Ingénierie de la cognition : Algorithmes Ad-Hoc

A.Revel revel@ensea.fr

1 Introduction

L'objectif de ce cours est de vous faire d couvrir les techniques permettant d'implémenter des systèmes « intelligents » sur la base d'une conception orientée vers la résolution de problèmes particuliers (Algorithmes ad-hoc : plus court chemin, flots, ordonnancement...).

2 Algorithmes

Selon le dictionnaire de l'informatique, on appelle algorithme

> Un jeu de règles ou de procédures bien défini qu'il faut suivre pour obtenir la solution d'un problème dans un nombre fini d'étapes. Un algorithme peut comprendre des procédures et instructions algébriques, arithmétiques, et logiques, et autres. Un algorithme peut être simple ou compliqué. Cependant un algorithme doit obtenir une solution en un nombre fini d'étapes. Les algorithmes sont fondamentaux dans la recherche d'une solution par voie d'ordinateur, parce que l'on doit donner à un ordinateur une série d'instructions claires pour conduire à une solution dans un temps raisonnable.

Cette définition étant posée, on arrive naturellement à la notion de complexité algorithmique.

2.1 Notion de complexité algorithmique

Les questions qui sous-tendent la notion de complexité algorithmiques sont les suivantes :
- Comment qualifier un algorithme ?
- Combien de temps prend il ?
- Cet algorithme est il meilleur que cet autre ?
- Cet algorithme est-il applicable ?
- A-t'on un problème d'explosion combinatoire ? (problèmes NP-complets)
- Aura t'on un problème de ressources ?

2.1.1 Introduction : exemple du tri par insertion

Ce type tri consiste à insérer à la bonne place dans un tableau rangé les valeurs que l'on veut trier. Si, de plus, le tri se fait *sur place*, l'algorithme correspondant au tri du tableau A est donc le suivant :

1. pour $j \leftarrow 2$ à longueur[A]
   (a) faire $cle \leftarrow A[j]$
   (b) $i \leftarrow j - 1$
   (c) tant que $i > 0$ et $A[i] > cle$
      i. faire $A[i + 1] \leftarrow A[i]$

ii. $i \leftarrow i - 1$

(d) $A[i+1] \leftarrow cle$

A partir du deuxième élément, on considère chaque élément comme la clef que l'on souhaite insérer. Au regarde ensuite dans les éléments qui le précède celui qui est plus petit et on décale les autres à partir de cet élément.

A partir de cet exemple, nous allons introduire les notions d'*analyse* d'algorithme.

- Analyser un algorithme permet de déterminer le temps de calcul et/ou les ressources nécessaires pour le faire fonctionner. Cela a surtout un intérêt pour pouvoir comparer différents algorithmes.
- On paramètre le temps d'exécution d'un algorithme par rapport à la taille de son entrée. La *taille d'entrée*, dépend en fait du type d'algorithme utilisé (longueur d'un tableau, nombre de bits significatifs...)
- Le temps d'exécution dépend du nombres d'opérations élémentaires réalisées et du coût unitaire constant c_i de l'opération i.

Tri-Insertion(A)	coût	fois
pour $j \leftarrow 2$ à longueur[A]	c_1	n
faire $cle \leftarrow A[j]$	c_2	$n-1$
$i \leftarrow j-1$	c_3	$n-1$
tant que $i > 0$ et $A[i] > cle$	c_4	$\sum_{j=2}^{n} t_j$
faire $A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^{n}(t_j - 1)$
$i \leftarrow i-1$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$A[i+1] \leftarrow cle$	c_7	$n-1$

Les commentaires ont un coût de calcul nul puisqu'ils ne sont pas exécutés.

t_j est le temps au bout duquel la condition $A[i] > cle$ n'est plus réalisée.

Le temps nécessaire pour exécuter l'algorithme sur un tableau de taille n est donc :

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n}(t_j - 1) + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7(n-1)$$

Regardons le cas le plus favorable ou *meilleur cas*. C'est celui qui correspond à un tableau déjà trié. Dans ce cas $t_j = 1$.

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

Le temps d'exécution est une *fonction linéaire* du nombre d'entrée.

Dans le *pire des cas*, le tableau est trié à l'envers. Dans ce cas, $t_j = j$.

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} j + c_5 \sum_{j=2}^{n}(j-1) + c_6 \sum_{j=2}^{n}(j-1) + c_7(n-1)$$

Rappelons-nous que :

$$\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$$

On a donc :

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n+1)}{2}\right) + c_6\left(\frac{n(n+1)}{2}\right) + c_7(n-1)$$

Ou encore :

$$T(n) = \left(\frac{c_4 + c_5 + c_6}{2}\right) n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4 - c_5 - c_6}{2} + c_7\right) n - (c_2 + c_3 + c_4 + c_7)$$

Qui est une *fonction quadratique* des entrées.

En analyse de complexité on étudie toujours le pire cas puisque cela donne une borne supérieure de la complexité de l'algorithme.

Par ailleurs, le cas moyen et le pire cas sont souvent de complexité équivalente. Considérons par exemple le tri par insertion. En moyenne, la moitié des éléments sont mal rangés. On teste donc la moitié des éléments du tableaux avant l'insertion, d'où $t_j = \frac{j}{2}$. Le temps d'exécution de l'algorithme est donc une fonction quadratique, tout autant que pour le pire cas !

En fait, lorsqu'on étudie la complexité d'un algorithme, on ne s'intéresse pas au temps de calcul réel nécessaire pour un nombre d'entrées donné mais à un *ordre de grandeur* de ce temps de calcul. Pour une complexité polynomiale, par exemple, on ne s'intéressera qu'au terme de plus grand ordre. Le temps de calcul de l'algorithme de tri par insertion, par exemple, est dans le pire de cas en $\Theta(n^2)$. Ainsi, un algorithme en $\Theta(n \log n)$ est plus rapide que cet algorithme.

2.1.2 Notations asymptotiques

Notation $\Theta(n)$

On note $\Theta(g(n))$ l'ensemble des fonctions :

$$\Theta(g(n)) = \left\{ f(n) : \exists\, (c_1, c_2) \in \Re^{*+} \text{ et } n_0 \text{ tq } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0 \right\}$$

Cela revient à dire qu'asymptotiquement, $f(n)$ est égale à $g(n)$ à un facteur constant près. On dit que $g(n)$ est une *borne approchée asymptotiquement* de $f(n)$. On borne la fonction $f(n)$ à la fois *par excès* et *par défaut*.

Exercice 1 *Trouver c_1 et c_2 satisfaisant l'inégalité dans le cas du tri par insertion.*

Exercice 2 *Montrer que si $f(n) = \Theta(kg(n))$ alors $f(n) = \Theta(g(n))$.*

Notation $O(n)$

On note $O(g(n))$ l'ensemble des fonctions :

$$O(g(n)) = \left\{ f(n) : \exists\, (c, n_0) \in \Re^{*+} \text{ tq } 0 \leq f(n) \leq c \cdot g(n)\ \forall n \geq n_0 \right\}$$

Ce qui peut s'interpréter comme : certain multiple de $g(n)$ est une borne supérieure de $f(n)$. On dit que $g(n)$ est une *borne supérieure asymptotique* de $f(n)$.
On remarquera que $f(n) = \Theta(g(n))$ implique $f(n) = O(g(n))$.

Exercice 3 *montrer que toute fonction linéaire se trouve dans* $O(n^2)$.

Notation $\Omega(n)$

On note $\Omega(g(n))$ l'ensemble des fonctions :

$$\Omega(g(n)) = \left\{ f(n) : \exists\, (c, n_0) \in \Re^{*+} \text{ tq } 0 \leq c \cdot g(n) \leq\ f(n)\ \forall n \geq n_0 \right\}$$

Ce qui peut s'interpréter comme : certain multiple de $g(n)$ est une borne inférieure de $f(n)$. On dit que $g(n)$ est une *borne inférieure asymptotique* de $f(n)$.

Exercice 4 *montrer que le tri par insertion est en* $\Omega(n)$ *mais aussi en* $\Omega(n^2)$ *dans le pire des cas.*

Théorème 1 *Pour 2 fonctions quelconques* $f(n)$ *et* $g(n)$, $f(n) = \Theta(g(n))$ *ssi* $f(n) = O(g(n))$ *et* $f(n) = \Omega(g(n))$.

ATTENTION : il est dangereux d'additionner trop hâtivement des bornes approchées asymptotiquement !

