RECHERCHE OPERATIONNELLE

INTRODUCTION

La recherche opérationnelle est une discipline dont le but est de fournir des méthodes pour répondre à un type précis de problème, c'est-à-dire à élaborer une démarche universelle pour un type de problème qui aboutit à la ou les solutions les plus efficaces. La particularité de la recherche opérationnelle est que les méthodes proposées sont des démarches rationnelles basées sur des concepts et outils mathématiques et/ou statistiques.

Généralement, ces méthodes sont employées sur des problèmes tels que leur utilisation "manuelle" devient impossible. C'est pourquoi, du fait qu'elles sont rationnelles, les démarches proposées par la recherche opérationnelle peuvent être traduites en programmes informatiques.

Cette traduction d'une démarche en un programme informatique n'est pas sans difficulté. Tout d'abord, le temps d'exécution du programme résultant et/ou la place occupée dans la mémoire de l'ordinateur peuvent ne pas être acceptables. Ainsi, une méthode en recherche opérationnelle sera jugée sur ces critères de temps et de place. Plus une méthode sera rapide et peu gourmande en mémoire, plus elle sera considérée bonne.

Les ordinateurs ont une structure particulière qui fait que toutes les propriétés des mathématiques traditionnelles ne sont pas toujours respectées. Ainsi, une démarche prouvée fonctionner admirablement en théorie peut s'avérer être complètement inexploitable en pratique. Notamment, les nombres réels dans un ordinateur ne peuvent pas être représentés de manière exacte, ils sont arrondis. On voit donc facilement qu'une répétition excessive d'arrondis dans un calcul peut entraîner des erreurs importantes dans les résultats finaux. Les méthodes employées en recherche opérationnelle doivent prendre en compte ce genre de problème.

PLAN DU COURS

Dans ce cours, nous verrons différents outils de recherche opérationnelle sans apporter de justifications mathématiques très détaillées et rigoureuses. Après quelques exemples qui permettront de mieux cerner le domaine de la recherche opérationnelle, nous introduirons un outil à la fois graphique et théorique: les **graphes**. Afin de mieux appréhender la complexité d'un problème ou la rapidité d'un algorithme, nous nous intéresserons à la **théorie de la complexité**. Enfin, nous verrons un autre outil important de la recherche opérationnelle qui est la **programmation linéaire**. L'avantage de cet outil est d'apporter une solution générique à la résolution de nombreux problèmes. De plus, cet outil est disponible sous différentes formes pour une utilisation informatique. Voici le plan du cours.

- **Les arbres**
- **Représentation des graphes**
- **Efficacité des algorithmes, complexité des problèmes**
- **Recherche du plus court chemin**
- **Ordonnancement, recherche du plus long chemin**
- **Recherche du flot maximum**
- **Programmation linéaire**

EXEMPLES

- **Chemin le plus court / le plus long**

  Soit un ensemble de villes et des chemins directs reliant ces villes entre elles. Le problème dit "du plus court chemin" consiste à trouver pour une ville de départ donnée et une ville d'arrivée donnée le chemin le plus court qui relie ces deux villes. Le problème peut également être de trouver un chemin le plus court pour chaque couple de villes. Pour certains problèmes, trouver le plus long chemin entre deux points peut être intéressant.

- **Ordonnancement / planification**

  Considérons la gestion d'un grand projet. Il est constitué de différentes étapes à réaliser. Il est logique de penser que certaines tâches doivent être effectuées avant d'autres alors que certaines peuvent très bien être effectuées en même temps. Ainsi, on établit une certaine relation d'ordre entre les étapes. Un premier problème consiste à trouver une planification des tâches qui aboutisse à la réalisation du projet en un minimum de temps. Ensuite, il peut être intéressant de détecter les étapes dites "critiques" dont le moindre retard peut affecter toute la suite du projet.

- **Flot maximum**

  Soit des châteaux d'eau ayant un débit constant. Ils desservent un certain nombre de villes, chacune ayant des besoins quantifiés constants. L'eau est acheminée à travers des conduits dont le débit maximum est connu. Le problème est de trouver un moyen de satisfaire au mieux les demandes de chaque ville. En d'autres termes, essayer d'apporter le plus d'eau possible vers les villes.

- **Flot de coût minimum**

  Il s'agit d'un problème semblable à celui du flot maximum mais on suppose en plus qu'un coût fonction du débit est associé à l'utilisation d'un conduit. Le problème devient alors de satisfaire les villes mais de la manière la moins onéreuse.

- **Sac à dos**

  Un randonneur prépare son sac à dos pour partir en excursion. Bien entendu, il veut

éviter d'avoir un sac trop lourd et décide de se limiter dans le choix des objets qu'il emporte afin de ne pas dépasser un certain poids. Cependant, il veut emporter le maximum de choses utiles. Pour cela, il affecte une valeur quantitative à chaque objet en plus de son poids (plus la valeur est importante, plus le randonneur juge l'ob jet important). Le problème peut donc se formuler de la manière suivante: trouver l'ensemble des objets dont la somme des utilités est maximum tout en ne dépassant pas un poids fixé.

- **Affectation**

  Des modifications de postes sont effectuées dans une entreprise. Plusieurs personnes doivent être affectées à de nouveaux postes. Ainsi, chacun classe par ordre de préférence les postes qu'il veut occuper. Le problème ici est d'attribuer à chaque personne un poste tout en essayant de satisfaire au mieux le souhait de chacun.

- **Voyageur de commerce**

  Un voyageur de commerce doit démarcher dans un certain nombre de villes. Il connaît bien entendu la distance qui sépare les villes entre elles. Cependant, le voyageur de commerce veut perdre le moins de temps possible dans ses déplacements. Le problème est donc de trouver un chemin qui passe par toutes les villes une et une seule fois et qui soit le court possible.

Dans tous ces exemples, il existe une méthode simple pour résoudre le problème. En effet, il suffit d'énumérer toutes les possibilités et d'en dégager la ou les meilleures. Cependant, on s'aperçoit que plus le problème est compliqué en terme d'éléments mis en jeu, plus le nombre de possibilités croît de manière non pas linéaire (proportionnelle) mais plutôt exponentielle. Par exemple, le problème d'affectation présenté précédemment avec 100 personnes a 100! (100 x 99 x 98 x ... x 1) solutions. Le simple fait de rajouter une personne dans le problème va multiplier par 101 le nombre de solutions.

Généralement en recherche opérationnelle, on a souvent à traiter des problèmes dont le nombre de solutions devient rapidement difficile à imaginer. Bien que les exemples vus ici soient petits, il faut bien comprendre qu'en réalité, on sera confronté à des problèmes de taille beaucoup plus importante. Ce qui explique que l'on cherche des méthodes toujours plus efficaces pour résoudre les problèmes.

---

1. LES GRAPHES

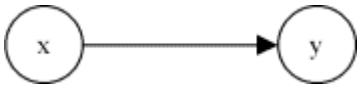
DEFINITIONS ET THEOREMES

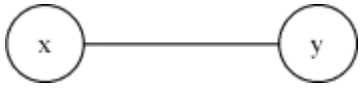
Graphe

Un graphe est un ensemble de noeuds qui sont reliés entre eux par des arcs. Mathématiquement, un graphe est représenté par un couple de deux ensembles G = (X;U) où X est l'ensemble des noeuds et U l'ensemble des arcs.

Arc

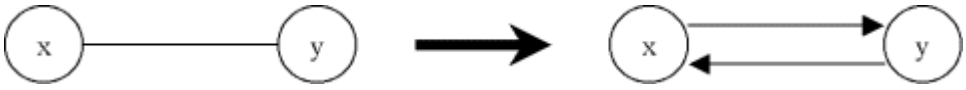
Un arc relie deux noeuds entre eux, il sera donc représenté par un couple (x;y) où x et y sont des noeuds. Un arc peut être orienté, c'est-à-dire que l'ordre de x et de y est important dans le couple (x;y). Un arc peut ne pas être orienté et dans ce cas, l'ordre de x et de y dans le couple (x;y) n'a aucune importante, donc (x;y) = (y;x). Les arcs sont représentés de la manière suivante.

- Arc orienté: 

- Arc non orienté: 

Remarque

Un arc non orienté peut toujours être transformé en une situation où l'on n'a que des arcs orientés.



C'est pourquoi, dans la suite du cours, on utilisera le plus souvent des graphes orientés, c'est-à-dire des graphes dont les arcs sont tous orientés.

Boucle

On appelle boucle un arc dont l'extrémité initiale est égale à son extrémité finale. Par exemple, (x;x) est une boucle.

Adjacence

Pour un arc $u = (x;y)$ on dit que:

- x est adjacent à y,
- y est adjacent à x,
- x et y sont adjacents à u,
- u est adjacent à x et y.

Degré

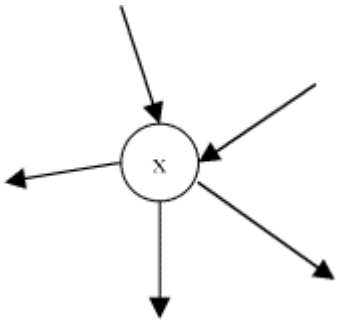
Le demi-degré extérieur d'un noeud est le nombre d'arcs adjacents qui en partent.
On le note $d^+(x)$ et $d^+(x) = |\{u \in U \mid u = (x;y) \text{ où } y \in X\}|$.

Le demi-degré intérieur d'un noeud est le nombre d'arcs adjacents qui y arrivent.
On le note $d^-(x)$ et $d^-(x) = |\{u \in U \mid u = (y;x) \text{ où } y \in X\}|$.

Le degré d'un noeud est le nombre d'arcs qui lui sont adjacents.
On le note $d(x)$ et $d(x) = d^+(x) + d^-(x)$.

Exemple

Dans le graphe suivant, $d^+(x) = 3$, $d^-(x) = 2$, $d(x) = 5$.



Graphe régulier

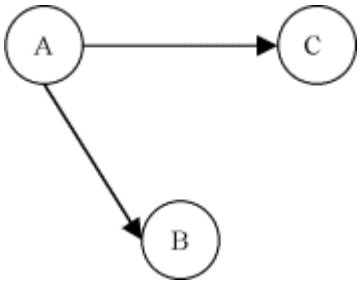
Un graphe est dit régulier si les degrés de tous ses sommets sont égaux.

Graphe complet

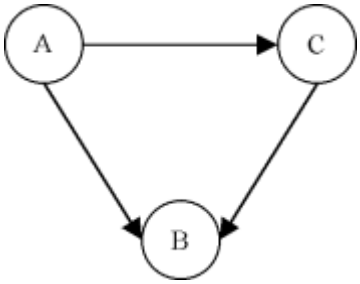
Un graphe est dit complet si tous les noeuds sont adjacents deux à deux.
Autrement dit, $(x;y) \notin U \Rightarrow (y;x) \in U$.

Exemple

Graphe non complet:



Graphe complet:



Chaîne

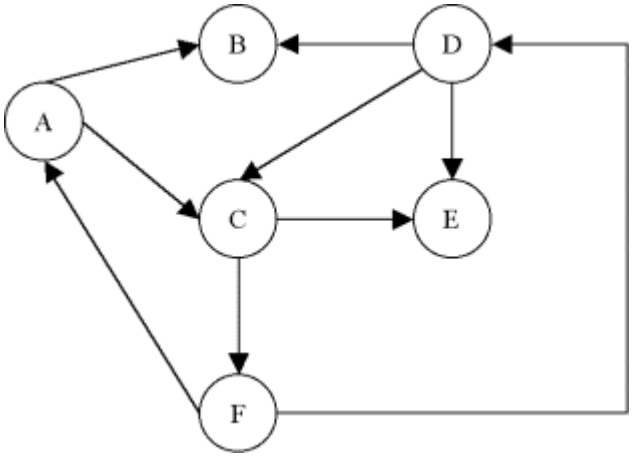
Une chaîne de x à y est une séquence d'arcs c = ((x;e),(z;e),(z;a)...(s;t),(u;t),(u;y)) où deux arcs qui se suivent sont adjacents et où x doit être une extrémité du premier arc et y une extrémité du dernier.

Chemin

Un chemin de x à y est une chaîne dans laquelle les arcs sont orientés et tels que:

- x est l'extrémité initiale du premier arc,
- y est l'extrémité terminale du dernier arc,
- l'extrémité terminale d'un arc est l'extrémité initiale de l'arc qui le suit dans la séquence.

Exemple



- ch1 = ((A;C),(C;E)) est un chemin de A à E.
- ch2 = ((A;C),(C;F),(F;A),(A;C),(C;E)) est un chemin de A à E.
- ch3 = ((A;C),(C;F),(F;D),(D;C),(C;E)) est un chemin de A à E.
- ch4 = ((A;B),(B;D),(D;E)) est une chaîne de A à E.
- ch5 = ((A;B),(B;D),(D;C),(C;A),(A;B),(B;D),(D;E)) est une chaîne de A à E.
- ch6 = ((A;B),(B;D),(D;C),(C;F),(F;D),(D;E)) est une chaîne de A à E.

Chemin, chaîne simples

Un chemin simple est un chemin qui ne contient pas plusieurs fois le même arc. Dans l'exemple précédent, ch1 et ch3 sont des chemins simples mais pas ch2.

Une chaîne simple est une chaîne qui ne contient pas plusieurs fois le même arc. Dans l'exemple précédent, ch4 et ch6 sont des chaînes simples mais pas ch5.

Chemin, chaîne élémentaires

Un chemin élémentaire est un chemin qui ne passe pas plus d'une fois par un noeud. Dans l'exemple précédent, ch1 est un chemin élémentaire mais pas ch2, ni ch3.

Une chaîne élémentaire est une chaîne qui ne passe pas plus d'une fois par un noeud. Dans l'exemple précédent, ch4 est un chemin élémentaire mais pas ch5, ni ch6.

Connexité, forte connexité

On définit la connexité par une relation entre deux noeuds de la manière suivante.

> x et y ont une relation de connexité ⇔ il existe une chaîne entre x et y ou bien x = y.

On définit la forte connexité par une relation entre deux noeuds de la manière suivante.

> x et y ont une relation de forte connexité ⇔ (il existe un chemin de x à y et un chemin de y à x) ou bien x = y.

