableau

Variables de base	x_1	x_2	$\ldots x_j \ldots$	x_{m+n}	Valeurs
x_i	a_i^1	a_i^2	$\ldots$	a_i^{m+n}	c_i
Z	z_1	z_2	$\ldots$	z_{m+n}	c

dans lequel les lignes correspondent aux équations

$$\sum_{j=1}^{j=m+n} a_i^j x_j = c_i$$

la dernière ligne du tableau correspondant à

$$Z = \sum_{j=1}^{j=m+n} z_j x_j + c$$

La solution à optimiser correspond aux valeurs $x_j = 0$ pour les variables hors base, $x_i = c_i$ pour les variables de base, et $Z = c$.

> **Remarque** : dans le cas fréquent où $x_1 = x_2 = \ldots = x_m = 0$ est une solution de base réalisable, le tableau associé correspond exactement aux données du problème : les variables de base sont les variables d'écart x_{m+i}, et les coefficients a_i^j, z_j, c_i et c sont ceux des équations définissant les x_{m+i} et Z).

Si x_i est une variable de base, alors on doit avoir $a_i^i = 1$, et $a_i^j = 0$ pour toute variable de base $x_j \neq x_i$: c'est-à-dire que l'équation associée à x_i permet d'exprimer x_i en fonction des variables hors base :

$$x_i = - \sum_{x_j \text{ hors base}} a_i^j x_j + c_i$$

De même, si x_j est une variable de base, on doit avoir $z_j = 0$, de façon que la fonction économique s'exprime en fonction des variables hors base :

$$Z = - \sum_{x_j \text{ hors base}} z_j x_j + c$$

Si tous les coefficients z_j sont négatifs, le maximum est atteint : en effet, du fait des contraintes $x_j \geq 0$, Z ne peut dépasser la valeur c.

Sinon, il est possible d'améliorer la solution en faisant entrer dans la base une variable hors base x_q (on choisira celle qui correspond au plus grand des coefficients z_j positifs).

On obtient une nouvelle solution de base en faisant sortir de la base l'un des x_p : pour que les contraintes soient respectées, on doit choisir p de façon que le quotient $\frac{c_p}{a_p^q}$ soit minimal parmi les $\frac{c_i}{a_i^q}$ définis (tels que $a_i^q \neq 0$) et positifs.

L'identité

$$x_p + a_p^q x_q + \sum_{x_j \text{ hors base}, j \neq q} a_p^j x_j = c_p$$

permet d'exprimer x_q en fonction des nouvelles variables hors base :

$$x_q = -\frac{1}{a_p^q} x_p - \sum_{x_j \text{ hors base}, j \neq q} \frac{a_p^j}{a_p^q} x_j + \frac{1}{a_p^q} c_p$$

En remplaçant x_q par cette expression dans les équations associées aux lignes du tableau, on obtiendra des équations correspondant à la nouvelle base :

$$\sum_{x_j \text{ hors base}} a_i^j x_j + x_i = c_i$$

$$Z = \sum_{x_j \text{ hors base}} z_j x_j + c$$

Dans la pratique, le tableau associé à la nouvelle solution s'obtient comme suit à partir du précédent :

- dans la ligne d'indice p, on remplace l'indice p par l'indice q, et on divise tous les coefficients par a_p^q;
- pour toute ligne d'indice $i \neq p$, on remplace chaque coefficient a_i^j par $a_i^j - \frac{a_p^j}{a_p^q} a_i^q$; de même, on remplace c_i par $c_i - \frac{c_p}{a_p^q} a_i^q$;
- dans la dernière ligne, on remplace chaque z_j par $z_j - \frac{a_p^j}{a_p^q} z_q$, et la valeur c par $c + \frac{c_p}{a_p^q} z_q$.

Exemple

La résolution de l'exemple du chapitre par la méthode des tableaux réduits commence par l'introduction de variables d'écart $u, v, w \geq 0$, avec

$$\begin{cases} x + 2y - w = 60 \\ 3x + 2y - v = 120 \\ 3x + y - u = 90 \end{cases}$$

et la fonction économique $Z = -x - y$ à maximiser. La solution de base non dégénérée $x = 0, y = 90$ correspond à la base y, v, w; en remplaçant y par $-3x + u + 90$ on obtient

$$\begin{cases} 5x - 2u + w = 120 \\ 3x - 2u + v = 60 \\ 3x + y - u = 90 \\ Z = 2x - u - 90 \end{cases}$$

ce qui correspond au tableau

	x	y	u	v	w		
w	5		-2		1	120	120/5 = 24
v	3		-2	1		60	60/3 = 20
y	3	1	-1			90	90/3 = 30
Z	**2**		-1			-90	

On fait entrer x dans la base, et sortir v :

	x	y	u	v	w		
w			4/3	-5/3	1	20	20/(4/3)=15
x	1		-2/3	1/3		20	20/(-2/3) est négatif
y		1	1	-1		30	30/1=30
Z			**1/3**	-2/3		-50	

On fait entrer u dans la base, et sortir w :

	x	y	u	v	w	
u			1	-5/4	3/4	15
x	1			-1/2	1/2	30
y		1		1/4	-3/4	15
Z				-1/4	-1/4	-45

Les coefficients z_j sont tous deux négatifs : le maximum est atteint. On retrouve ainsi la solution $x = 30$, $y = 15$.

Index