Notation $o(n)$

La borne supérieure asymptotique peut être ou non asymptotiquement approchée (ex. $2n^2 = O(n^2)$ est approchée asymptotiquement alors que $2n = O(n^2)$ ne l'est pas). On note $o(g(n))$ le fait que la borne supérieure ne soit pas asymptotiquement approchée.
$o(g(n))$ représente donc l'ensemble des fonctions :

$$o(g(n)) = \left\{ f(n) : \exists\, (c, n_0) \in \Re^{*+} \text{ tq } 0 \leq f(n) <\ c \cdot g(n)\ \forall n \geq n_0 \right\}$$

Ainsi :
$2n = o(n^2)$ mais $2n^2 \neq o(n^2)$.

Il en résulte qu'à l'infini, la fonction $f(n)$ devient négligeable par rapport à la fonction $g(n)$:

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0$$

Notation $\omega(n)$

De même, la borne inférieure asymptotique peut être ou non asymptotiquement approchée. On note alors $\omega(g(n))$ le fait que la borne inférieure ne soit pas asymptotiquement approchée.
$\omega(g(n))$ représente donc l'ensemble des fonctions :

$$\omega(g(n)) = \left\{ f(n) : \ \exists\, (c, n_0) \in \Re^{*+} \text{ tq } 0 \leq c \cdot g(n) < \ f(n) \ \forall n \geq n_0 \right\}$$

Ainsi :
$\frac{n^2}{2} = \omega(n)$ mais $\frac{n^2}{2} \neq \omega(n^2)$.
Il en résulte qu'à l'infini, la fonction $f(n)$ devient négligeable par rapport à la fonction $g(n)$:

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = +\infty$$

2.1.3 Propriétés

2.1.4 Transitivité (vrai pour toutes les notations)

$$f(n) = \Theta(g(n)) \text{ et } g(n) = \Theta(h(n)) \Longrightarrow f(n) = \Theta(h(n))$$

Réflexivité (vrai pour Θ, O et Ω)

$$f(n) = \Theta(f(n))$$

Symétrie

$$f(n) = \Theta(g(n)) \text{ ssi } g(n) = \Theta(f(n))$$

Symétrie transposée

$$f(n) = O(g(n)) \text{ ssi } g(n) = \Omega(f(n))$$

$$f(n) = o(g(n)) \text{ ssi } g(n) = \omega(f(n))$$

Trichotomie

$\forall (a, b) \in \Re^2$, une seule de ces propositions est valide : $f(n) = o(g(n)), f(n) = \Theta(g(n)), f(n) = \omega(g(n))$

2.1.5 Résolution pour un algorithme récursif

Dans le cas d'un algorithme récursif, le temps d'exécution de l'algorithme pour n entrées est exprimé en fonction du temps d'exécution de ce même algorithme pour un nombre d'entrées réduit. La forme générale à résoudre est la suivante :

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Avec $a \geq 1$ et $b \geq 1$, et $f(n)$ un fonction positive asymptotiquement.
Cela revient en fait à considérer que la récurrence divise le problème en a sous-problèmes de taille $\frac{n}{b}$, le coût dû à la division du problème correspondant à une fonction $f(n)$.

Théorème 2 *Dans le cadre définit ci-dessus, $T(n)$ peut alors être borné de la manière suivante :*

1. *Si $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ pour un $\varepsilon > 0$ donné, alors $T(n) = \Theta\left(n^{\log_b a}\right)$*
2. *Si $f(n) = \Theta\left(n^{\log_b a}\right)$, alors $T(n) = \Theta\left(n^{\log_b a} \ln n\right)$*
3. *Si $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$ pour un $\varepsilon > 0$ donné, et si $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ pour $c < 1$ et tous les n suffisamment grands, alors $T(n) = \Theta(f(n))$*

Exercice 5 *Déterminer les bornes asymptotiques pour les récurrences suivantes :*

1. $T(n) = 4T(\frac{n}{2}) + n$
2. $T(n) = 4T(\frac{n}{2}) + n^2$
3. $T(n) = 4T(\frac{n}{2}) + n^3$

2.1.6 Exemple du tri par TAS

L'idée est d'utiliser les propriétés de la structure de TAS pour construire un tableau rangé de valeurs.

Procédure Entasser

Cette procédure est appelée pour construire la structure de TAS. A l'appel, on considère que les arbres GAUCHE(I) et DROIT(I), ont des structures de TAS. Cependant, $A[i]$, peut être plus petit que ses fils, ce qui viole la propriété des TAS. Le rôle de la procédure est de propager si nécessaire la valeur de $A[i]$ dans le TAS de manière que le sous-arbre enraciné en i devienne un TAS.

Entasser(A,i)

1. $l \leftarrow$ GAUCHE(I)
2. $r \leftarrow$ DROIT(I)
3. **si** $l \leq taille[A]$ *et* $A[l] > A[i]$
   alors $max \leftarrow l$
   sinon $max \leftarrow i$
4. **si** $r \leq taille[A]$ *et* $A[r] > A[max]$
   alors $max \leftarrow r$
5. **si** $max \neq i$
   alors *échanger* $A[i]$ *et* $A[max]$
   ENTASSER(A,MAX)

Complexité On décompose le temps l'algorithme en 2 :

1. le temps de calcul nécessaire pour permuter le max avec la racine (si besoin est).
2. le temps de calcul pour appliquer l'algorithme sur le sous-arbre qui ne respecte pas la structure de TAS dans le pire des cas.

Le temps de permutation est constant, quelque soit le nombre de données. La complexité est donc en $\Theta(1)$.
Le pire des cas pour Entasser est d'appliquer la procédure à un TAS entièrement déséquilibré, c'est-à-dire dont le sous-arbre gauche est un arbre complet d'une hauteur supérieure à celui du sous-arbre droit. On peut montrer que dans ce cas, le nombre de données à traiter est inférieur à $\frac{2n}{3}$.
Le temps de calcul peut donc s'exprimer sous forme récursive :

$$T(n) = T\left(\frac{2n}{3}\right) + \Theta(1)$$

En appliquant directement le théorème général avec $f(n) = \Theta(1)$, $a = 1$ et $b = \frac{3}{2}$, on a $f(n) = \Theta(n^0) = \Theta(n^{\log_b a})$. D'où :

$$T(n) = O(\ln n)$$

Construction d'un TAS

Pour convertir un tableau quelconque en TAS, il suffit d'utiliser la procédure ENTASSER en sens inverse en remontant à partir des feuilles qui sont considérées comme des TAS à 1 élément.

Construire-Tas(A)

1. *taille[A]*$\leftarrow$ *longueur[A]*
2. *pour* $i \leftarrow \frac{longeur[A]}{2}$ *à 1*
   faire ENTASSER(A,I)

Complexité Il y a n appels à Entasser, donc Construire-Tas est en $O(n \ln n)$.

Exercice 6 *En fait, on peut montrer que Construire-Tas est en $O(n)$ en calculant la somme des calculs effectués.*

Algorithme de tri par TAS

L'idée de cet algorithme de tri consiste à construire un TAS à partir d'un tableau. Ensuite, on sait que l'élément maximum est à la racine de l'arbre ; il suffit donc de l'échanger avec le dernier élément du tableau et de décrémenter d'une unité la taille de celui-ci. Comme la propriété de TAS peut être violée, il faut utiliser la procédure ENTASSER sur le nœud considérer et réitérer l'algorithme.

Trier-Tas(A)

1. CONSTRUIRE-TAS(A)
2. *pour* $i \leftarrow longeur[A]$ *à 2*
   faire échanger $A[1]$ *et* $A[i]$
   $taille[A] \leftarrow taille[A] - 1$
   ENTASSER(A,1)

Complexité La procédure Trier-Tas fait un appel à Construire-Tas, qui est en $O(n)$ et n appels à Entasser.
La complexité du tri par TAS (heap-sort) est donc en $O(n \ln n)$.

Utilisation sur les files de priorité

La structure de données en tas est souvent utilisée pour gérer des systèmes de files d'attente comme on trouve dans les ordonnanceurs de systèmes d'exploitation.
Une *file de priorité* est une structure permettant de gérer un ensemble d'éléments associés chacun à une *clé*. Les opérations valides sur une file sont :

Insérer(S,x) Insère l'élément x dans l'ensemble S

Maximum(S) Recherche l'élément maximum de l'ensemble S

Extraire-Max(S) Retourne et supprime de la file l'élément maximum de S

Exercice 7
1. *Écrire la fonction* EXTRAIRE-MAX-TAS(A)
2. *Écrire la fonction* INSÉRER-TAS(A,CLÉ)

Réponse :

Extraire-Max-Tas(A)

1. *si* $taille[A] < 1$
   alors erreur débordement
2. $max \leftarrow A[1]$
3. $A[1] \leftarrow A[taille[A]]$
4. $taille[A] \leftarrow [A] - 1$
5. ENTASSER(A,1)
6. *retouner max*

La procédure INSÉRER-TAS(A,CLÉ) incrémente la taille du TAS d'une unité puis cherche l'endroit où insérer la clé.