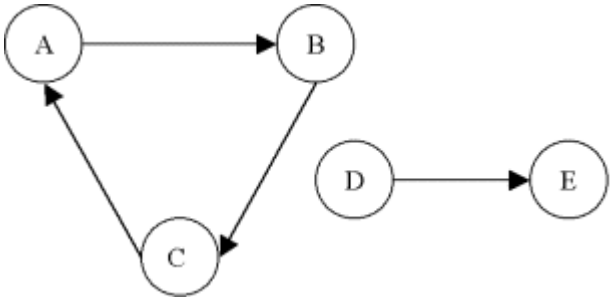
Graphe connexe, fortement connexe

Un graphe est dit connexe si tous ses noeuds ont deux à deux la relation de connexité.

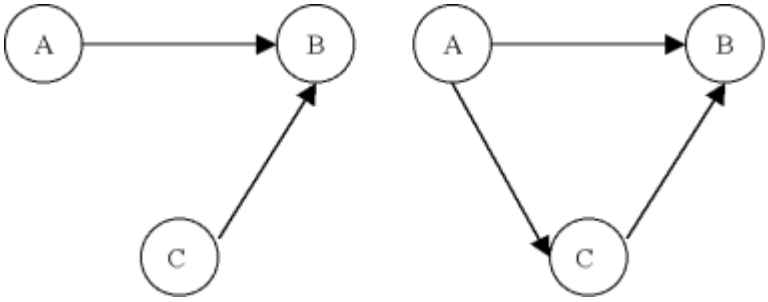
Un graphe est dit fortement connexe si tous ses noeuds ont deux à deux la relation de forte connexité.

Exemple

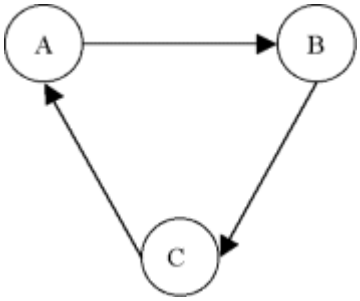
Graphe non connexe:



Graphes connexes mais non fortement connexes:



Graphe fortement connexe:



Composante connexe, fortement connexe

On appelle composante connexe un ensemble de noeuds qui ont deux à deux la relation de connexité. De plus, tout noeud en dehors de la composante n'a pas de relation de connexité avec aucun des éléments de la composante.

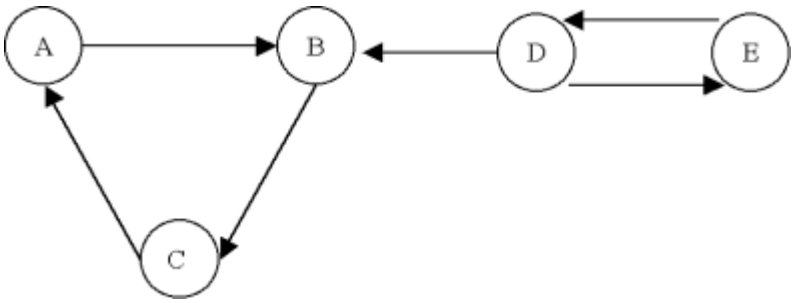
De même, on appelle composante fortement connexe un ensemble de noeuds qui ont deux à deux la relation de forte connexité. De plus, tout noeud en dehors de la composante n'a pas de relation de forte connexité avec aucun des éléments de la composante.

Dans l'exemple précédent du graphe non connexe, {A,B,C} est une composante fortement connexe du graphe et {D,E} est une composante connexe.

Graphe réduit

On appelle graphe réduit du graphe G le graphe G' pour lequel chaque noeud est associé à une composante fortement connexe de G. De plus, un arc relie un noeud x' à un noeud y' dans le graphe G' si il existe un arc qui relie x à y dans G où x appartient à la composante fortement connexe de G associée à x' et où y appartient à la composante fortement connexe de G associée à y'.

Exemple



Ce graphe G n'est pas fortement connexe, car on ne peut pas trouver de chemin de A à D par exemple. Par contre, on identifie deux composantes fortement connexes A' = {A,B,C}et B' = {D,E}. Le graphe réduit du graphe G' est:

Circuit, cycle

Un circuit contenant un noeud x est un chemin de x à x. Un circuit est une séquence circulaire d'arcs et donc, contrairement au chemin, un circuit n'a ni début, ni fin.

De même, un cycle contenant un noeud x est une chaîne de x à x. Un cycle est également une séquence circulaire d'arcs.

Un circuit (ou une chaîne) est élémentaire si le chemin (ou la chaîne) associé(e) est élémentaire.

Sous-graphe, graphe partiel, sous-graphe partiel

Soit un graphe G = (X;U), X' inclu dans X et U' inclu dans U et U'' = {(x;y) ∈ U | x ∈ X' et y ∈ X'}.

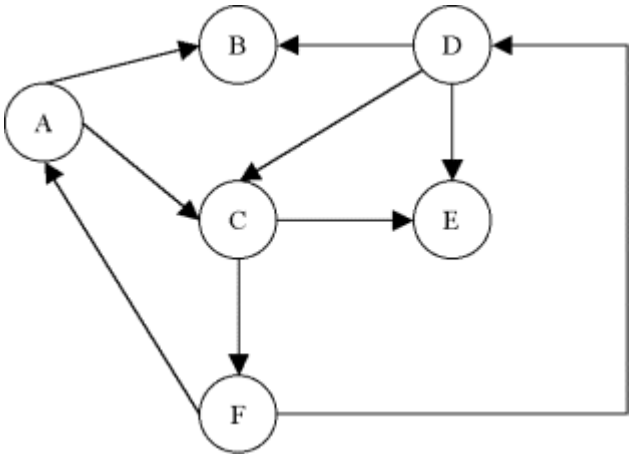
Le graphe (X';U'') est appelé "sous-graphe" de G.
Le graphe (X;U') est appelé "graphe partiel" de G.
Le graphe (X';U' ∩ U'') est appelé "sous-graphe partiel" de G.

En d'autres termes:

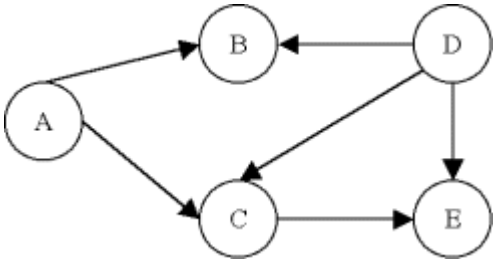
Un sous-graphe de G, c'est G privé de quelques noeuds et des arcs adjacents à ces noeuds.
Un graphe partiel de G, c'est G privé de quelques arcs.
Un sous-graphe partiel de G, c'est un graphe partiel d'un sous-graphe de G.

Exemple

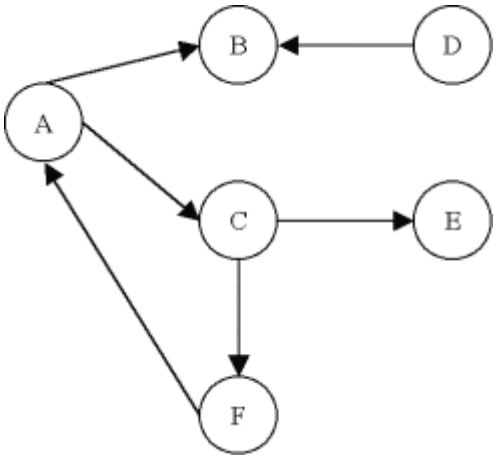
Un graphe G:



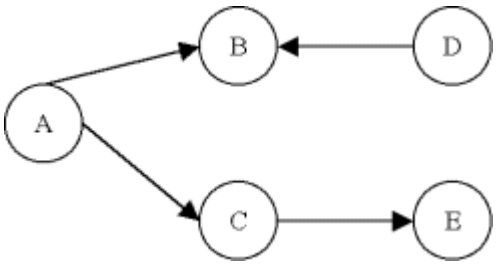
Un sous-graphe de G:



Un graphe partiel de G:



Un sous-graphe partiel de G:



COMPOSANTES FORTEMENT CONNEXES

Lors de la conception d'un réseau de communication notamment, il peut être intéressant de savoir si la configuration choisie permet une communication de n'importe quel point à n'importe quel autre. Un moyen de le vérifier est de représenter le réseau sous la forme d'un graphe et de vérifier qu'il est fortement connexe. Pour cela, l'algorithme suivant est proposé. Il détermine la composante fortement connexe d'un graphe contenant un point donné. Ensuite, pour le problème énoncé, il suffit de choisir un point au hasard, d'exécuter l'algorithme et de vérifier que la composante fortement connexe trouvée est bien le graphe en entier. Un algorithme permettant de trouver toutes les composantes fortement connexes est également proposé.

Recherche d'une composante fortement connexe

Cet algorithme recherche la composante fortement connexe d'un graphe G contenant un sommet a. L'idée de cet algorithme est de parcourir le graphe à partir du point a dans le sens direct (i.e. en suivant les flèches des arcs) et de créer un ensemble des noeuds parcourus. La même chose est effectuée dans le sens indirect (i.e. en suivant les flèches des arcs en sens inverse) et de créer un deuxième ensemble des noeuds parcourus. Le premier ensemble regroupe les noeuds accessibles à partir de a et le deuxième ensemble regroupe les noeuds qui peuvent atteindre a. L'intersection de ces deux ensembles donne les noeuds qui à la fois peuvent atteindre a et sont accessibles à partir de a. Cette intersection est donc la composante fortement connexe qui contient a.

Algorithme

Titre: ComposanteFortementConnexe
Entrées: G = (X;U) un graphe, a un sommet.
Sortie: X' un sous-ensemble de sommets.
Variables intermédiaires: X1 et X2 deux sous-ensemble de sommets, examiné() une fonction, x un noeud.

Début
```
X1 ← {a};
pour tout x ∈ X faire examiné(x) ← faux;

tant que ∃ x ∈ X1 | non examiné(x) faire
 examiné(x) ← vrai;
 pour tout u = (x;y) ∈ U | y ∉ X1 faire X1 ← X1 ∪ {y};
fin tant que;

X2 ← {a};
pour tout x ∈ X1 faire examiné(x) ← faux;

tant que ∃ x ∈ X2 | non examiné(x) faire
 examiné(x) ← vrai;
 pour tout u = (y;x) ∈ U | y ∉ X2 faire X2 ← X2 ∪ {y};
fin tant que;

X' ← X1 ∩ X2;
```
Fin

Recherche de toutes les composantes fortement connexes

Pour trouver toutes les composantes fortement connexes d'un graphe, il suffit de choisir au hasard un noeud et de déterminer, grâce à l'algorithme précédent, la composante fortement connexe qui le contient. On obtient alors une première composante fortement connexe X1. Ensuite, parmi les noeuds qui ne font pas partie de X1, on en prend un au hasard pour déterminer la composante fortement connexe qui le contient. On obtient X2. On recommence ainsi jusqu'à ce que tous les noeuds appartiennent à une composante fortement connexe.

Algorithme

Titre: ComposantesFortementConnexes
Entrées: G = (X;U) un graphe.
Sortie: C = {C_1,...,C_n} un ensemble de composantes connexes.
Variables intermédiaires: X' un sous-ensemble de sommets, i un entier, x un noeud.

Début
```
X' ← X;
i ← 1;

tant que X' ≠ ∅ faire
 choisir x dans X';
 C_i ← ComposanteFortementConnexe(G,x);
 X' ← X' - C_i;
 i ← i + 1;
fin tant que;
```
Fin

2. LES ARBRES

DEFINITIONS ET THEOREMES

Nombre d'arcs dans un graphe

Soit n le nombre de noeuds d'un graphe G = (X;U), n = |X|.
Soit m le nombre d'arcs de G, m = |U|.

Si G est connexe, $m \geq n - 1$.
Si G est sans cycle, $m \leq n - 1$.

Arbre

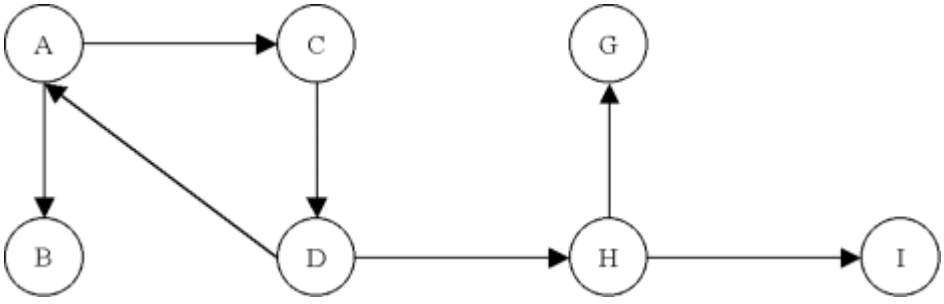
Un arbre est un graphe connexe sans cycle. Il a donc n - 1 arcs. On peut donc dire qu'un arbre est un graphe qui connecte tous les noeuds entre eux avec un minimum d'arcs.

Remarques

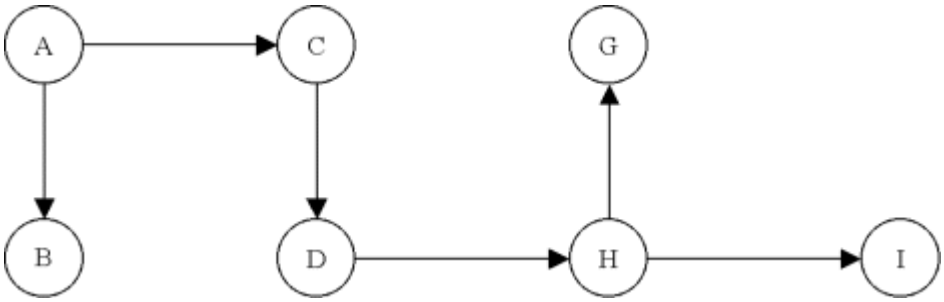
- L'ajout du moindre arc supplémentaire dans un arbre crée un cycle.
- Un graphe connexe possède un graphe partiel qui est un arbre.