Insérer-Tas(A)

1. $taille[A] \leftarrow taille[A] + 1$
2. $i \leftarrow taille[A]$
   tant que $i > 1$ *et* $A[pere(i)] < cle$
   faire $A[i] \leftarrow A[pere(i)]$
   $i \leftarrow pere(i)$
3. $A[i] \leftarrow cle$

3 Notion de récursivité

3.1 Introduction

Définition

On appelle récursive une fonction faisant appel à elle même.

Exemple : la suite de Fibonacci

Elle est définie par :

$$U_0 = U_1 = 1$$
$$U_n = U_{n-1} + U_{n-2}$$

Quelle est la valeur de U_4 ?

Intérêt

- Concision des algorithmes
- Analyse plus « naturelle » de certains problèmes qui se prêtent plus au raisonnement récursif, et dont la résolution par une méthode itérative serait beaucoup plus complexe
- Moins de variables locales nécessaires.

Par contre :

- L'exécution d'un programme récursif demande plus de place mémoire et plus de temps à cause de la répétition des appels.

3.2 Typologie

3.2.1 Fonction aux paramètres définis récursivement

Fonction de Ackerman

La fonction de Ackerman est la fonction définie ci-dessous.

```
int ack(int m, int n)
{
   if(m == 0) return n + 1;
   else
      if(n == 0)
        return ack(m - 1, 1);
      else
        return ack(m - 1, ack(m, n - 1));
}
```

C'est fonction est excessivement récursive... à tel point que $Ack(5,1)$ n'est pas calculable !

3.2.2 Fonction mutuellement récursives

Ce sont des couples (ou plus) de fonctions qui s'appellent les unes les autres.

Linéarisation de cosinus

$$\cos(n \cdot x) = \sin x \cdot \cos(n-1)x + \cos x \cdot \sin(n-1)x$$
$$\sin(n \cdot x) = \cos x \cdot \cos(n-1)x - \sin x \cdot \sin(n-1)x$$

3.2.3 Fractales

Fractale est un mot inventé par Benoit Mandelbrot en 1974 sur la racine latine fractus qui signifie brisé. Fractal était au départ un adjectif : les objets fractals. On nomme fractale (nom féminin) une courbe ou surface de forme irrégulière ou morcelée qui se crée en suivant des règles déterministes ou stochastiques.

Tapis de Sierpinski

C'est un carré dont on évide récursivement la partie centrale.

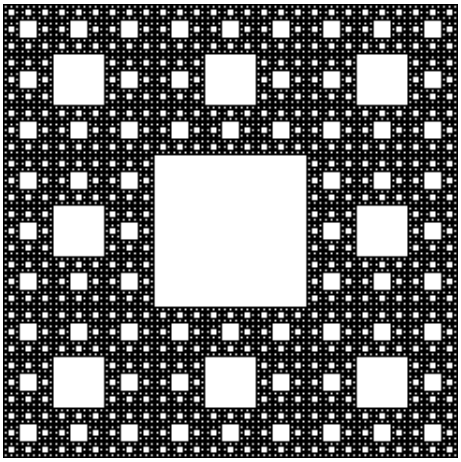

Fig. 1. *Tapis de Sierpinski*

Ensemble de Mandelbrot

Cet ensemble est défini comme le lieu des points où la suite $z_{n+1} = z_n^2 + c$ converge, avec c un point du plan complexe.

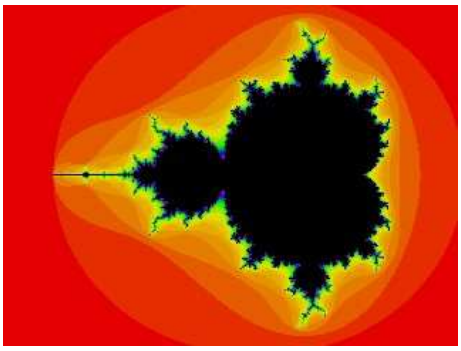

Fig. 2. *Ensemble de Mandelbrot*

3.3 Problèmes classiques définis récursivement

3.3.1 Le problème du porte-monnaie

Connaissant une somme s et un ensemble d'entiers positifs, on veut déterminer, lorsque c'est possible, un sous ensemble dont la somme (cumulée) des éléments vaut s. Les entiers sont rangés dans un tableau T.

```
int sommeExacte(int t, int i)
{
   return t==0 ?1 :
      t<0||i==n ?0 : sommeExacte(t-T[i],i+1) ?
          (printf(" %d ",T[i]),1) :
           sommeExacte(t,i+1) ;
}
```

3.3.2 Les tours de Hanoï

Principe

Un jeu comporte 3 tours A, B, et C. La tour A comporte n disques superposés du plus grand au plus petit. Comment faire migrer les n disques de A vers C en respectant le fait qu'un disque de grande taille ne peut être posé sur un disque plus petit ?

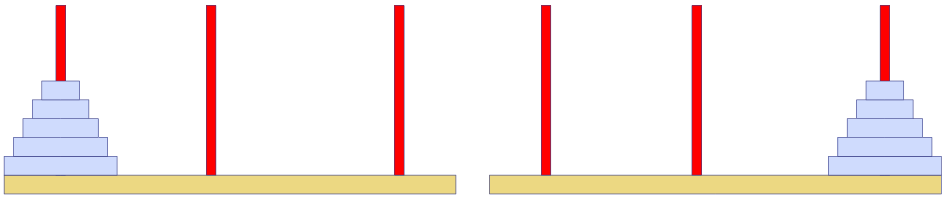

Fig. 3. *Tour de Hanöi. A gauche, la position de départ, à droite, celle d'arrivée.*

Comment formaliser le problème ?

- Déplacer n disques de A vers C c'est
  - Déplacer n-1 disques de A vers B
  - Déplacer 1 disque de A vers C
  - Déplacer n-1 disques de B vers C

Algorithme

```
Hanoi(A,B,C,n)
{
   if (n==0) return
       Hanoi(A,C,B,n-1)
       Déplacer(A,C)
       Hanoi(B,A,C,n-1)
}
```

3.3.3 Placement des reines sur un échiquier

Principe

On se pose le problème de savoir si l'on peut placer n reines sur un échiquier de taille n^2. Et si oui, comment ?

Comment formaliser le problème ?

L'idée est, encore une fois, de parcourir l'ensemble des solutions possibles. Pour les valeurs successives de i, on place une reine sur la ligne i et sur une colonne $j = pos[i]$ en vérifiant bien qu'elle n'est pas en prise. Le tableau *pos* que l'on remplit récursivement contient les positions des reines déjà placées.

Algorithme

```
procedure Reines (i: integer)
     begin
     if i > Nreines then
          Imprimer_Solution
     else
          begin
          for j:= 1 to Nreines do
          if Compatible (i,j) then
```

```
            begin
            pos[i] := j;
            Reines(i+1)
            end;
        end;
    end;
```

3.4 Dérécursification

3.4.1 Introduction

On l'a vu, la récursivité est souvent facile à mettre en place mais elle consomme beaucoup de ressources. Il est donc parfois nécessaire, pour des raisons de performances de dérécursifier un problème posé naturellement de manière récursive.

3.4.2 Formalisation

On considère une action récursive A (exprimée sous la forme d'une fonction) définie de la fa on suivante :

```
fonction A (X : un param)
{
        si base(X) alors alpha(X)
        sinon
beta(X);
                A(T(X));
                gamma(X)
}
```

La fonction T de transformation des paramètres de la fonction A permet de caractériser la liste des appels récursifs engendrés. Elle doit permettre une convergence vers le cas de *base* pour que l'action A se termine.

3.4.3 Principe de la dérécursification

Le principe de la transformation consiste à interpréter l'exécution d'un appel $A(X)$ comme la succession de trois actions :

1. itération de parcours de la liste des appels dans laquelle on applique l'action *beta* chacune des valeurs de paramètres.
2. application de l'action *alpha*
3. itération de parcours de la liste en sens inverse des appels et pour laquelle on applique l'action *gamma* à chacune des valeurs de paramètres.