Exemple

Un graphe connexe:



Un arbre extrait du graphe précédent:



Forêt

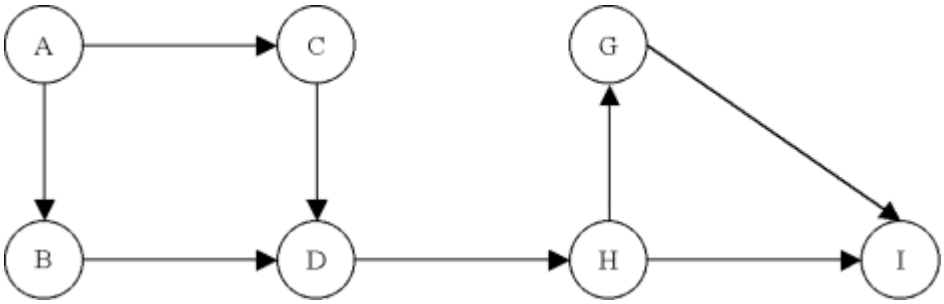
On appelle forêt un graphe dont chaque composante connexe est un arbre.

Racine, antiracine

Un noeud a d'un graphe G est une racine de G s'il existe un chemin joignant a à chaque noeud du graphe G.

Un noeud a d'un graphe G est une antiracine de G s'il existe un chemin joignant chaque noeud du graphe G à a.

Exemple



A est une racine du graphe.
I est une antiracine du graphe.

Arborescence, anti-arborescence

Un graphe G est une arborescence de racine a si G est un arbre et si a est une racine.

Un graphe G est une anti-arborescence d'antiracine a si G est un arbre et si a est une antiracine.

ARBRE DE COUT MINIMUM / MAXIMUM

Imaginons que l'on associe une valeur, un poids, à chaque arc d'un graphe G. Le problème de l'arbre de coût maximum consiste à trouver un arbre, graphe partiel de G, dont la somme des poids des arcs est maximum. En pratique, les problèmes se ramène plutôt à trouver l'arbre de coût minimum, ce qui ne change pas fondamentalement le principe des algorithmes proposés ici.

Par exemple, minimiser le coût d'installation de lignes électriques entre des maisons peut être modélisé par la recherche d'un arbre de coût minimum. En effet, on veut connecter toutes les maisons entres elles sans avoir de lignes inutiles (d'où la recherche d'un arbre). Ensuite, on veut utiliser le moins de câble possible, aussi on associera à chaque possibilité de connexion la longueur de câble nécessaire et on cherchera à minimiser la longueur totale de câble utilisée.

Dans ce cours, deux algorithmes sont proposés. L'efficacité de chacun d'eux dépend du choix de représentation du graphe et de la structure même du graphe.

ALGORITHME DE KRUSKAL

Le principe de l'algorithme de Kruskal pour trouver un arbre de poids minimum dans un graphe G est tout d'abord de trier les arcs par ordre croissant de leur poids. Ensuite, dans cet ordre, les arcs sont ajoutés un par un dans un graphe G' pour construire progressivement l'arbre. Un arc est ajouté seulement si son ajout dans G' n'introduit pas de cycle, autrement dit, si G' reste un arbre. Sinon, on passe à l'arc suivant dans l'ordre du tri.

Algorithme

Titre: Kruskal
Entrées: G = (X;U) un graphe.
Sortie: U' un ensemble d'arcs.
Variables intermédiaires: i un entier.

Début

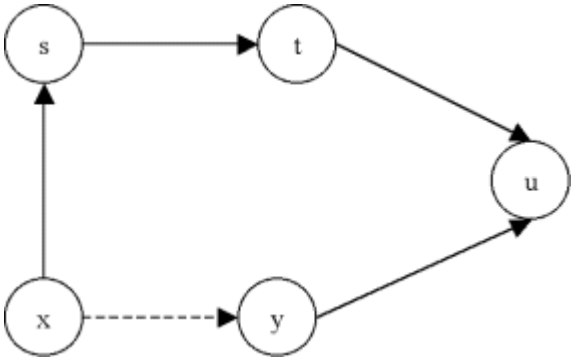
```
trier les arcs de G dans l'ordre croissant des poids;
[On les notera u_1, u_2, ... , u_m.]

U' ← ∅;

pour i ← 1 à m faire
 si le graphe (X;U' ∪ {u_i}) ne contient pas de cycle alors
  U' ← U' ∪ {u_i};
 fin si;
fin tant que;
```

Fin

Justification

Supposons que l'algorithme est effectué quelques itérations. Considérons maintenant l'ajout de l'arc u = (x;y) dans l'arbre. On suppose que cet ajout n'introduit pas de cycle dans G'. Est-on certain que u permet la construction de l'arbre de coût minimum ? En fait, x et y doivent être connectés d'une manière ou d'une autre. Donc si ce n'est pas u qui les relie, ce

sera une autre chaîne C = (x,s,t,u,y) de x à y comme le montre la figure ci-dessous. Comme u n'introduit pas de cycle, cette autre chaîne n'est pas encore construite, elle le sera plus tard. Cela signifie que tous les arcs de cette chaîne ont un coût supérieur ou égal à celui de u. Donc, le fait de choisir la chaîne C pour connecter x à y est plus onéreuse que de connecter x à y par u. En effet, supprimons n'importe quel arc de la chaîne C et remplaçons le par u. On obtiendra toujours un arbre mais de coût plus faible. Cela justifie le choix de u et donc la démarche globale de l'algorithme de Kruskal.



ALGORITHME DE PRIM

Le principe de l'algorithme de Prim pour trouver un arbre de poids minimum dans un graphe G est de fusionner, deux par deux les noeuds de G pour obtenir finalement un arbre. Par fusionner, on entend remplacer deux noeuds par un seul. Tous les arcs adjacents à l'un ou l'autre des anciens noeuds deviennent adjacents au nouveau noeud. Le choix des noeuds que l'on fusionne est fait en choisissant au hasard un noeud et en cherchant un arc adjacent (autre qu'une boucle) qui a le coût le plus faible. Ce qui fournit le deuxième noeud de la fusion.

Algorithme

Titre: Prim
Entrées: G = (X;U) un graphe.
Sortie: U' un ensemble d'arcs.
Variables intermédiaires: x un noeud, u un arc.

Début

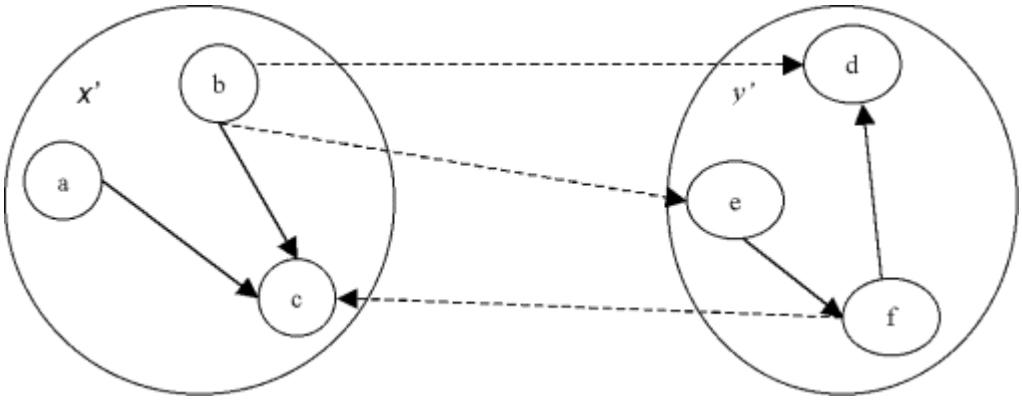
```
U' ← ∅;

tant que |U'| < n - 1 faire
 choisir x ∈ X;

 choisir u ∈ U | u = (x;y) ou u = (y;x) avec y ≠ x
 et coût(u) = min{coût(v) | v ∈ U avec v = (x;y) ou v = (y;x), x ≠ y};

 U' ← U' ∪ {u};
 fusionner x et y; [x et y deviennent un seul noeud.]
fin tant que;
```

Fin

Justification

Tout d'abord, si on choisit un arc u = (x;y) où x et y sont des noeuds qui ne sont pas le résultat d'une fusion, on peut dire que le graphe résultant ({x,y};{u}) est un arbre de poids minimum.

Maintenant, si on choisit un arc u = (x';y') où x' et y' sont des noeuds qui sont le résultat de fusions, la figure ci-dessous peut représenter la situation. En fait, x' et y' représentent des arbres de poids minimum (hypothèse de récurrence). Si u' est l'arc entre x' et y' de poids minimum, le graphe ({a,b,c,d,e,f};{(a;c),(b;c),(d;f),(e;f),u}) sera un arbre de poids minimum. Donc à la dernière étape de l'algorithme, on obtiendra bien un arbre de poids minimum qui est un graphe partiel de G.



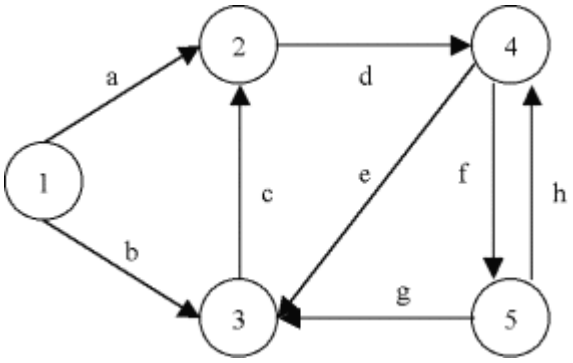
3. REPRESENTATION DES GRAPHES

MATRICE D'INCIDENCE NOEUD-ARC

Un graphe peut être représenté par une matrice n x m (n = |X| et m = |U|), dite d'incidence, pouvant contenir uniquement les valeurs 0, 1, -1. Chaque ligne de la matrice est associée à un noeud et chaque colonne à un arc. Ainsi, une case indique la relation qu'il existe entre un noeud et un arc.

- 0 signifie que le noeud et l'arc ne sont pas adjacents,
- 1 signifie que le noeud est l'extrémité initiale de l'arc,
- -1 signifie que le noeud est l'extrémité terminale de l'arc.

Exemple



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

Remarques

- Seulement 2m cases de la matrice sont non nulles sur mn cases.
- Cette représentation occupe beaucoup de place en mémoire.
- De plus, son utilisation apporte rarement de bons résultats au niveau des algorithmes. Notamment, pour parcourir le graphe, son emploi est difficile.
- Par contre, pour quelques problèmes comme le flot de coût minimum, cette matrice a

une signification directe importante et peut donc s'avérer utile.

MATRICE D' ADJACENCE NOEUD-NOEUD

Un graphe peut être représenté par une matrice n x n (n = |X|), dite d'adjacence, pouvant contenir uniquement les valeurs 0, 1. Chaque ligne et chaque colonne de la matrice représente un noeud. Ainsi, une case indique la relation qu'il existe entre deux noeuds.

- 0 signifie que les deux noeuds ne sont pas reliés par un arc,
- 1 signifie que les deux noeuds sont reliés par un arc orienté.

Exemple

Le graphe précédent sera représenté par la matrice suivante.

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

Remarques

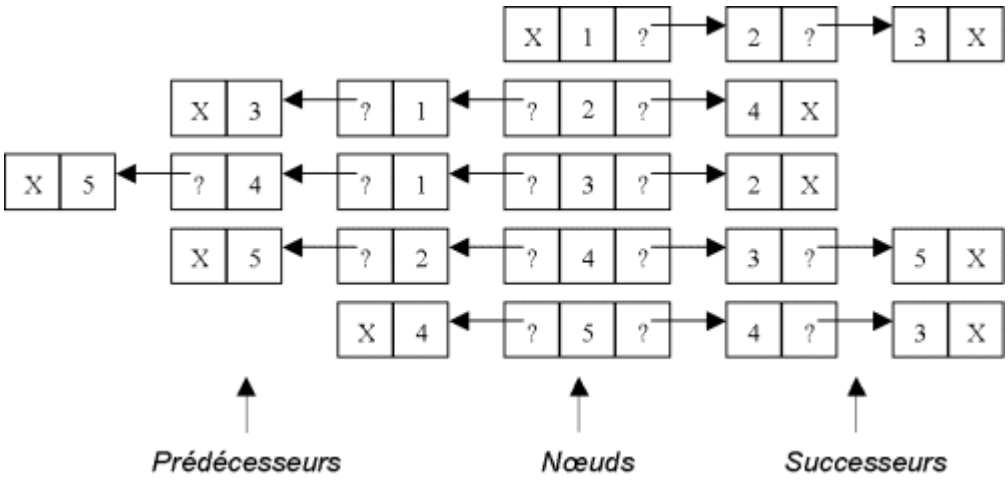
- Seulement m cases de la matrices sont non nulles sur n^2 cases.
- Cette représentation est efficace au niveau de l'espace mémoire utilisé lorsque le graphe est suffisamment dense (i.e. lorsqu'il y a suffisamment d'arcs).
- Elle permet d'implémenter assez facilement les algorithmes.
- Deux arcs ayant les mêmes extrémités ne peuvent pas être représentés avec cette matrice.

LISTES

Un graphe peut être représenté par des listes. Nous proposons ici une possibilité mais de nombreuses autres peuvent convenir. On définit tout d'abord une liste des noeuds et à chaque noeud, on associe une liste de noeuds successeurs et une liste de noeuds prédécesseurs.

Exemple

Le graphe précédent sera représenté de la manière suivante.



Remarques

- Cette représentation est nettement plus efficace au niveau de la mémoire occupée que les deux représentations précédentes.
- Elle est très bien adaptée au parcours du graphe, aussi bien en sens direct qu'en sens indirect.
- Cette structure est souple. L' ajout et la suppression d' arcs ou de noeuds sont plus aisés qu' avec les représentations matricielles.

4. EFFICACITE DES ALGORITHMES, COMPLEXITE DES PROBLEMES

INTRODUCTION

Considérons le jeu d'échecs. On sait qu'il existe une procédure finie qui permet de déterminer la meilleure stratégie à partir d'une situation donnée. Cependant, jusqu'à présent, on ne connaît pas d'autre méthode que l'exploration de tous les coups possibles. En résumé, on sait qu'en un temps fini, on peut apporter la solution au problème. Cependant, sur le plan pratique on s'aperçoit immédiatement que l'énumération exhaustive prendrait trop de temps. La première notion qui apparaît donc est celle d'algorithme efficace. Pour un problème donné, on cherche à avoir un algorithme dit efficace, c'est-à-dire que le temps nécessaire à son exécution ne soit pas trop important. Ensuite, intervient la notion de problème facile ou difficile. Un problème sera dit facile si on peut le résoudre facilement, autrement dit s'il ne faut pas trop de temps pour trouver la solution. Ainsi, s'il existe un algorithme efficace pour un problème donné, alors ce dernier est dit facile. Mais comment caractériser les problèmes difficiles. Un problème pour lequel on ne connaît pas d'algorithme efficace, est-il difficile ou facile ? Ce n'est pas parce que l'on ne connaît pas d'algorithme efficace qu'il n'est pas facile et qu'un jour on ne trouvera pas un moyen de le résoudre. A l'inverse, il peut exister des problèmes intrinsèquement compliqués, pour lesquels on ne pourra jamais trouver d'algorithme efficace.