Le problème principal consiste à retrouver la liste inverse des paramètres. 2 cas sont alors envisageables.

T est inversible

On suppose que l'on dispose d'une fonction T_{inv} inverse de la fonction T.

```
fonction A(X : un param)
pc <- X;
c <- 1
tant que non base(pc)
   beta(pc);
   pc <- T(pc);
   c <- c + 1;
alpha(pc)
tant que c != 1
   c <- c - 1;
   pc <- Tinv(pc);
   gamma(pc)
```

T n'est pas inversible

Si la fonction T_{inv} n'est pas définie, on peut utiliser la technique suivante :
– On conserve les valeurs de la fonction T au fur et à mesure de l'itération.
– On utilise alors ces valeurs dans l'ordre inverse lors de la deuxième itération.
– On utilise pour cela une pile P d'éléments de type param.

```
fonction A(X)
pc <- X
P <- []
tant que non base(pc)
   beta(pc)
   P <= pc
   pc <- T(pc)
alpha(pc)
tant que P !=[]
   pc <= P
   gamma(pc)
```

3.4.4 Exemple : la fonction factorielle

Cette fonction est définie récursivement par :

```
fonction factorielle (n : un entier > 0)
{
   si n == 1 alors retourner(1)
   sinon
    retourner(n*factorielle(n-1))
}
```

On identifie :
- $base(x) = (x == 1)$
- $alpha(x) = return1$
- $beta(x) = x*$
- $T(x) = (n-1)$ dont l'inverse $T_{inv}(x) = x+1$
- $gamma(x) =$

On obtient donc :

```
fonction factIter (n : un entier > 0)
{
   pc <- n; c <- 1
   tant que non (pc=1)
      f <- f * pc;
      pc <- pc - 1;
      c <- c + 1
   tant que c !=  1
      c <- c - 1;
      pc <- pc + 1
   retourner(f)
}
```

En simplifiant l'écriture on obtient :

```
fonction factIter (n : un entier > 0)
{
   pc <- n;
   tant que non (pc=1)
      f <- f * pc;
      pc <- pc - 1;
   retourner(f)
}
```

4 Graphes et Arbres

4.1 Introduction

Les premiers travaux en théorie des graphes sont dus à Euler (1736) auquel les habitants de Könisberg demandèrent s'il était possible de traverser chacun des 7 ponts de la ville une fois exactement et revenir à sa position de départ. En posant le problème sous la forme d'un graphe, Euler montra alors l'impossibilité de le résoudre.

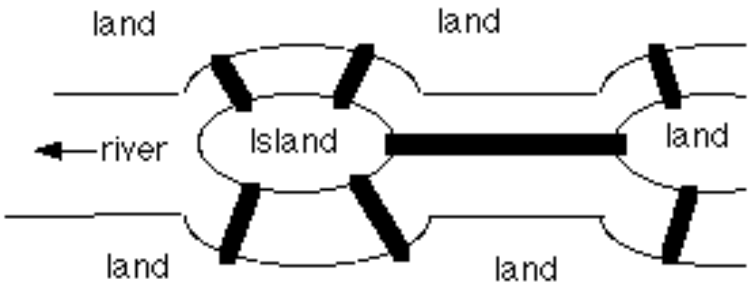

Fig. 4. *Le problème d'Euler*

Par la suite, la théorie des graphes a été adaptée à la formalisation de nombreux problèmes que nous verrons plus tard :
- planification de transport
- découverte du plus court chemin
- recherche de solution
- câblage de circuits imprimés
- ordonnancement de projets
- réseaux de neurones
- techniques de renforcement
- représentation de chaînes de Markov
- architectures de réseaux
- etc.

4.2 Définitions

4.2.1 Graphes non-orientés

graphe non-orienté : $G = (S, A)$, où S est un ensemble de *sommets* et A un ensemble d'*arêtes* (relation binaire sur S). Dans un graphe non-orienté, il n'existe pas de *boucle* (arête d'un sommet vers lui-même).

incidence : Si (u, v) est une arête d'un graphe non-orienté, (u, v) est *incidente* aux sommets u et v.

degré : le degré d'un sommet est le nombre d'arêtes qui lui sont incidentes. Un sommet de degré 0 est dit *isolé*.

cycle : c'est une chaîne $< v_0, ..., v_k >$ dont tous les sommets sont distincts et dont le premier sommet coïncide avec le dernier ($v_0 = v_k$).

cycle (élémentaire) : c'est un cycle ne contenant pas d'autre cycle. Un graphe sans cycle est dit *acyclique*.

Nombre cyclomatique : On appelle *nombre cyclomatique*, le nombre de cycles indépendants d'un graphe.

connexité : un graphe est *connexe* si chaque paire de sommets est reliée par une chaîne.

composantes connexes : Les *composantes connexes* d'un graphe sont constituées des classes d'équivalence de sommets induits par la relation "est accessible à partir de".

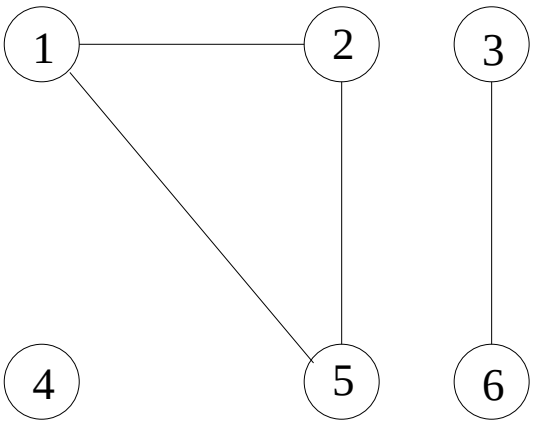

Fig. 5. *Graphe non-orienté*

Exemple 1

1. *Quelles sont les arêtes incidentes aux sommets 2 ? (rep. (1,2) et (2,5)).*
2. *Dans la figure 5, quel est le degré du sommet 2 (rep. 2), et du 4 (rep. 0, sommet isolé) ?*
3. *Trouver un cycle partant de 1. (rep. $< 1, 2, 5, 1 >$).*
4. *Quelles sont les composantes connexes du graphes représenté figure 5 ? (rep. {1,2,5}, {3,6} et {4}).*

Exercice 8

1. *Représenter la ligne C du RER sous la forme d'un graphe (on ne considérera que les têtes de lignes et les intersections).*
2. *Quelles sont les arêtes incidentes au sommet C2 ?*
3. *Quel est le degré maximal de ce graphe ?*
4. *Donner pour ce graphe, des exemples de cycles, élémentaires ou non.*
5. *Ce graphe est-il complet ?*

6. *Ce graphe est-il un arbre ?*

Exercice 9

- *Reprendre l'ensemble de ces questions pour la ligne A du RER.*
- *Comment pourrait-on définir le concept de "ligne de RER" dans le jargon de la théorie des graphes ?*
- *Quelle seraient les composantes connexes du graphe représentant le RER (sans tenir compte des intersections entre lignes) ?*

4.2.2 Graphes orientés

graphe orienté : $G = (S, A)$, où S est un ensemble de *sommets* et A un ensemble d'*arcs*. A est une relation binaire sur S. Pour un arc (u, v), on dit que l'arc *part* de u et *arrive* en v.

boucles : Un arc qui part d'un sommet et arrive sur le même sommet est appelé une *boucle*.

notion de degré :

degré sortant : nombre d'arcs qui partent d'un sommet.

degré entrant : nombre d'arcs qui arrivent à un sommet.

degré : $(degre\_entrant) + (degre\_sortant)$.

notion de chemin :

chemin : un *chemin* de *longueur* k, d'un sommet u vers un sommet u' est une séquence de sommets $< v_0, ..., v_k >$ tels que, $u = v_0$ et $u' = v_k$. Le chemin *contient* les sommets v_0, ..., v_k et les arcs (v_0, v_1), ..., (v_{k-1}, v_k).

accessibilité : u' est accessible depuis u *via* p s'il existe un chemin p de u à u'. existe un chemin

chemin élémentaire : un chemin est *élémentaire* si tous les sommets du chemin sont distincts.

sous-chemin : un *sous-chemin* d'un chemin p est une sous-séquence contiguë de ses sommets.

circuit : un chemin est un circuit si $v_0 = v_k$. Ce circuit est *élémentaire* si tous les sommets qui le composent sont distincts. Un circuit d'un sommet vers lui même est une boucle.

graphe élémentaire : un graphe orienté est *élémentaire* s'il est sans boucle.

connexité forte : un graphe est *fortement connexe* si chaque sommet est accessible à partir de n'importe quel autre.

composantes fortement connexes : Les *composantes fortement connexes* d'un graphe forment les classes d'équivalence de la relation entre sommets "sont accessibles mutuellement".