De nombreuses personnes se sont penchées sur ces problèmes et ont développé une théorie dite de la complexité. Nous ne verrons pas ici en détail les fondements de cette théorie mais tenterons plutôt d'acquérir une vision globale de cette problématique. Dans un premier temps, on va voir comment l'efficacité d'un algorithme est mesurée avant de définir plus précisément ce qu'est un algorithme efficace. Ensuite, On discutera de la classification des problèmes selon leur difficulté à être résolus.

COMPLEXITE D'UN ALGORITHME

On désigne par complexité d'un algorithme le nombre d'opérations nécessaires à celui-ci pour s'exécuter. Bien évidemment, ce nombre peut varier en fonction de ce que l'on appelle les données d'entrées, c'est-à-dire les paramètres que l'on donne à l'algorithme. Par exemple, un algorithme de tri d'éléments dans un tableau ne s'exécutera pas avec le même nombre d'opérations s'il y a 10 éléments ou s'il y en a 100. Ainsi, on cherchera à estimer la complexité d'un algorithme en fonction de la taille des données entrées. Par exemple, dans le cas du tableau, on exprimera la complexité en fonction de la taille du tableau. Pour une matrice, ce serait en fonction de sa largeur et de sa hauteur. De plus, la nature même des données pour une même taille peut ne pas aboutir au même nombre d'opérations à l'exécution de l'algorithme. En effet, si le tableau est déjà trié, l'exécution de l'algorithme de tri risque d'être très rapide comparée au cas d'un tableau totalement en désordre. C'est pour cela que l'on estime le nombre d'opérations dans le pire des cas.

En résumé, on mesure l'efficacité d'un algorithme par une expression mathématique

qui indique le nombre d'opérations indispensables à l'exécution de l'algorithme en fonction de la taille des données en entrées tout en supposant le pire des cas.

Le critère de rapidité n'est pas toujours celui qui nous intéresse. On peut aussi vouloir estimer la place utilisée par un algorithme dans la mémoire de l'ordinateur. Dans ce cas, on parle de complexité spatiale alors que jusqu'à présent on considérait la complexité temporelle. La complexité spatiale peut se définir d'une manière semblable à la complexité temporelle. Dans la suite de ce cours, on s'intéressera uniquement à la complexité temporelle.

ALGORITHME EFFICACE

Une fois la complexité définie de manière succincte, on peut tenter de définir ce qu'est un algorithme efficace. **Un algorithme sera dit efficace si sa complexité est bornée par un polynôme ayant la taille des données comme variable.** Par exemple, un algorithme qui a en entrée un tableau de n éléments et qui a une complexité de n^2 est un algorithme efficace. On dit aussi que l'algorithme est polynomial. Cette définition est justifiée par le fait qu'on s'intéresse aux performances des algorithmes quand la taille des données en entrée devient très importante. Par exemple, considérons les algorithmes A, B et C. Leur complexité sont les suivantes.

- $C_a = 80n$,
- $C_b = 10n^2$,
- $C_c = n!$.

Avec 4 éléments, il faut respectivement 320, 160 et 24 opérations aux algorithmes A, B et C pour s'exécuter. Le plus efficace pour 4 éléments est donc C. Considérons maintenant 20 éléments, il faut respectivement 1600, 4000 et $2.4\ 10^{18}$ opérations pour réaliser l'exécution. On s'aperçoit tout de suite que l'algorithme C n'est plus utilisable. Par contre, A et B restent applicables. Maintenant, avec 100 éléments, il faut respectivement 8000 et 100000 opérations. Bien évidemment, l'algorithme A est le plus performant, mais l'algorithme B reste applicable. Cet exemple justifie la notion d'efficacité. **Si le nombre d'opérations "n'explose pas" avec une augmentation de la taille des données, l'algorithme est considéré efficace.**

Certains pourraient demander ce qu'est précisément une opération. En général, on considère comme étant une opération élémentaire une affectation, une addition, un test... Mais cela est discutable puisque selon le langage et le compilateur, une opération sera exécutée plus ou moins vite, une addition s'exécutera plus ou moins vite qu'un test... Cependant, il faut bien comprendre que l'on s'intéresse au comportement général de l'algorithme face à des problèmes de grande taille. Ainsi, ce n'est pas utile de compter toutes les opérations dans le détail, ni de considérer le langage de programmation. Dans notre exemple, les coefficients 80 et 10 ne sont pas très importants, dès que la taille augmente, on s'aperçoit que c'est le terme en n qui prime. Ceci explique la difficulté à déterminer la complexité d'un algorithme. Il faut être très précis dans la démarche mais ne pas se soucier de la valeur exacte en terme de temps d'exécution de chaque opération.

PROBLEME D' OPTIMISATION COMBINATOIRE, DE RECONNAISSANCE

Problème d'optimisation combinatoire

Un problème d'optimisation combinatoire est un problème qui consiste à chercher une meilleure solution parmi un ensemble de solutions réalisables.

Problème de reconnaissance

Un problème de reconnaissance est un problème qui consiste à apporter une réponse "oui" ou "non" à une question.

A chaque problème d'optimisation combinatoire, on peut associer un problème de reconnaissance de la manière suivante.

> Soit un problème d'optimisation combinatoire:
> Trouver $s' \in S \mid f(s') = \min\{f(s) \mid s \in S\}$.
>
> Soit a un nombre, on définit le problème de reconnaissance associé:
> Existe-t-il $s' \in S \mid f(s') \leq a$?

Un problème d'optimisation combinatoire est au moins aussi difficile que le problème de reconnaissance associé. De plus, on peut généralement prouver que le problème de reconnaissance n'est pas plus facile que le problème d'optimisation combinatoire. En d'autres termes, cela signifie qu'un problème d'optimisation combinatoire est souvent du même niveau de difficulté que le problème de reconnaissance associé. Cela justifie que la suite de ce chapitre ne concerne que les problèmes de reconnaissance.

PROBLEME FACILE, DIFFICILE

Problèmes décidables

Tout d'abord, on fait une distinction entre les problèmes décidables et les problèmes indécidables. **Les problèmes indécidables sont ceux pour lesquels aucun algorithme, quel qu'il soit, n'a été trouvé pour les résoudre.** Ainsi, les problèmes décidables sont ceux pour lesquels il existe au moins un algorithme pour les résoudre.

La classe NP

Parmi les problèmes décidables, les plus simples à résoudre sont regroupés dans la classe NP. **Un problème appartient à la classe NP si quelqu'un ayant la solution au problème peut démontrer que c' est la solution en un temps polynomial**Les autres problèmes décidables sont considérés comme très difficiles. La classe NP est également décomposée en trois

catégories qui permettent d'identifier les problèmes les plus simples et les problèmes les plus compliqués de la classe.

La classe P

La classe P, qui regroupe les problèmes les plus simples de la classe NP, contient les problèmes pour lesquels on connaît au moins un algorithme polynomial pour les résoudre. Pour le reste de la classe NP, on n'est pas sûr qu'il n'existe pas un algorithme polynomial pour résoudre chacun de ses problèmes. Ainsi, on sait que P est inclu dans NP mais on n'a pas pu prouver que P n' est pas NP.

Réduction polynomiale

Soit deux problèmes P1 et P2. On dit que P1 est réduit au problème P2 si on peut résoudre P1 en utilisant un algorithme pour P2 comme sous-routine. Cette réduction est dite polynomiale si l'algorithme pour P1 est polynomial en comptant l'appel à la sous-routine de P2 comme une opération élémentaire.

P2 est au moins aussi difficile que P1. En effet, si P2 appartient à la classe P, P1 y appartient aussi, car l'algorithme ci-dessus est polynomial. Si P2 n'est pas polynomial, rien n'empêche P1 de l'être s'il existe un algorithme polynomial pour le résoudre.

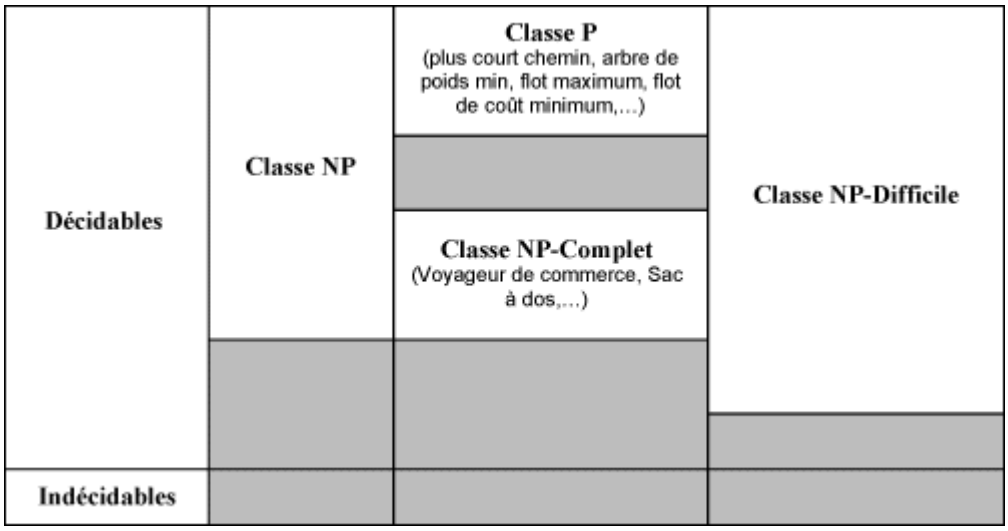
La classe NP-Complet

La classe NP-Complet regroupe les problèmes les plus difficiles de la classe NP. Elle contient les problèmes de la classe NP tels que n'importe quel problème de la classe NP leur est polynomialement réductible. Entre eux, les problèmes de la classe NP-Complet sont aussi difficiles.

La classe NP-Difficile

La classe NP-Difficile regroupe les problèmes (pas forcément dans la classe NP) tels que n'importe quel problème de la classe NP leur est polynomialement réductible.

Tableau récapitulatif



5. RECHERCHE DU PLUS COURT CHEMIN

DEFINITIONS

Réseau

Un réseau est un graphe G = (X;U) auquel on associe une fonction d: U → IR qui à chaque arc fait correspondre sa "longueur". On note R = (X;U;d) un tel réseau.

Longueur

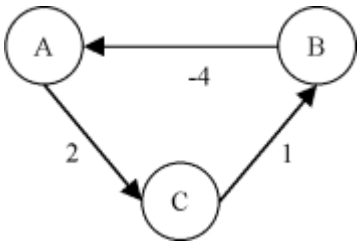
La longueur d'un chemin (d'une chaîne, d'un circuit ou d'un cycle) est la somme des longueurs de chaque arc qui le compose. Par convention, un chemin (une chaîne, un circuit ou un cycle) qui ne contient pas d'arc est de longueur nulle.

Circuit absorbant

Un circuit est absorbant si sa longueur est négative.

Exemple

La longueur du circuit suivant est -1. C'est donc un circuit absorbant.



PROBLEMES DE PLUS COURT CHEMIN

Différents problèmes peuvent être posés autour de la recherche de plus court chemin. Un premier problème A consiste à rechercher le plus court chemin entre deux points donnés. C'est-à-dire déterminer le chemin de plus petite longueur qui relie ces points. En fait, pour résoudre ce problème, on résout le problème B. Ce dernier consiste à déterminer, à partir d'un point donné le plus court chemin pour aller à tous les autres noeuds. Enfin, le problème C consiste à trouver, pour n'importe quelle paire de noeuds, le chemin le plus court entre

eux. Le problème A étant résolu par le problème B, on s'intéressera uniquement aux problèmes B et C.

Le problème B peut être résolu de nombreuses manières. Tout dépend des hypothèses émises sur la structure du réseau. Lorsque le réseau est sans circuit, on appliquera l'algorithme de Bellman. Lorsque le réseau n'a que des longueurs positives ou nulles (mais éventuellement avec des circuits), on utilisera la méthode de Dijkstra. Enfin, dans un réseau sans hypothèse particulière, on dispose d'un algorithme général. On s'intéressera ici à l'algorithme de Dijkstra.

Le problème C peut être résolu également de différentes manières. On s'intéressera ici à l'algorithme de Dantzig qui émet l'hypothèse qu'il n'y a pas de circuit absorbant dans le graphe. Il faut noter que le stockage de la solution du problème C peut prendre beaucoup de place. C'est pourquoi on se limitera ici à calculer les plus courtes distances et non pas à déterminer les plus courts chemins. Cependant, la démarche reste exactement la même.

ALGORITHME DE DIJKSTRA

Cet algorithme détermine **les plus courts chemins d'un point s à tous les autres points d'un réseau** R = (X;U;d). Il suppose que **les longueurs sur les arcs sont positives ou nulles.** L'idée de cet algorithme est de partager les noeuds en deux groupes: ceux dont on connaît la distance la plus courte au point s (ensemble S) et ceux dont on ne connaît pas cette distance (ensemble S'). On part avec tous les noeuds dans S'. Tous les noeuds ont une distance infinie (p(x) = +oo) avec le point s, excepté le point s lui-même qui a une distance nulle (p(s) = 0). A chaque itération, on choisit le noeud x qui a la plus petite distance au point s. Ce noeud est déplacé dans S. Ensuite, pour chaque successeur y de x, on regarde si la distance la plus courte connue jusqu'à lors entre s et y ne peut pas être améliorée en passant par x. Si c'est le cas, p(y) est modifiée. Ensuite, on recommence avec un autre noeud.

Algorithme

Titre: Dijkstra
Entrées: R = (X;U;d) un réseau, s un sommet.
Sorties: p() une fonction indiquant la plus courte distance qui sépare un point de s, pred() une fonction indiquant par quel arc on arrive à un point si on emprunte le plus court chemin.
Variables intermédiaires: S et S' deux sous-ensembles de sommets, x et y deux noeuds, sortie un booléen.

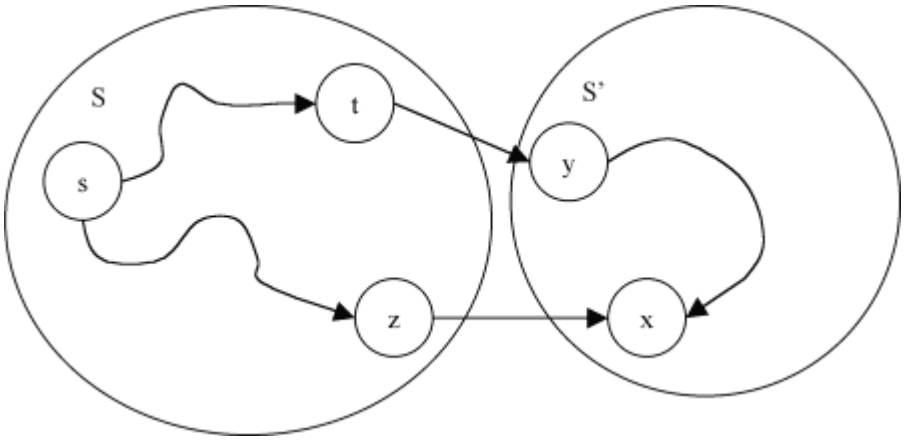
```
Début
 S ← ∅;
 S' ← X;

 pour tout x ∈ X faire
  p(x) ← +oo;
  pred(x) ← nil;
 fin pour;

 p(s) ← 0;
 sortie ← faux;

 tant que S' ≠ ∅ et non sortie faire
  choisir x ∈ S' tel que p(x) = min{p(y) | y ∈ S'};
  sortie ← (p(x) = +oo);
```

```
  S ← S ∪ {x};
  S' ← S' - {x};

  pour chaque arc u = (x;y) | y ∈ S' faire
   si p(y) > p(x) + d(u) alors
    p(y) ← p(x) + d(u);
    pred(y) ← u;
   fin si;
  fin pour;
 fin tant que;
```

Fin

Justification

Pour qu'un noeud x ait une distance p(x) différente de +oo , il faut qu'il ait un de ses prédécesseurs dans S. De plus, p(x) est la plus petite distance de s à x en empruntant uniquement les noeuds de S. Supposons que p(x) soit la plus petite distance de tout l'ensemble S'. Supposons également que le plus court chemin pour aller de s à x passe par t et y (cf. figure). Cela signifie que y est le prédécesseur de x. Il faudra donc que $p(y) \leq p(x)$. Cependant, si le chemin le plus court pour aller de s à x passe par t et y alors le chemin le plus court pour aller de s à y passe par t. Ce qui signifie que y connaît sa plus courte distance à s. Malheureusement, elle est supérieure à p(x) ce qui contredit l'hypothèse que p(x) est la plus courte distance de S'.



ALGORITHME DE DANTZIG

Cet algorithme détermine **la plus courte distance entre tous les couples de sommets d'un réseau** R = (X;U;d). On suppose que **le réseau ne contient pas de circuit absorbant**. On commence par numéroter les noeuds d'une manière quelconque. Ensuite, une matrice D de taille n x n est construite indiquant pour chaque noeud la plus courte distance qui le sépare d'un autre noeud. Chaque ligne et chaque colonne représentent donc un noeud. L'idée est de considérer un sous-réseau pour lequel on détermine les plus courtes distances entre ses noeuds. Ensuite, itérativement, on rajoute un noeud dans ce sous-réseau, on met les plus courtes distances à jour et on recommence jusqu'à ce que le sous-réseau soit le réseau lui-même.

A l'itération k, le sous-réseau a k noeuds et ce sont ceux numérotés de 1 à k. A l'itération k + 1, on ajoute le noeud k + 1 dans le sous-réseau. Les distances sont mises à jour de la manière suivante. Tout d'abord on détermine la plus courte distance entre le noeud k + 1 et

n'importe quel autre noeud. Cela s'exprime par:

$$D(k+1,i) \leftarrow \min\{d(x_{k+1},x_j) + D(j,i)\} \text{ pour } i = 1...k.$$

Cela veut dire que l'on considère que pour aller de $k+1$ à i, on passe par un des noeuds de 1 à k. On conserve la longueur du plus court chemin. De la même manière, on détermine la plus courte distance entre n'importe quel noeud et le noeud $k+1$.

$$D(i,k+1) \leftarrow \min\{D(i,j) + d(x_j,x_{k+1})\} \text{ pour } i = 1...k.$$

Enfin, on met à jour la plus courte distance de chaque noeud de 1 à k vers chaque noeud de 1 à k car peut être que le plus court chemin entre deux de ces noeuds passent par $k+1$.

$$D(i,j) \leftarrow \min\{D(i,j) , D(i,k+1) + D(k+1,j)\} \text{ pour } i, j = 1...k.$$

Algorithme

Titre: Dantzig
Entrées: $R = (X;U;d)$ un réseau.
Sorties: D une matrice contenant les plus courtes distances.
Variables intermédiaires: i, j et k des entiers.

Début

```
pour i ← 1 à n faire
 pour j ← 1 à n faire
  si i = j alors D(i,i) ← 0;
  sinon D(i,j) ← +oo;
 fin pour;
fin pour;

pour k ← 1 à n faire
 pour i ← 1 à k - 1 faire
  D(k,i) ← min{d(xk,xj) + D(j,i) | (xk;xj) ∈ U};
 fin pour;

 pour i ← 1 à k - 1 faire
  D(i,k) ← min{D(i,j) + d(xj,xk) | (xj;xk) ∈ U};
 fin pour;

 pour i ← 1 à k - 1 faire
  pour j ← 1 à k - 1 faire
   D(i,j) ← min{D(i,j) , D(i,k) + D(k,j)};
  fin pour;
 fin pour;
fin pour;
```

Fin

6. ORDONNANCEMENT, RECHERCHE DU PLUS LONG CHEMIN

PROBLEMES D' ORDONNANCEMENT

Les problèmes d'ordonnancement sont apparus au départ dans la planification de grands projets. Le but était de gagner du temps sur leur réalisation. De tels projets sont constitués de nombreuses étapes, également appelées tâches. Des relations temporelles existent entre ces dernières. Par exemple:

- Une étape doit commencer à une date précise;
- Un certain nombre de tâches doivent être terminées pour pouvoir en démarrer une autre;
- Deux tâches ne peuvent être réalisées en même temps (elles utilisent une même machine par exemple);
- Chaque tâche nécessite une certaine quantité de main d'oeuvre. Il faut donc éviter, à chaque instant, de dépasser la capacité totale de main d'oeuvre disponible.

Toutes ces contraintes ne sont pas simples à prendre en compte dans la résolution du problème. Ici, nous allons nous intéresser uniquement aux deux premiers types de contraintes. On cherchera à déterminer une planification, un ordonnancement des étapes qui minimise le temps total de réalisation du projet. A partir de cette planification, nous verrons que le temps de certaines étapes peut éventuellement être modifié sans entraîner un retard du projet, alors que d'autres, les tâches dites "critiques", retardent entièrement le projet au moindre retard local.

METHODE PERT

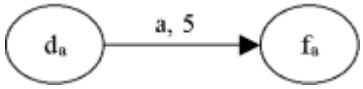
Il existe deux grandes méthodes pour résoudre le problème énoncé ci-dessus. Il y a la méthode américaine CPM (*Critical Path Method*) avec sa variante PERT (*Program Evaluation and Review Technique*) et la méthode française MPM (Méthode des Potentiels). Nous nous intéresserons ici uniquement à la méthode PERT. En voici les grandes étapes.

Représentation sous la forme d'un graphe

Imaginons qu'un problème nous soit énoncé sous la forme suivante. Un projet est décomposé en 7 étapes (a, b ... g). La durée de chaque étape a été estimée et certaines contraintes de succession ont été établies (cf. tableau ci-après). Il s'agit maintenant de trouver la planification, i.e. les dates de démarrage de chaque étape, qui minimise le temps total de réalisation du projet, et d'indiquer dans ce cas les étapes critiques.

	Durée	Opérations antérieures
a	1	aucune
b	1	aucune
c	2	b
d	3	a
e	2	a
f	2	e
g	2	d,c

La méthode PERT préconise de représenter le problème sous la forme d'un graphe. Les noeuds vont représenter des événements et les arcs des durées séparant ces événements. Tous d'abord, on défini deux sommets D et F qui représentent respectivement le début et la fin du projet. Ensuite, chaque étape est décomposée en deux événements (i.e. deux noeuds), représentant le début et la fin de l'étape, et une durée (i.e. un arc).

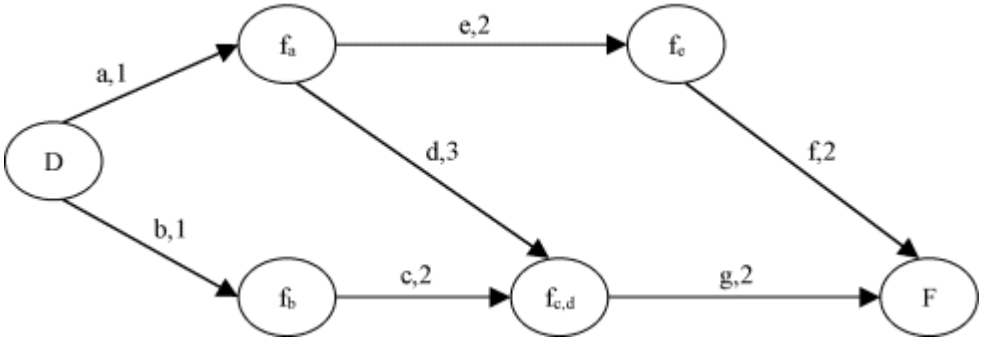
Etape a de durée 5: 

La succession de deux étapes consistera à ajouter une étape fictive de durée nulle pour indiquer que l'événement de fin de la première étape et l'événement de début de la seconde apparaissent au même instant.

a précède b: 

Dans ce cas précis, on peut fusionner les noeuds f_a et d_b pour simplifier le graphe.

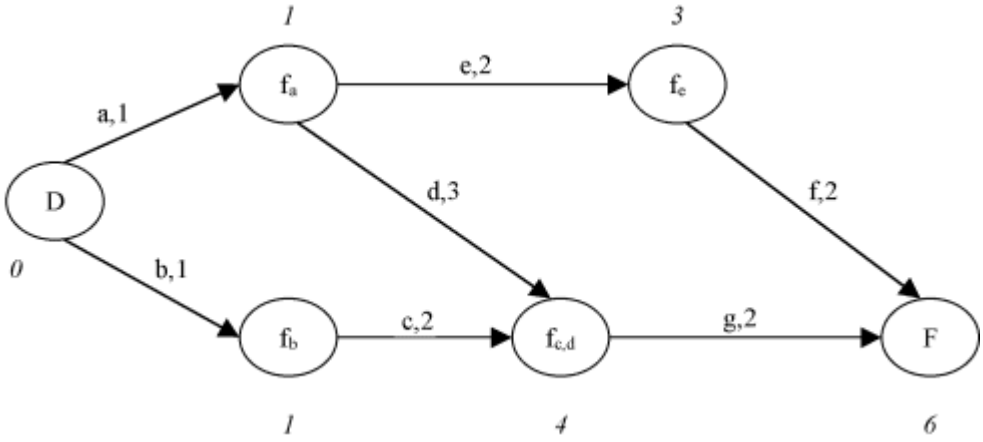
Finalement, on obtiendra le graphe suivant.



Détermination des dates au plus tôt

Ensuite, pour chaque événement on détermine la date à partir de laquelle il peut au plus tôt se produire. On considère la date de l'événement D comme étant 0. Ainsi, la date au plus tôt d'un événement, c'est le temps minimum qu'il faut pour que toutes les tâches antérieures

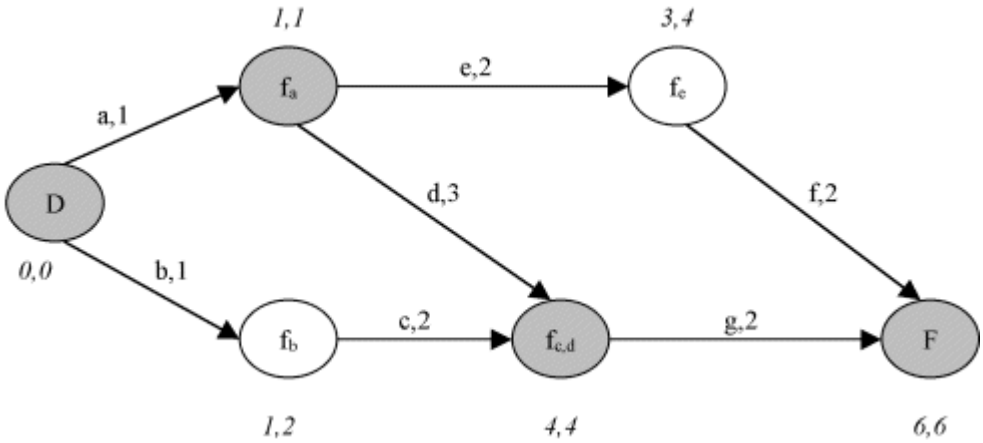
soient effectuées. Toutes les tâches antérieures sont effectuées signifie qu'en suivant n'importe quel chemin en partant de D, sa longueur (i.e. la durée totale des tâches parcourues) est inférieure à la date de l'événement concerné. Autrement dit, on recherche la plus longue durée qui sépare l'événement de D. Cela revient à rechercher un plus long chemin entre D et l'événement (cf. algorithme de Bellman ci-après). Dans notre exemple, on obtient:



Finalement, on obtient pour chaque événement la date à laquelle il peut apparaître au plus tôt. Ce qui fournit la planification des tâches qui aboutit à la plus petite durée possible du projet. Maintenant, à partir de cette planification, on tente de déterminer les tâches qui ne doivent absolument pas être retardées par rapport à la planification obtenue sous peine de voir le projet entièrement retardé.