Exemple 2

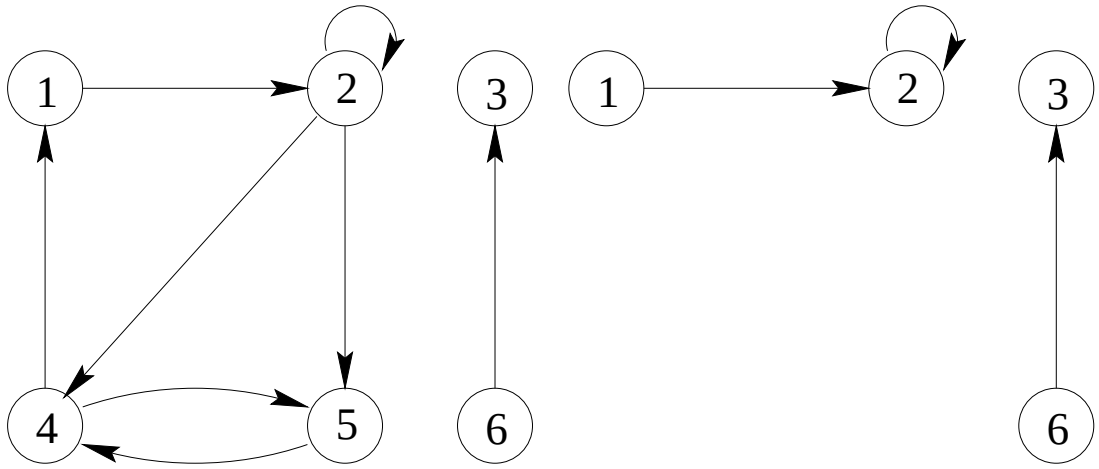

Fig. 6. *2 Graphes orientés*

1. *Quels sont les arcs qui partent du sommet 2 ? (rep. (2,2),(2,4) et (2,5)).*
2. *Quels sont les arcs qui arrivent en 2 ? (rep. (1,2) et (2,2)).*
3. *Dans la figure 6, quel est le degré du sommet 2 ? Expliciter en fonction des degrés entrant et sortant. (rep. degré entrant : 2 ; degré sortant : 3 ; degré : 5).*
4. *Trouver un chemin élémentaire pour aller de 1 à 4. (ex.* $< 1, 2, 5, 4 >$ – $< 2, 5, 4, 5 >$ *n'est pas élémentaire).*
5. *Trouver un circuit élémentaire partant de 1 (rep.* $< 1, 2, 4, 1 >$ – $< 1, 2, 4, 5, 4, 1 >$ *n'est pas élémentaire).*
6. *Quelles sont les composantes fortement connexes du graphe 6 ? (rep.* $\{1,2,4,5\}$, $\{3\}$ *et* $\{6\}$ *– tous les sommets de* $\{1,2,4,5\}$ *sont mutuellement accessibles –* $\{3,6\}$*, ne forme pas une composante fortement connexe).*
7. *Quel est le sous graphe engendré par l'ensemble des sommet* $\{1,2,3,6\}$ *? (rep. c'est le sous-graphe orienté représenté sur la même figure).*

Exercice 10

– *Exprimer la généalogie de lignée d'Edward III sous la forme d'un graphe orienté, en considérant que les arcs signifient "le descendant direct de".*
– *Quel est le degré entrant maximal d'un sommet ?*
– *Quel est le degré sortant maximal d'un sommet ?*
– *En déduire le degré maximal d'un sommet quelconque.*
– *Quel est le cycle de longueur maximale dans ce graphe ?*
– *Quels sont les cycles élémentaires pouvant être trouvés dans ce graphe ?*
– *Quel est le nombre cyclomatique de ce graphe ?*
– *Ce graphe est-il fortement connexe ?*
– *Existe-t-il des circuits ?*

4.2.3 Définitions communes

ordre : L'*ordre* d'un graphe est son cardinal.

adjacence : S'il existe un arc ou une arête (u, v), on dit que u est *adjacent* à v. Pour un graphe non-orienté, la relation d'adjacence est symétrique. Pas nécessairement pour un graphe orienté.

isomorphisme : 2 graphes sont *isomorphes* s'il existe une bijection $f : S \to S'$ t.q. $(u, v) \in A$ ssi $(f(u), f(v)) \in A'$.

sous-graphe : $G' = (S', A')$ est un *sous-graphe* de $G = (S, A)$, si $S' \subseteq S$ et $A' \subseteq A$.

graphe engendré : Soit $S' \subseteq S$. Le sous-graphe *engendré* par S' est le graphe $G' = (S', A')$, où $A' = \{(u, v) \in A : i, v \in S'\}$.

graphe partiel : Soient $G = (S, A)$ et $A' \subset A$. Le *graphe partiel* engendré par A' est le graphe ayant le même ensemble de sommets que G et dont les arcs sont les arcs de A' (si G est le graphe de toutes les routes de France, le graphe des autoroutes est un graphe partiel de G). $G' = (S', A')$, où $A' = \{(u, v) \in A : i, v \in S'\}$.

version orientée : La *version orientée* d'un graphe non-orienté $G = (S, A)$ est le graphe orienté $G' = (S, A')$ où $(u, v) \in A'$ ssi $(u, v) \in A$ (chaque arête (u, v) est remplacée par 2 arcs (u, v) et (v, u)).

version non-orientée : La *version non-orientée* d'un graphe orienté G, est le graphe orienté $G' = (S, S, A')$ où $(u, v) \in A$ ssi $u \neq v$ et $(u, v) \in A$ (arcs sans leur orientation, suppression des boucles).

Exemple 3

1. *Dans la figure 5, 1 est-il adjacent à 2 ? (rep. oui)*
2. *Dans la figure 6, 1 est-il adjacent à 2 ? (rep. non)*
3. *la version orientée de la version non-orientée d'un graphe orienté est elle égale à ce graphe (rep. uniquement si le graphe est élémentaire).*
4. *Dans la figure 7-a), les graphes sont-ils isomorphes ? (rep. oui). Même question pour 7-b). (rep. non – le graphe du haut comporte un sommet de degré 4 – sommet 1).*

Exercice 11

– *Donner les versions orientées des graphes représentant les lignes A et C du RER.*
– *Donner la version non-orientée du graphe généalogique d'Edward III.*

4.2.4 Graphes "spéciaux"

graphe complet : Un *graphe complet* est un graphe non-orienté dans lequel les sommets sont tous adjacents 2 à 2.

graphe biparti : Un *graphe biparti* est un graphe non-orienté $G = (S, A)$, dans lequel S peut être partitionné en 2 ensembles S_1 et S_2 t.q. $(u, v) \in A \Rightarrow u \in S_1$ et $v \in S_2$ (ou $u \in S_2$ et $v \in S_1$).

multigraphe : Un *multigraphe* est un graphe non-orienté dans lequel il peut exister plus d'un arête reliant un sommet à un autre.