Détermination des dates au plus tard

De la même manière, on va tenter de déterminer la date à laquelle un événement doit au plus tard arriver pour que le projet se termine dans les temps prévus. Pour cela, connaissant la date de fin d'une étape, on est capable de déterminer la date au plus tard à laquelle l'étape doit commencer, il s'agit de la date de fin de l'étape moins sa durée. Par exemple, l'événement F doit se produire à la date 6, cela signifie que la tâche f de durée 2 doit démarrer au plus tard à la date 4 = 6 - 2. De même, quand un événement est le point de départ de plusieurs étapes, on prendra tout naturellement la date au plus tard la plus petite parmi les dates au plus tard de chaque étape. On s'aperçoit que cela revient à rechercher la durée la plus longue (i.e. le chemin le plus long) qui sépare un événement de l'événement de fin F. Cette durée est ensuite soustraite à la durée totale du projet pour fournir la date au plus tard de chaque événement. Dans notre exemple, on obtient:



Analyse et identification des tâches critiques

Une fois la date au plus tôt et la date au plus tard de chaque événement trouvées, on peut analyser la situation. La date au plus tôt fournie la date planifiée pour chaque événement. Ensuite, la date au plus tard indique de combien un événement peut être retardé sans retarder le projet complet. Cela identifie les étapes critiques. Il est à noter que sur le plus long chemin entre D et F, toutes les étapes sont critiques. Ici, les étapes critiques sont a, d et g. Une étape est critique si la date de début et la date de fin est critique et si la différence entre ces dates est égale à la durée de la tâche. En d'autres termes, une tâche est critique si les intervalles des dates de début et de fin sont tels qu'ils ne permettent pas une augmentation de la durée de la tâche.

ALGORITHME DE BELLMAN

La détermination des dates au plus tôt et des dates au plus tard est basée sur la recherche d'un plus long chemin. Comme **le graphe considéré ne comporte pas de circuit**, un algorithme comme celui de Bellman, plus simple que celui de Dijkstra, est très bien adapté. Il est à noter que la méthode de Bellman fonctionne aussi bien pour un plus court chemin que pour un plus long chemin, contrairement à celle de Dijkstra qui est valide uniquement pour les plus courts chemins.

L'idée de l'algorithme de Bellman pour la recherche de plus court chemin (ou de plus long chemin) est tout à fait semblable à celle de Dijkstra. La différence réside dans le choix du noeud à chaque nouvelle itération. Dans l'algorithme de Dijkstra, c'est le noeud avec la plus courte distance qui est choisi alors que dans celui de Bellman, on choisit un noeud dont tous les prédécesseurs ont déjà été traités, d'où la nécessité de ne pas avoir de circuit. Donc, sans plus de détail, voici l'algorithme de Bellman pour la recherche d'un plus court chemin.

Algorithme

Titre: Bellman
Entrées: R = (X;U;d) un réseau, s un sommet.
Sorties: p() une fonction indiquant la plus courte distance qui sépare un point de s, pred() une fonction indiquant par quel arc on arrive à un point si on emprunte le plus court chemin.
Variables intermédiaires: S et S' deux sous-ensembles de sommets, x et y deux noeuds, sortie un booléen.

Début

```
S ← {s};
S' ← X - {s};

pour tout x ∈ X faire
 p(x) ← +oo;
 pred(x) ← nil;
fin pour;

p(s) ← 0;
sortie ← faux;

tant que ∃ x ∈ S' | tous ses prédécesseurs ∈ S faire
 S ← S ∪ {x};
```

```
  S' ← S' - {x};

  pour chaque arc u = (y;x) | y ∈ S faire
   si p(x) > p(y) + d(u) alors
   [Pour le plus long chemin, remplacer par p(x) < p(y) + d(u).]
    p(x) ← p(y) + d(u);
    pred(x) ← y;
   fin si;
  fin pour;
 fin tant que;
```

Fin

---

7. RECHERCHE DU FLOT MAXIMUM

DEFINITIONS

Réseau de transport

Un réseau de transport est un graphe sans boucle, où chaque arc est associé à un nombre $c(u) \geq 0$, appelé "capacité de l'arc u". En outre, un tel réseau vérifie les hypothèses suivantes.

- Il existe un seul noeud s qui n'a pas de prédécesseurs, tous les autres en ont au moins un. Ce noeud est appelé l'entrée du réseau, ou la source.
- Il existe également un seul noeud p qui n'a pas de successeurs, tous les autres en ont au moins un. Ce noeud est appelé la sortie du réseau, ou le puits.

Flot

Un flot f dans un réseau de transport est une fonction qui associe à chaque arc u une quantité $f(u)$ qui représente la quantité de flot qui passe par cet arc, en provenance de la source et en destination du puits.

Un flot doit respecter la règle suivante: la somme des quantités de flot sur les arcs entrants dans un noeud (autre que s et p) doit être égale à la somme des quantités de flot sur les arcs sortants de ce même noeud. En d'autres termes, la quantité totale de flot qui entre dans un noeud est égale à la quantité totale de flot qui en sort.

Flot compatible

Un flot f est compatible avec un réseau si pour tout arc u, $0 \leq f(u) \leq c(u)$. Autrement dit, pour chaque arc, le flot qui le traverse ne doit pas dépasser la capacité de l'arc.

Flot complet

Un flot f est complet si pour tout chemin allant de la source au puits, il y a au moins un arc saturé, i.e. le flot qui le traverse est égal à la capacité de l'arc.

PROBLEME DU FLOT MAXIMUM

Connaissant les capacités des arcs d'un réseau de transport, le problème du flot maximum

consiste à trouver quelle est la quantité maximum de flot qui peut circuler de la source au puits. L'algorithme le plus connu pour résoudre ce problème est celui de Ford et Fulkerson. Nous verrons deux approches pour cette méthode. La première est basée sur la recherche d'une chaîne dans le réseau alors que la seconde construit un graphe "d'écart" dans lequel on recherche un chemin.

ALGORITHME DE FORD-FULKERSON, CHAINE AUGMENTANTE

On part d'un flot compatible. Le plus évident est le flot nul, i.e. pour tout arc u, $f(u) = 0$. Ensuite, on cherche une chaîne reliant la source au puits telle que son flot peut être augmenté. Si on n'en trouve pas, le problème est résolu. Sinon, on augmente le flot sur cette chaîne. Ensuite, on recommence à chercher une chaîne augmentante et ainsi de suite. Une chaîne augmentante, i.e. une chaîne pour laquelle le flot peut être augmenté est une chaîne pour laquelle les arcs dans le sens direct n'ont pas atteint leur limite maximum et les arcs en sens indirect ont un flot non nul qui les traverse. L'augmentation de flot maximum pour une chaîne est le minimum des écarts entre le flot courant et le flot maximal pour les arcs directs ou le flot courant pour les arcs indirects. Autrement dit, une chaîne C est augmentante si:

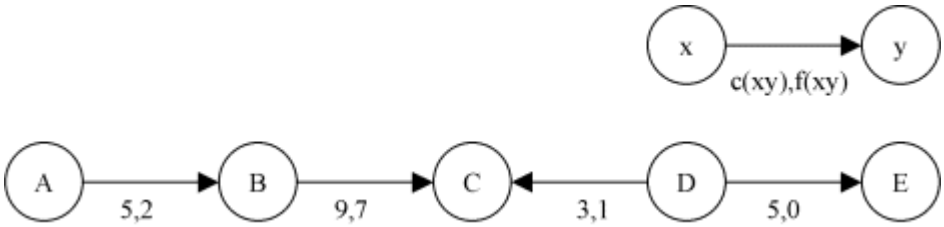
- pour tout arc u direct, $f(u) < c(u)$,
- pour tout arc u indirect, $f(u) > 0$.

Le flot sur cette chaîne C peut être augmenté de:

$$\min(\{c(u) - f(u) \mid u \in C \text{ et } u \text{ sens direct}\} \cup \{f(u) \mid u \in C \text{ et } u \text{ sens indirect}\})$$

Exemple

Voici une chaîne augmentante de A à E faisant partie d'un réseau de transport.

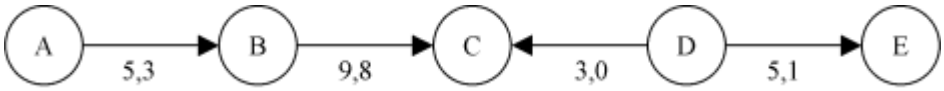


Dans cette chaîne, on peut augmenter le flot de:

- 3 entre A et B,
- 2 entre B et C,
- 1 entre C et D,
- 5 entre D et E.

On augmentera donc de 1 le flot dans cette chaîne. Ce qui signifie:

- augmenter de 1 le flot entre A et B,
- augmenter de 1 le flot entre B et C,
- diminuer de 1 le flot entre D et C,
- augmenter de 1 le flot entre D et E.



On remarque que pour les arcs en sens inverse, augmenter le flot signifie réduire le flot dans le sens direct. Entre D et C, le flot est réduit de 1 pour permettre l'arrivée d'une unité de flot sur C par B tout en conservant l'équilibre du noeud. D ayant une unité de trop, son équilibre n'est pas respecté. C'est pourquoi une unité de flot supplémentaire circule entre D et E.

Algorithme principal

Titre: FordFulkerson1
Entrées: R = (X;U;c) un réseau, s et p deux sommets.
Sorties: f() une fonction indiquant le flot circulant à travers chaque arc.
Variables intermédiaires: C un sous-ensemble d'arcs, u un arc, m et a deux réels.

Début

```
pour tout arc u ∈ U faire f(u) ← 0;

tant que ∃ une chaîne C augmentante entre s et p ∈ R faire
 m ← +oo;

 pour tout u ∈ C faire
  si u en sens direct alors a ← c(u) - f(u) sinon a ← f(u);
  si a < m alors m ← a;
 fin pour;

 pour tout u ∈ C faire
  si u en sens direct alors f(u) ← f(u) + m sinon f(u) ← f(u) - m;
 fin pour;
fin tant que;
```

Fin

Algorithme de recherche d'une chaîne augmentante

L'idée est la même que pour la recherche d'un chemin (cf. méthode avec le graphe d'écart). On parcours le réseau jusqu'à visiter le noeud p. Cependant, on utilisera un arc pour se déplacer:

- s'il est direct et qu'il n'a pas atteint son flot maximum,
- ou s'il est indirect et qu'il n'a pas un flot nul.

Algorithme

Titre: RechercherChaineAugmentante
Entrées: R = (X;U;c) un réseau, s et p deux sommets, f() le flot courant.
Sorties: pred() une fonction indiquant par quel arc on arrive à un noeud donné à partir de s.
Variables intermédiaires: X' un sous-ensemble de noeuds, accessible() une fonction qui indique si un noeud est accessible à partir de s, x et y deux noeuds.

Début

```
X' ← {s};

pour tout x ∈ X faire
 accessible(x) ← faux;
 pred(x) ← nil;
fin pour;

marqué(s) ← vrai;

tant que X' ≠ ∅ et non accessible(p) faire
 choisir x ∈ X';
 X' ← X' - {x};

 pour tout u = (x;y) ∈ U faire
  si non accessible(y) et f(u) < c(u) alors
   X' ← X' ∪ {y};
   pred(y) ← x;
   accessible(y) ← vrai;
  fin si;
 fin pour;

 pour tout u = (y;x) ∈ U faire
  si non accessible(y) et f(u) > 0 alors
   X' ← X' ∪ {y};
   pred(y) ← x;
   accessible(y) ← vrai;
  fin si;
fin tant que;
```

Fin

ALGORITHME DE FORD-FULKERSON, GRAPHE D' ECART

Comme pour la méthode précédente, on part d'un flot compatible. Ensuite, on construit un graphe d'écart à partir de ce flot. Ce graphe d'écart représente les modifications de flot possibles sur chaque arc. Sur ce graphe, les noeuds ont exactement la même signification que dans le réseau de transport. Par contre, un arc indiquera de combien il est possible d'augmenter le flot entre deux noeuds. Ainsi, pour un arc u = (x;y), on créera dans le graphe d'écart:

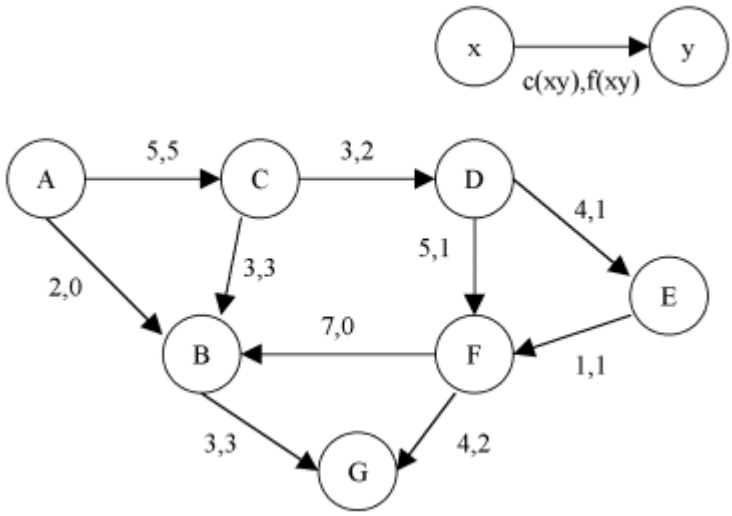
- un arc de x à y de capacité c'((x;y)) = c(u) - f(u) si c(u) ≠ f(u),
- un arc de y à x de capacité c'((y;x)) = f(u) si f(u) ≠ 0.

Ensuite, dans ce graphe d'écart, on cherchera un chemin de la source au puits. Si on n'en trouve pas, le problème est résolu. Sinon, on augmente le flot sur ce chemin, qui correspond en fait à une chaîne si l'on se ramène à la première méthode. Le flot sera augmenté de la plus petite capacité des arcs du chemin. Autrement dit, le chemin C sera augmenté de:

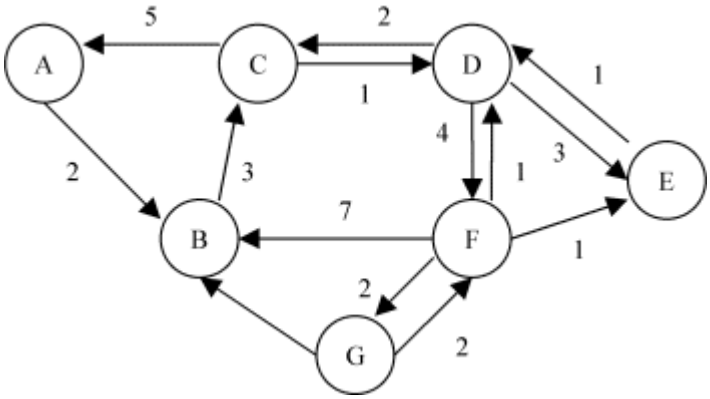
$\min\{c'(u) \mid u \in C\}$

Exemple

Voici un réseau transport dans lequel circule un flot.