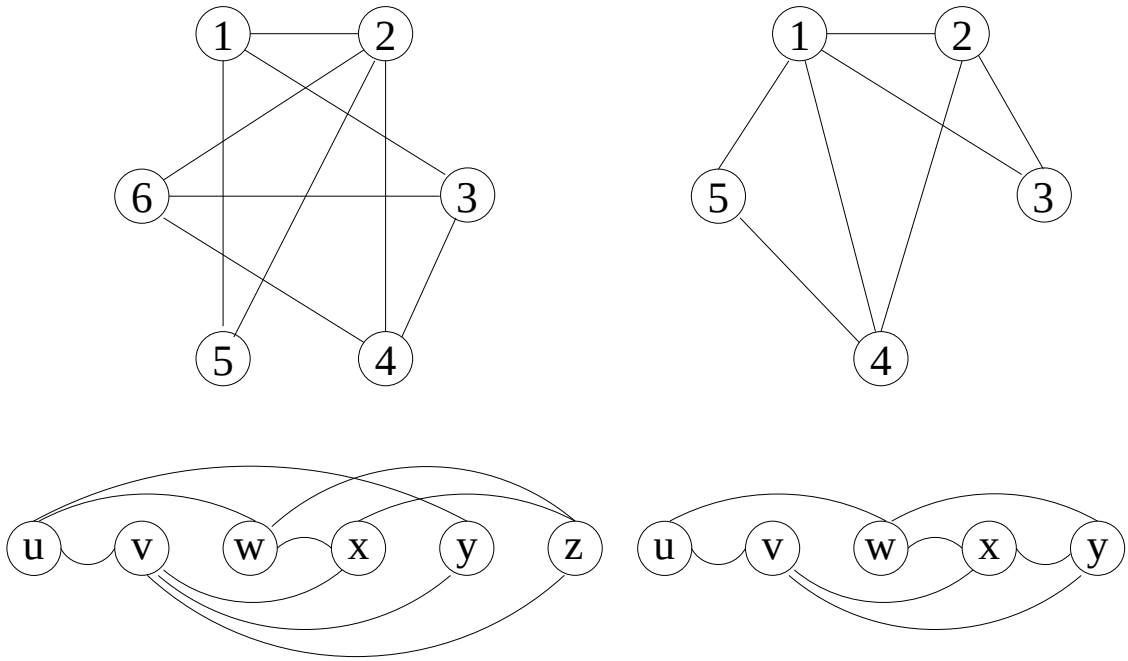

Fig. 7. *a) Graphes isomorphes. b) Graphes non-isomorphes*

graphe symétrique : on dit qu'un graphe est *graphe symétrique*, s'il existe autant d'arêtes (u, v) que d'arêtes (v, u).

foret : Un graphe non-orienté acyclique est appelé une *foret*.

arbre : Un graphe non-orienté connexe acyclique est appelé un *arbre*.

graphe planaire : on dit qu'un graphe est *planaire* s'il admet une représentation sur un plan P.

graphe aux arêtes ou graphe aux arcs : le graphe aux arêtes ou graphe aux arcs d'un graphe $G = (S, A)$, est un graphe noté $L(G)$ dont les sommets représentent les arêtes de G et les arêtes le fait que les arêtes du graphe G ont une extrémité commune dans G.

Exemple 4 Graphe planaire : *étant donné 3 villas que l'on veut relier par des conduites à une usine de production d'eau et une usine de production de gaz. Il est impossible de relier les 3 villas aux 2 usines sans que les conduites se croisent ! Le graphe n'est pas planaire.* ***APPLICATION*** *: électronique.*

Exercice 12 *Structures de réseaux*

1. *Représenter sous la forme d'un graphe les architectures de rséau suivantes :*
   - *Point à point*
   - *Multipoints*

- *Anneau logique*
- *étoile*
- *Bus*
- *Arborescent*
- *Réseau maillé*

4.3 Arbres

nœud : Lorsqu'on parle des sommets d'un arbre on parle plutôt de *nœud*.

Propriété des arbres 1 *Soit $G = (S, A)$ un graphe non-orienté. Les affirmations suivantes sont équivalentes.*

1. *G est un arbre*
2. *2 sommets quelconques de G sont reliés par un chemin élémentaire.*
3. *G est connexe, mais si un sommet quelconque est ôté de A, le graphe résultant n'est plus connexe.*
4. *G est connexe, et $|A| = |S| - 1$.*
5. *G est acyclique, et $|A| = |S| - 1$.*
6. *G est acyclique, mais si une arête quelconque est ajoutée à A, le graphe résultant contient un cycle.*

4.3.1 Arbres enracinés

arbre enraciné : Un *arbre enraciné* est un arbre dans lequel l'un des sommets (appelé la *racine*) se distingue des autres. Ce qui impose un sens de parcours de l'arbre (*arborescente*).

descendance :

- ***ancêtre :*** Un nœud quelconque sur le chemin unique allant de la racine r à x est appelé un *ancêtre* de x (*ancêtre propre* si $x \neq y$).
- ***descendant :*** si x est un ancêtre de y, alors y est un *descendant* de x (*descendant propre* si $x \neq y$).
- ***sous-arbre racine :*** le *sous-arbre racine* de x est l'arbre composé des descendants de x enraciné en x.

parenté : si le dernier arc sur le chemin de la racine d'un arbre vers un nœud x est (y, x), alors y est le *père* de x et x est le *fils* de y. Deux nœuds ayant le même père sont *frères*

feuille : un nœud sans fils est un *nœud externe* ou encore, une *feuille*. Un nœud qui n'est pas une feuille est un *nœud interne*.

degré : Le nombre de fils d'un nœud x dans un arbre enraciné est appelé *degré*.

profondeur : La profondeur de x dans un arbre enraciné est la longueur du chemin entre la racine r et le nœud x.

hauteur : La *hauteur* est la profondeur du nœud le plus profond de l'arbre.

arbre ordonné : un *arbre ordonné* est un arbre enraciné dans lequel les fils de chaque nœuds sont ordonnés.

Exemple 5 *Les dictionnaires informatiques.*

Exercice 13

1. *Représenter le graphe de la ligne RER A sous la forme d'un arbre enraciné à la station CHATELET.*
2. *Lister l'ensemble des sous-arbres construits à partir de cette racine.*
3. *Quelles sont les feuilles de l'arbre ?*
4. *Quelle est la profondeur de cet arbre ?*

4.3.2 Arbres binaires

Arbre binaire 1 *Un arbre binaire T est un structure définie sur un ensemble fini de nœuds qui :*
- *ne contient aucun nœud, ou*
- *est formé de 3 ensembles de nœuds disjoints :*
  - *un nœud* racine
  - *un arbre binaire appelé* sous-arbre gauche
  - *un arbre binaire appelé* sous-arbre droit

arbre vide : l'arbre binaire qui ne contient aucun nœud est appelé *arbre vide* ou *nil*.

fils : Si le sous-arbre gauche (resp. droit) n'est pas vide, sa racine est appelée *fils gauche* (resp. *droit*) de la racine de l'arbre entier.

fils manquant : Si un sous-arbre est l'arbre vide, on dit que le fils est *absent* ou *manquant*.

arbre binaire complet : un *arbre binaire complet* est un arbre dont les nœuds sont soit des feuilles, soit des nœuds de degré 2.

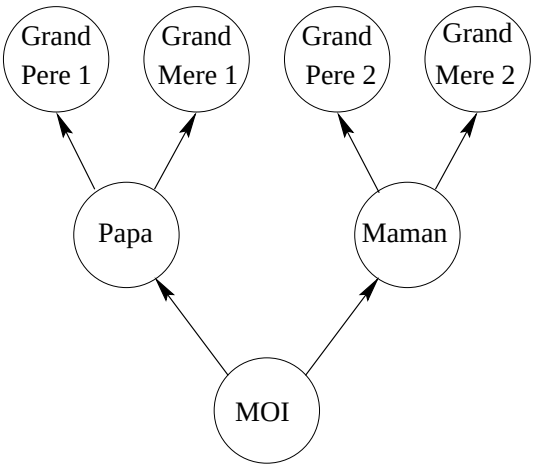

Fig. 8. *Arbre généalogique*

Exemple 6 *Représentation des opérations arithmétiques.*

Exercice 14

1. *Montrer que le graphe de la figure 8, est un graphe binaire.*
2. *Cet arbre est-il complet ?*
3. *Montrer que la hauteur d'un arbre binaire complet comportant n feuilles est $log_2 n$, et que le nombre de nœuds internes est $2^h - 1$.*

Arbre binaire de recherche

Un *arbre binaire de recherche* est un arbre binaire dont les nœuds sont associés à une clé numérique et satisfont à la relation suivante :

> Si x est un nœud de l'arbre et y un nœud appartenant au sous-arbre gauche (resp.) de x, alors `Cle[y]`$\leq$`Cle[x]` (resp. `Cle[x]`$\leq$`Cle[y]`).

– écrire l'arbre binaire de recherche représentant le jeu "trouver un nombre entre 1 et 10".

4.4 Implémentation des graphes

Jusqu'à présent, nous avons présenté les graphes comme une structure purement mathématique. Nous nous intéressons maintenant à leur mode de représentation informatique.

4.4.1 Liste d'adjacence (ou d'incidence sommet-sommet)

La structure de *liste d'adjacence* est souvent privilégiée car elle permet de représenter un graphe peu dense sans utiliser beaucoup de ressources (ceux pour qui $|A| \ll |S|^2$).
La représentation en liste d'adjacence est un tableau indexé par le numéro d'un sommet du graphe, et dont la structure est celle d'une liste chaînée des sommets adjacents à ce sommet.
Pour un graphe orienté, la somme des longueurs de toutes les listes d'adjacence vaut $|A|$ et $2 \cdot |A|$, pour en graphe non-orienté. Cette représentation demande donc peut de place mémoire.
La liste d'adjacence peut être facilement adaptée à la représentation des *graphes pondérés* (i.e. aux graphes donc chaque arêtes possède un poids). Pour cela, il suffit de stocker ce poids dans la liste chaînée représentation l'arete.
Un inconvénient de ce genre de représentation est que pour déterminer si un arc existe, il est nécessaire de parcourir toute la liste d'adjacence.

Exemple 7
- *Représenter sous la forme d'une liste d'adjacence le graphe de la figure 6.*
- *Représenter sous la forme d'une liste d'adjacence le graphe de la figure 5.*

4.4.2 Matrice d'adjacence (ou d'incidence sommet-sommet)

Si le graphe est plus dense ($|A| \approx |A|^2$) on préfère représenter le graphe sous la forme de sa *matrice d'adjacence*. Cette représentation est aussi utilisée quand on désire savoir rapidement si un arc entre 2 sommets existe ou non.
Dans cette représentation, on suppose que les sommets sont numérotés et on construit un matrice $|S| \times |S|$, $M = (a_{ij})$ t.q. :

$$a_{ij} = \begin{cases} 1 & \text{si } (i,j) \in A \\ 0 & \text{sinon} \end{cases}$$

Exemple 8
- *Représenter sous la forme d'une matrice d'adjacence le graphe de la figure 6.*
- *Représenter sous la forme d'une matrice d'adjacence le graphe de la figure 5.*

4.4.3 Représentation des multigraphes

On peut remarquer que les représentation en matrice d'adjacence, ne permettent pas de représenter les multigraphes.

La représentation en matrice d'incidence sommet-arcs ou sommet arêtes (selon que le graphe est orienté ou non), consiste à utiliser une matrice dont les lignes correspondent aux sommets et les colonnes aux arcs.
Dans le cas d'un graphe orienté, si un arc part du sommet, la composante correspondante de la matrice est $+1$ et de -1 si il arrive à ce sommet. Dans tous les autres cas, la composante est nulle (pour le cas du graphe non-orienté, il suffit d'indiquer $+1$ si une arête arrive ou part d'un sommet).

4.4.4 Représentation des arbres

Comme on l'a précisé plus haut, un arbre est un graphe et peut, à ce titre, être représenté tel quel. Cependant, vue sa structure particulière, il existe des représentations spécifiques.

Arbre binaire/Généralisation aux arbres n-aires

Pour chaque nœud, on utilise 3 pointeurs :
– Un sur le père (Nil si c'est la racine)
– Un sur le fils gauche (Nil s'il n'y en a pas).
– Un sur le fils droit (Nil s'il n'y en a pas).
Ce type de représentation peut être aisément généralisée à des arbres n-aires. Dans ce cas, chaque nœud dispose d'un pointeur sur chacun des fils potentiels.
On voit cependant que ce type de représentation n'est pas économique en terme d'occupation mémoire. Par ailleurs, elle ne permet pas de représenter des structures arborescentes dans lesquelles le nombre de fils est important et inconnu à l'avance.

Représentation *fils-gauche/frère droit*

Pour résoudre ce problème il est possible d'utiliser la représentation dite "*fils-gauche/frère droit*". Chaque nœud possède alors :
– Un pointeur sur le père
– Un pointeur sur le fils gauche
– Un pointeur sur le frère droit
Ce mécanisme permet donc de créer une liste chaînée des ferres présents à un niveau donné de l'arbre.

Représentation par "*TAS*"

Un tas est un tableau `A` qui peut être vu comme un arbre binaire presque complet. Chaque nœud de l'arbre correspond à un élément du tableau qui contient la valeur du nœud. L'arbre est complètement rempli à tous les niveaux, sauf, parfois, sur le plus bas, rempli à partir de la gauche, jusqu'à un certain point.
La racine de l'arbre est `A[1]`. Sachant l'indice `i` d'un nœud, les indices de son `Père(i)`, de son fils `Gauche(i)` et de son fils `Droit(i)`, sont donnés par :

Père(i) : retourner($\frac{i}{2}$)

Gauche(i) : retourner($2 \cdot i$)

Droit(i) : retourner($2 \cdot i + 1$)

Sachant que ces opérations sont habituellement des opérations élémentaires réalisées aux niveaux du microprocesseur (décalage à gauche ou à droite avec ou sans remplissage).
Enfin, la propriété la plus importante des tas est que pour tout nœud `i` autre que la racine on a :

$$A[Pere(i)] \geq A[i]$$

Nous verrons plus tard comment utiliser la structure de TAS pour effectuer un tri.

5 Algorithmes sur les arbres

5.1 Algorithmes sur les arbres binaires de recherche

Les opération possibles sur un arbre binaire sont celles qui consistent à rechercher :
– si un élément est présent dans un arbre (RECHERCHER).
– le maximum (resp. minimum) (MAXIMUM resp. MINIMUM).
– le prédécesseur (resp. le successeur), conditionnellement à un parcours donné (infixe, préfixe, postfixe – PRÉDÉCESSEUR resp. SUCCESSEUR).

Rechercher

Le principe est de savoir si un nœud correspondant à une clé k donnée est présent dans l'arbre. La fonction RECHERCHER(x, k), recherche dans l'arbre dont la racine est x, la clé k et renvoi un pointeur sur le nœud contenant cette clé ou NIL si la clé n'est pas présente dans l'arbre.

Rechercher(x,k)

1. *si* $x =$NIL *ou* $k =$*clé[x]*
   alors retourner x
2. *si* $k <$*clé[x]*
   alors retourner RECHERCHER(GAUCHE[X],k)
   sinon retourner RECHERCHER(DROITE[X],K)

Version itérative :

Rechercher(x,k)

1. *tant que* $x \neq$NIL *et* $k \neq$*clé[x]*
   faire si $k <$*clé[x]*
   alors $x \leftarrow$*gauche[x]*
   sinon $x \leftarrow$*droit[x]*
2. *retourner* x

Exercice 15

1. *écrire un programme itératif permettant de trouver l'élément* minimal *d'un arbre binaire de rechercher.*
2. *écrire un programme récursif permettant de trouver l'élément* maximal *d'un arbre binaire de rechercher.*

3. *En utilisant les procédures écrites ci-dessus, écrire le programme permettant de trouver le* successeur *d'un nœud de l'arbre (en considérant un parcours infixe).*
4. *En utilisant les procédures écrites ci-dessus, écrire le programme permettant de trouver le* prédécesseur *d'un nœud de l'arbre (en considérant un parcours préfixe).*

Insertion

Pour insérer une nouvelle valeur v dans un arbre binaire de recherche, il faut rechercher de manière récursive dans l'arbre le sous-arbre dans lequel on doit insérer la valeur, jusqu'à atteindre une feuille. Dans ce cas, on insère l'élément à gauche ou à droite respectivement si la clé est inférieure ou supérieure à la clé de la feuille.

Insertion(T,z)

```
1. y ← Nil
2. x ← racine[T]
3. tant que x ≠Nil
     faire y ← x
       si clé[z]<clé[x]
         alors x ← gauche[x]
         sinon x ← droit[x]
4. p[z]← y
5. si y =Nil
     alors racine[T]← z
     sinon si clé[z]<clé[y]
       alors gauche[y]← z
       sinon droit[y]← z
```

Exercice 16 *Expliquer comment construire un arbre binaire de recherche à partir de la séquence* {*12,5,18,2,9,19,15,13,17*}

Suppression

Pour supprimer une valeur v (correspondant à un nœud z d'un arbre binaire de recherche, il faut considérer 3 cas :

1. Si z n'a pas de fils, il suffit de mettre le pointeur du père à Nil.
2. Si z n'a qu'un seul fils, le pointeur du père pointe sur le fils de z.

3. Si z a 2 fils, on détache le successeur de z, y, qui n'a pas de fils gauche et on remplace la clé de z par celles de y.