Le graphe d'écart correspondant est le suivant.



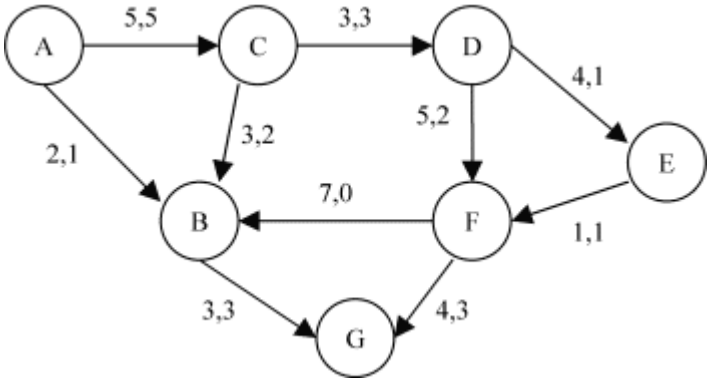
(A,B,C,D,F,G) est un chemin pour aller de A à G. On peut augmenter le flot de:

- 2 entre A et B,
- 3 entre B et C,
- 1 entre C et D,
- 4 entre D et F,
- 2 entre F et G.

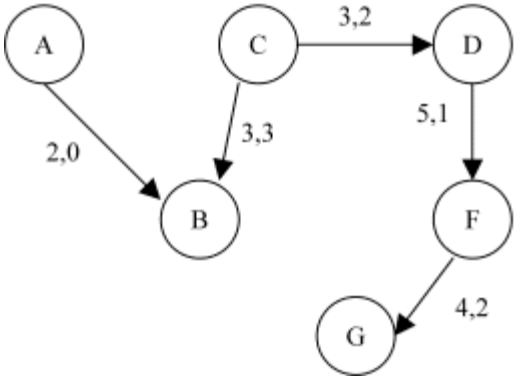
On augmentera donc le flot de 1 sur ce chemin, ce qui signifie:

- augmenter de 1 entre A et B,
- réduire de 1 entre C et B,

- augmenter de 1 entre C et D,
- augmenter de 1 entre D et F,
- augmenter de 1 entre F et G.



On remarque que le chemin (A,B,C,D,F,G) trouvé correspondait à la chaîne augmentante (A,B,C,D,F,G).



Algorithme principal

Titre: FordFulkerson2
Entrées: R = (X;U;c) un réseau, s et p deux sommets.
Sorties: f() une fonction indiquant le flot circulant à travers chaque arc.
Variables intermédiaires: G = (X;U') un graphe, c' () une fonction indiquant la capacité d'un arc de G, a() une fonction indiquant à quel arc de R correspond un arc de G, C un sous-ensemble d'arcs, u un arc, m un réel.

Début

```
pour tout arc u ∈ U faire f(u) ← 0;
construire le graphe d'écart G;

tant que ∃ un chemin C entre s et p ∈ G faire
 m ← +oo;
 pour tout u ∈ C faire si c'(u) < m alors m ← c'(u);

 pour tout u ∈ C faire
  si a(u) est direct alors f(a(u)) ← f(a(u)) + m;
  sinon f(a(u)) ← f(a(u)) - m;
 fin pour;
```

```
  construire le graphe d'écart G;
 fin tant que;
```
Fin

Construction du graphe d'écart

Pour construire le graphe d'écart G, il suffit de parcourir tous les arcs du réseau R et de créer pour chacun un ou deux arcs dans G selon la méthode expliquée précédemment.

Algorithme

Titre: ConstruireGrapheEcart
Entrées: R = (X;U;c) un réseau, f() le flot courant sur R.
Sorties: G = (X;U') le graphe d'écart, c'() une fonction indiquant la capacité d'un arc de G, a () une fonction indiquant à quel arc de R correspond un arc de G.
Variables intermédiaires: u un arc, x et y deux noeuds, m un réel.

Début
```
 pour tout u = (x;y) ∈ U faire
  si f(u) > 0 alors
   U' ← U' ∪ {(y;x)};
   c'((y;x)) ← f(u);
   a((y;x)) ← u;
  fin si;

  si f(u) < c(u) alors
   U' ← U' ∪ {(x;y)};
   c'((x;y)) ← c(u) - f(u);
   a((x;y)) ← u;
  fin si;
 fin pour;
```
Fin

Recherche d'un chemin

Pour rechercher un chemin entre s et p, il suffit de parcourir le graphe dans le sens des arcs et de marquer les noeuds que l'on visite. On ne visitera pas deux fois un même noeud. Pour effectuer ce parcours, on dispose d'un ensemble X' des noeuds restant à visiter. Au départ il n'y a que s. Ensuite, à chaque itération, on prend un noeud dans X', on le marque pour éviter de le visiter à nouveau, on l'enlève de X' et on met ses successeurs dans X', seulement s'ils ne sont pas déjà marqués. On s'arrête quand on a visité p.

Algorithme

Titre: RechercherChemin
Entrées: R = (X;U;c) un réseau, s et p deux sommets, f() le flot courant.
Sorties: pred() une fonction indiquant par quel arc on arrive à un noeud donné à partir de s.
Variables intermédiaires: X' un sous-ensemble de noeuds, accessible() une fonction qui indique si un noeud est accessible à partir de s.

Début

```
X' ← {s};

pour tout x ∈ X faire
 accessible(x) ← faux;
 pred(x) ← nil;
fin pour;

accessible(s) ← vrai;

tant que X' ≠ ∅ et non accessible(p) faire
 choisir x ∈ X';
 X' ← X' - {x};

 pour tout u = (x;y) ∈ U faire
  si non accessible(y) alors
   X' ← X' ∪ {y};
   pred(y) ← x;
   accessible(y) ← vrai;
  fin si;
 fin pour;
fin tant que;
```

Fin

8. PROGRAMMATION LINEAIRE

PRESENTATION

La programmation linéaire est un outil très puissant de la recherche opérationnelle. C'est un outil générique qui peut résoudre un grand nombre de problèmes. En effet, une fois un problème modélisé sous la forme d'équations linéaires, des méthodes assurent la résolution du problème de manière exacte. On distingue dans la programmation linéaire, la programmation linéaire en nombres réels, pour laquelle les variables des équations sont dans IR^+ et la programmation en nombres entiers, pour laquelle les variables sont dans IN. Bien entendu, il est possible d'avoir les deux en même temps. Cependant, la résolution d'un problème avec des variables entières est nettement plus compliquée qu'un problème en nombres réels.

Une des méthodes les plus connues pour résoudre des programmes linéaires en nombre réels est la méthode du Simplex. En théorie, elle a une complexité non polynomiale et est donc supposée peu efficace. Cependant, en pratique, il s'avère au contraire qu'il s'agit d'une bonne méthode.

De plus, de nombreux logiciels intégrant cette méthode existent. Certains sont utilisés via une interface graphique alors que d'autres permettent une communication par fichiers ce qui autorise l'u tilisation du programme de manière cachée dans le développement d'un autre logiciel.

Programme linéaire

La programmation linéaire permet la résolution d'un programme linéaire. Un programme linéaire est un système d'équations ou d'inéquations appelées "contraintes" qui sont linéaires (c'est-à-dire que les variables ne sont pas élevées au carré, ne servent pas d'exposant, ne sont pas multipliées entre elles...). Et à partir de ces contraintes, on doit optimiser une fonction également linéaire appelée objectif.

Exemples

Contraintes linéaires:

$$5x_1 - 2x_2 + 4x_3 = 8$$
$$x_1 + 3x_2 + 8x_3 \geq 25$$
$$9x_1 + 6x_2 - 3x_3 \leq 17$$

Contraintes non linéaires:

$$5x_1^2 + 2x_2 + 4x_3 = 8$$

$x_1x_2 + 8x_3 \geq 25$

Objectifs:

$\max: z = 3x_1 - 2x_2 + 8x_3$ (signifie maximiser z)
$\min: z = -3x_1 + x_3$ (signifie minimiser z)

Forme canonique

Pour résoudre un programme linéaire de manière automatique, il faut qu'il ait une certaine forme que l'on appelle "canonique". Dans ce cours, on a choisi la forme canonique suivante: un programme linéaire n'a que des contraintes d'infériorité et l'on tente de maximiser la fonction objectif. De plus toutes les variables sont positives.

Exemple

Le programme linéaire suivant est sous forme canonique.

$\max: z = 3x_1 - 2x_2 + 8x_3$

sous:
$5x_1 - 2x_2 + 4x_3 \leq 8$
$x_1 + 3x_2 + 8x_3 \leq 25$
$9x_1 + 6x_2 - 3x_3 \leq 17$
$x_1 , x_2 , x_3 \geq 0$

Transformations

Tout programme linéaire quelconque peut être ramené à une forme canonique. Voici quelques exemples de transformations possibles.

Si une variable x_1 est négative, on la remplace par une variable positive $x_1' = -x_1$. Par exemple:

$\max: z = 3x_1 - 2x_2 + 8x_3$		$\max: z = 3x_1 - 2x_2 - 8x_3'$
sous:		sous:
$5x_1 - 2x_2 + 4x_3 \leq 8$	$\Rightarrow$	$5x_1 - 2x_2 - 4x_3' \leq 8$
$x_1 + 3x_2 + 8x_3 \leq 25$		$x_1 + 3x_2 - 8x_3' \leq 25$
$9x_1 + 6x_2 - 3x_3 \leq 17$		$9x_1 + 6x_2 + 3x_3' \leq 17$
$x_1 , x_2 \geq 0$ et $x_3 \leq 0$		$x_1 , x_2 , x_3' \geq 0$

Si une variable n'a pas de contrainte de signe, on la remplace par deux variables positives x_1'

et x_1'' telles que $x_1 = x_1' - x_1''$. Par exemple:

$$\begin{array}{ll} \max: z = 3x_1 - 2x_2 + 8x_3 & \max: z = 3x_1 - 2x_2 + 8x_3' - 8x_3'' \\ \text{sous:} & \text{sous:} \\ 5x_1 - 2x_2 + 4x_3 \le 8 & 5x_1 - 2x_2 + 4x_3' - 4x_3'' \le 8 \\ x_1 + 3x_2 + 8x_3 \le 25 \quad \Rightarrow & x_1 + 3x_2 + 8x_3' - 8x_3'' \le 25 \\ 9x_1 + 6x_2 - 3x_3 \le 17 & 9x_1 + 6x_2 - 3x_3' + 3x_3'' \le 17 \\ x_1 , x_2 \ge 0 & x_1 , x_2 , x_3' , x_3'' \ge 0 \end{array}$$

Si le programme linéaire a une contrainte de supériorité, on la remplace par une contrainte d'infériorité en inversant le signe des constantes. Par exemple:

$$\begin{array}{ll} \max: z = 3x_1 - 2x_2 + 8x_3 & \max: z = 3x_1 - 2x_2 + 8x_3 \\ \text{sous:} & \text{sous:} \\ 5x_1 - 2x_2 + 4x_3 \le 8 & 5x_1 - 2x_2 + 4x_3 \le 8 \\ x_1 + 3x_2 + 8x_3 \le 25 \quad \Rightarrow & x_1 + 3x_2 + 8x_3 \le 25 \\ 9x_1 + 6x_2 - 3x_3 \ge 17 & -9x_1 - 6x_2 + 3x_3 \le -17 \\ x_1 , x_2 , x_3 \ge 0 & x_1 , x_2 , x_3 \ge 0 \end{array}$$

Si le programme linéaire a une contrainte d'égalité, on la remplace par deux contraintes équivalentes, l'une d'infériorité, l'autre de supériorité. Les variables du programme doivent satisfaire ces deux contraintes, ce qui revient alors à l'égalité de départ. Par exemple:

$$\begin{array}{ll} \max: z = 3x_1 - 2x_2 + 8x_3 & \max: z = 3x_1 - 2x_2 + 8x_3 \\ \text{sous:} & \text{sous:} \\ 5x_1 - 2x_2 + 4x_3 \le 8 & 5x_1 - 2x_2 + 4x_3 \le 8 \\ x_1 + 3x_2 + 8x_3 \le 25 \quad \Rightarrow & x_1 + 3x_2 + 8x_3 \le 25 \\ 9x_1 + 6x_2 - 3x_3 = 17 & 9x_1 + 6x_2 - 3x_3 \le 17 \\ x_1 , x_2 , x_3 \ge 0 & 9x_1 + 6x_2 - 3x_3 \ge 17 \\ & x_1 , x_2 , x_3 \ge 0 \end{array}$$

$$\Rightarrow \quad \begin{array}{l} \max: z = 3x_1 - 2x_2 + 8x_3 \\ \text{sous:} \\ 5x_1 - 2x_2 + 4x_3 \le 8 \\ x_1 + 3x_2 + 8x_3 \le 25 \\ 9x_1 + 6x_2 - 3x_3 \le 17 \\ -9x_1 - 6x_2 + 3x_3 \le -17 \\ x_1 , x_2 , x_3 \ge 0 \end{array}$$

Application

Une entreprise fabrique 2 produits X et Y. Pour sa conception, chaque produit fini nécessite 3 produits intermédiaires A, B et C. Pour fabriquer un produit X, on a besoin de 2 produits A, de 2 produits B et de 1 produit C. De même, pour fabriquer un produit Y, on a besoin de 3 produits A, de 1 produit B et de 3 produits C. En outre, l'entreprise dispose d'une quantité limitée de produits A, B et C. Elle a 180 produits A, 120 produits B et 150 produits C. Sachant que le prix de revient de X est 3 francs et que celui de Y est de 4 francs, combien de produits X et Y faut-il fabriquer pour maximiser le profit ?

On modélise ce problème par un programme linéaire. Soit x et y les quantités de produits X et Y fabriqués.

La quantité totale de produits A utilisée est $2x + 3y$. Cette quantité ne doit pas dépasser 180, d'où la première contrainte.

$$2x + 3y \leq 180$$

De même, pour les produits B et C, on obtient:

$$2x + y \leq 120$$
$$x + 3y \leq 150$$

Bien entendu, les quantités x et y sont positives.

$$x , y \geq 0$$

Enfin, on tente de maximiser le profit qui est le total des bénéfices sur la vente des produits X plus celui des produits Y.

$$\max: 3x + 4y$$

Le programme linéaire est donc le suivant.

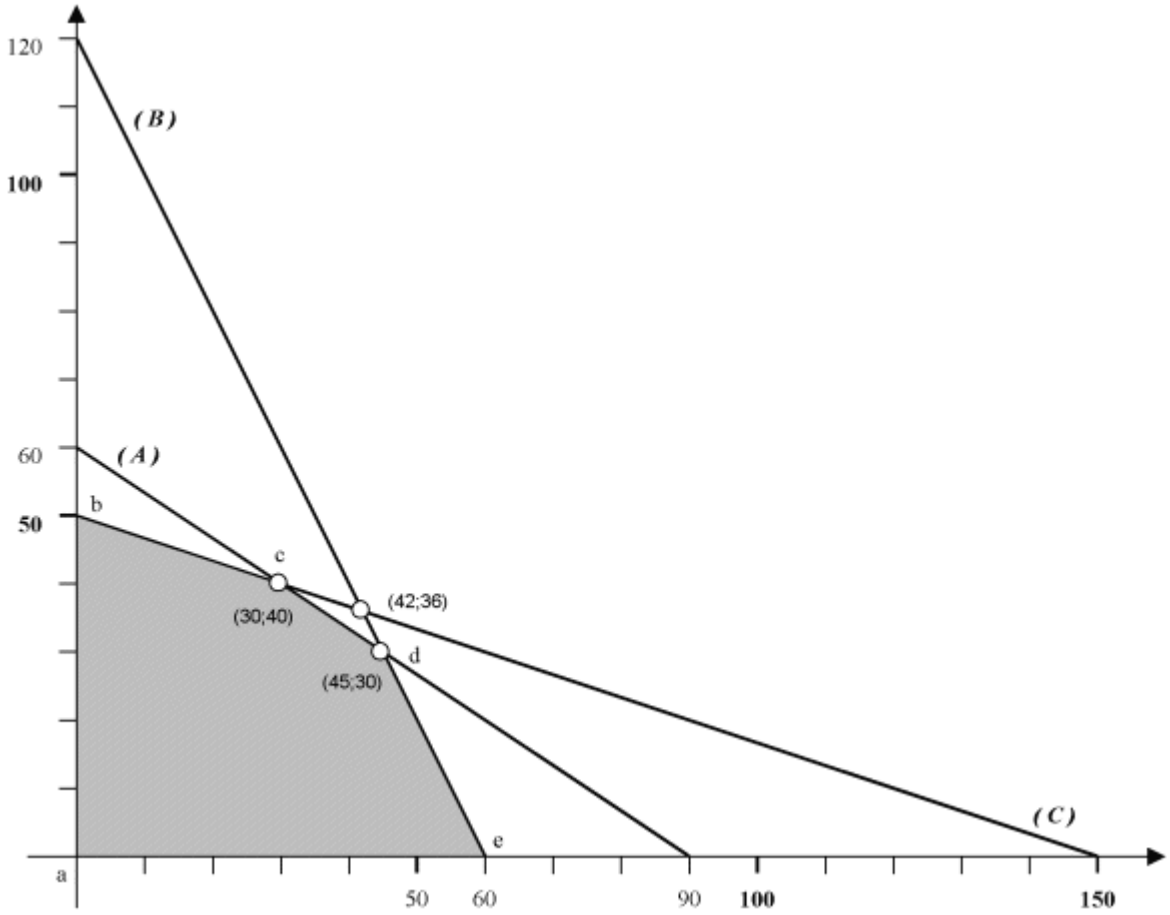
$$\max: z = 3x + 4y$$

sous:

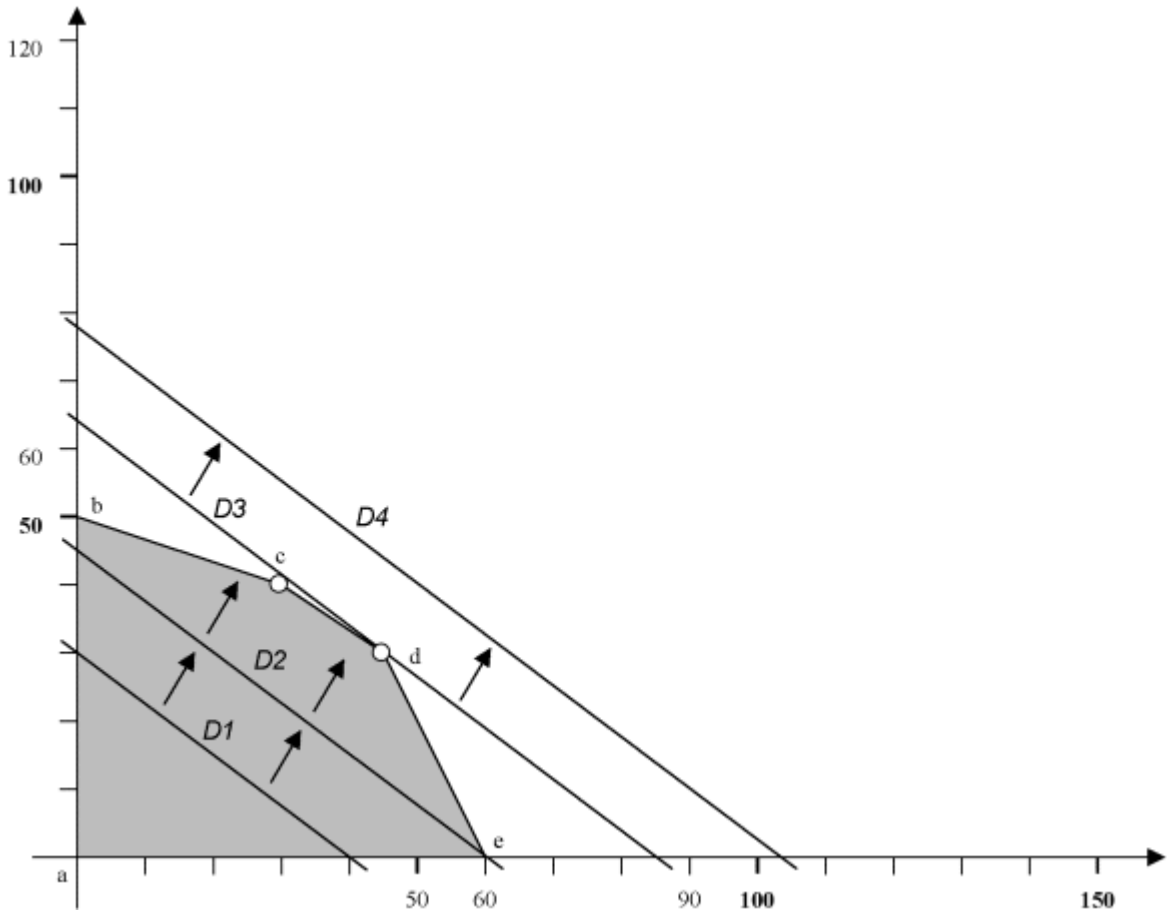
$$2x + 3y \leq 180 \quad \text{(A)}$$
$$2x + y \leq 120 \quad \text{(B)}$$
$$x + 3y \leq 150 \quad \text{(C)}$$
$$x , y \geq 0$$

REPRESENTATION GRAPHIQUE

On peut représenter le problème dans un espace à deux dimensions.



Les solutions admissibles sont représentées par la zone grisée (a,b,c,d,e). Ce sont les solutions qui satisfont les contraintes. Considérons maintenant la fonction z = 3x + 4y. Pour z = 120, on a une droite D1, 3x + 4y = 120, qui représente les solutions pour lesquelles le profit vaut 120. Si z = 180, on a une autre droite D2, 3x + 4y = 180, qui représente les solutions pour lesquelles le profit vaut 180. On remarque que D2 est parallèle à D1 et on s'aperçoit facilement qu'en déplaçant la droite vers le haut on augmente le profit z. Donc pour résoudre graphiquement le problème, on va faire "glisser" la droite vers le haut jusqu'à ce qu'elle ait un minimum de points communs avec la surface grisée. Le point restant ici est d. Il représente la solution pour laquelle le profit est maximum. En effet, si on prend un profit plus important, représenté par exemple par D4, on s'aperçoit que toutes les solutions sont en dehors de la surface grisée. Donc la solution du problème est de produire 45 produits X et 30 produits Y pour obtenir le profit maximum de z = 3x + 4y = 255 francs.



LA METHODE DU SIMPLEX

Globalement, la méthode du Simplex va se déplacer le long de la forme (a,b,c,d,e), de sommet en sommet jusqu'à trouver le meilleur point. Pour essayer de mieux comprendre comment fonctionne cette méthode, considérons le problème de production précédent. Dans un premier temps, les contraintes sont ramenées à des égalités en introduisant de nouvelles variables de la manière suivante.

max: $z = 3x + 4y$

sous:

$2x + 3y = 180 - u$ (A)
$2x + y = 120 - v$ (B)
$x + 3y = 150 - w$ (C)
$x , y , u , v , w \geq 0$

Ensuite, on va se placer sur le point $a = (0;0)$ qui est une solution admissible du problème. Dans un cas plus général, il faut noter que trouver une solution admissible pour démarrer la méthode du Simplex n'est pas forcément évident. On a alors:

$x = 0$ $u = 180$
$y = 0$ $v = 120$
$w = 150$

En regardant $z = 3x + 4y$, on s'aperçoit que la moindre augmentation de x ou de y augmente

le profit z. Donc on va augmenter l'une des deux variables au maximum. Par exemple, prenons y et utilisons les contraintes pour exprimer y en fonction de toutes les autres variables.

$y = 60 - 1/3u - 2/3x$	$y \leq 60$	(A)
$y = 120 - v - 2x$	$y \leq 120$	(B)
$y = 50 - 1/3w - 1/3x$	$y \leq 50$	(C)

Comme toutes les variables sont positives, on peut en déduire, à partir de chaque contrainte, une borne maximum pour y (cf. ci-dessus). y doit satisfaire les 3 contraintes. Ici la plus grande valeur de y possible est 50. A l'aide de la contrainte C, on remarque que x doit rester à 0 et que w est forcé à 0. Maintenant, les variables nulles sont x et w. Quant aux autres variables, on s'arrange pour les exprimer uniquement avec x et w de manière à obtenir leur valeur. On fait de même pour le profit z.

$y = 50 - 1/3x - 1/3w$	$= 50$	(C)
$u = 180 - 2x - 3y = 30 - x + w$	$= 30$	(A)
$v = 120 - 2x - y = 70 - 5/3x + 1/3w$	$= 70$	(B)
x	$= 0$	
w	$= 0$	
$z = 3x + 4y = 200 + 5/3x - 4/3w$	$= 200$	

On se trouve donc au point b = (x;y) = (0;50) avec un profit z = 200.

Maintenant, on reprend le même raisonnement que pour le point a = (0;0). Le profit est exprimé par $z = 200 + 5/3x - 4/3w$. En augmentant x, on augmente le profit. On regarde de combien on peut augmenter x.

$x = 30 - u + w$	$x \leq 30$	(A)
$x = 3/5\ (70 - v + 1/3w) = 42 - 3/5v + 1/5w$	$x \leq 42$	(B)
$x = 3\ (50 - y - 1/3w) = 150 - 3y - w$	$x \leq 150$	(C)

Ici, on prendra x = 30, la valeur maximum qu'il peut atteindre. Les variables qui sont nulles sont u et w d'après la contrainte A. On calcule les nouvelles valeurs des autres variables.

$x = 30 - u + w$	$= 30$	(A)
$v = 70 - 5/3\ (30 - u + w) + 1/3w = 20 + 5/3u - 4/3w$	$= 20$	(B)
$y = 50 - 1/3\ (30 - u + w) - 1/3w = 40 + 1/3u - 2/3w$	$= 40$	(C)
u	$= 0$	
w	$= 0$	
$z = 200 + 5/3\ (30 - u + w) - 4/3w = 250 - 5/3u + 1/3w$	$= 250$	

On se trouve donc au point c = (x;y) = (30;40) avec un profit z = 250.

On recommence encore une fois le même raisonnement. Le profit est exprimé par $z = 250 - 5/3u + 1/3w$. En augmentant w, on augmente le profit. On regarde de combien on peut augmenter w.

$w = -30 + x + u$	$w \geq -30$	(A)
$w = 3/4\ (20 + 5/3u - v) = 15 + 5/4u - 3/4v$	$w \leq 15$	(B)

$w = 3/2\ (40 - y + 1/3u) = 60 - 3/2y + 1/2u \quad w \leq 60 \quad (C)$

Ici, on prendra $w = 15$, la valeur maximum qu'il peut atteindre. Les variables qui sont nulles sont u et v d'après la contrainte B. On calcule les nouvelles valeurs des autres variables.

$w = 15 + 5/4u - 3/4v$	$= 15$	(B)
$x = 30 - u + (15 + 5/4u - 3/4v) = 45 + 1/4u - 3/4v$	$= 45$	(A)
$y = 40 + 1/3u - 2/3\ (15 + 5/4u - 3/4v) = 30 - 1/2u + 1/2v$	$= 30$	(C)
u	$= 0$	
v	$= 0$	
$z = 250 - 5/3u + 1/3\ (15 + 5/4u - 3/4v) = 255 - 5/4u - 1/4v$	$= 255$	

On se trouve donc au point $d = (x;y) = (45;30)$ avec un profit $z = 255$.

On reprend encore une fois le même raisonnement. Le profit est exprimé par $z = 255 - 5/4u - 1/4v$. Ni u, ni v ne permettent d'augmenter le profit z. Cela signifie que l'on a obtenu le profit maximum. La solution recherchée est alors représentée par d le point courant. On aboutit bien à la même conclusion que par la représentation graphique. Il faut produire 45 produits X et 30 produits Y pour obtenir un profit maximum de 255 francs.