Supprimer(T,z)

1. *si gauche[z]=*NIL *ou droit[z]=*NIL
   alors $y \leftarrow z$
   sinon $y \leftarrow$ *Successeur(z)*
2. *si gauche[z]*$\neq$NIL
   alors $x \leftarrow$*gauche[y]*
   sinon $x \leftarrow$*droit[y]*
3. *si* $x \neq$NIL
   alors p[x]$\leftarrow$*p[y]*
4. *si p[y]=*NIL
   alors racine[T]$\leftarrow x$
   sinon si y=gauche[p[y]]
   alors gauche[p[y]]$\leftarrow x$
   sinon droit[p[y]]$\leftarrow x$
5. *si* $y \neq z$
   alors clé[z]$\leftarrow$*clé[y]*
6. *retourner y*

Exercice 17 *1. Construire l'arbre binaire de recherche issu de la séquence {15,16,5,12,3,20,10,18,13,23,6,7}*

2. Que se passe-t-il si l'on supprime l'élément 13 ?

3. Que se passe-t-il si l'on supprime l'élément 16 ?

4. Que se passe-t-il si l'on supprime l'élément 5 ?

5.1.1 Algorithmes sur les arbres rouge et noir : arbres de recherche équilibrés

Un *arbre rouge et noir* est un arbre binaire de recherche comportant un bit de stockage supplémentaire permettant d'indiquer si l'arbre est rouge ou noir. Si on impose certaines contraintes permettant d'affecter une couleur en fonction des couleurs voisines, on peut assurer que l'arbre est approximativement *équilibré* (c'est à dire que toutes le feuilles de l'arbre sont approximativement à la même profondeur).

Propriété des arbres rouge et noir 1 *1. Chaque nœud est soit rouge, soit noir*

2. Chaque feuille NIL *est noire*

3. *Si un nœud est rouge, ses 2 fils sont noirs*
4. *Chaque chemin simple liant un nœud à une feuille descendante contient le même nombre de nœuds noirs.*

hauteur noire c'est le nombre de nœuds noirs sur un chemin allant d'un nœud à une feuille (sans inclure le nœud).

Les opérations sur les arbres rouge et noir sont extrêmement rapide car on peut montrer qu'un tel arbre possédant n nœuds a au plus une hauteur de $2 \cdot log(n+1)$.

Rotations

Les opérations d'insertion et de suppression sur un arbre rouge et noir peuvent conduire à violer les propriétés caractéristiques de cette arbre. Pour retrouver un arbre rouge et noir il faut effectuer des modifications de couleurs afin de retrouver la structure convenable.
Pour cela, on utilise l'opération locale de *rotation de sous-arbre*.

Rotation-Gauche(T,x)

1. $y \leftarrow droit[x]$
2. $droit[x] \leftarrow droit[y]$
3. **si** $gauche[y] \neq Nil$
   alors $p[gauche[y]] \leftarrow x$
4. $p[y] \leftarrow p[x]$
5. **si** $p[x] = Nil$
   alors $racine[T] \leftarrow y$
   sinon si $x = gauche[p[x]]$
   alors $gauche[p[x]] \leftarrow y$
   sinon $droit[p[x]] \leftarrow y$
6. $gauche[y] \leftarrow x$
7. $p[x] \leftarrow y$

Insertion

On se pose maintenant le problème de l'insertion d'un élément dans ce type d'arbre. Quels sont alors les problèmes à résoudre ?
La principale difficulté dépend de la nécessité de respecter les contraintes de couleurs à chaque niveau de l'arbre. Pour cela, il faut vérifier que l'insertion ne viole pas les contraintes de l'arbre rouge-noir, et, le cas échéant, de modifier la structure de l'arbre par des opérations de rotations pour rétablir la structure.

On part du principe que le nœud inséré est rouge et on considère l'arbre rouge-noir comme un arbre ordinaire de recherche. On détermine alors les modifications à apporter le cas échéant pour restructurer l'arbre en arbre rouge-noir.

Inserer-rouge-noir(T,x)

```
1. Insérer-arbre-binaire(T,x)
2. couleur[x] ← ROUGE
3. tant que x ≠ racine[T] et couleur[p[x]] = ROUGE
     faire si  p[x] = gauche[p[p[x]]]
       alors y ← droit[p[p[x]]]
         si couleur[y] = ROUGE
           alors couleur[p[x]] ← NOIR
                 couleur[y] ← NOIR
                 couleur[p[p[x]] ← ROUGE
                 x ← p[p[x]]
           sinon si x = droit[p[x]]
           alors x ← p[x]
                 Rotation-Gauche(T,x)
           couleur[p[x]] ← NOIR
           couleur[p[p[x]] ← ROUGE
           Rotation-Droite(T,p[p[x]])
       sinon y ← gauche[p[p[x]]]
         si couleur[y] = ROUGE
           alors couleur[p[x]] ← NOIR
                 couleur[y] ← NOIR
                 couleur[p[p[x]] ← ROUGE
                 x ← p[p[x]]
           sinon si x = gauche[p[x]]
           alors x ← p[x]
                 Rotation-Droite(T,x)
           couleur[p[x]] ← NOIR
           couleur[p[p[x]] ← ROUGE
           Rotation-Gauche(T,p[p[x]])
4. couleur[racine[T]] ← NOIR
```

Le début du programme sert à positionner le nœud là où il devrait être dans le cas d'un arbre binaire de recherche. Puis à lui affecter la couleur rouge.
On recherche ensuite dans quelle mesure il y a invalidation des propriétés de l'arbre rouge-noir. En fait, la seule propriété qui peut être violée est celle qui affirme qu'un nœud rouge ne peut pas avoir de fils rouge.

La boucle `tant que` sert à faire remonter le long de l'arbre le non respect de cette propriété tout en maintenant la propriété indiquant que le nombre de nœuds noirs est le même sur tout chemin partant d'un nœud donné.
Il faut considérer 6 cas possibles, mais 3 sont symétriques :

Cas 1 à suivre...

5.2 Parcours d'un arbre

Parcours d'un arbre binaire

Le parcours *infixe* d'un arbre binaire, consiste à parcourir l'arbre en commençant par le sous arbre gauche, en continuant par la racine et en finissant par le sous-arbre droit.
Ce parcours peut être réalisé grâce à une procédure récursive.
Dans le parcours *préfixe*, la racine est parcourue d'abord ; dans le parcours *postfixe*, la racine est parcourue en dernier.

Exercice 18

1. *Représenter l'équation* $x^2 + 2 \cdot x + 1$ *sous la forme d'un arbre binaire.*
2. *Cette représentation est-elle unique ?*
3. *Que donne un parcours infixe de cet arbre ?*
4. *Que donne un parcours préfixe de cet arbre ?*
5. *Que donne un parcours postfixe de cet arbre ? (cette représentation est aussi appelée "polonaise inverse").*

Parcours en largeur d'abord

Le principe de cet algorithme est très simple ; il consiste à partir de la racine et d'énumérer l'ensemble de ses descendants directs puis réitérer le processus sur les descendants eux-mêmes. On effectue ainsi un parcours "de gauche à droite et de haut en bas".
On utilise 2 listes appelées OUVERT et FERME.
– La liste OUVERT est une file d'attente qui contient les sommets en attente d'être traités
– La liste FERME contient les sommet dont les successeurs, s'ils existent, ont déjà été énumérés.
L'algorithme est le suivant :

Largeur-D-abord

1. *Placer le sommet initial* s_0 *dans OUVERT*
2. *tant qu'OUVERT n'est pas vide*
   soit n le premier élément d'OUVERT
   mettre les fils de n en queue *d'OUVERT*
   {*effectuer le traitement pour n*}
   mettre n dans FERME
   – *fin tant que*

Exemple 9 *Recherche dichotomique dans un arbre pour trouver un nombre entre 1 et 10.*

Parcours en profondeur d'abord

Le principe de cet algorithme est de descendre le plus profondément dans l'arbre avant de se déplacer en largeur. On effectue ainsi un parcours " de haut en bas et de gauche à droite".
La encore, on utilise 2 listes appelées OUVERT et FERME.
L'algorithme est le suivant :

Profondeur-D-abord

1. *Placer le sommet initial s_0 dans OUVERT*
2. *tant qu'OUVERT n'est pas vide et que la profondeur n'est pas la profondeur limite*
   soit n le premier élément d'OUVERT
   mettre les fils de n en tête *d'OUVERT*
   {*effectuer le traitement pour n*}
   mettre n dans FERME
   – *fin tant que*

Problème : Il se peut que l'on s'enfonce indéfiniment dans l'arbre. **Solution :** Tester une profondeur maximum ⇒ Dans ce cas, le parcours en largeur d'abord n'est pas complet.

5.3 Problème de recherche arborescente

On s'intéresse maintenant à des problèmes dans lesquel on cherche un élément qui soit une solution optimale à un problème, suivant un critère donné a priori. Si on se place dans un ensemble fini de très grande taille, on ne connait pas d'algorithme polynomial générique permettant de trouver cet élément.
L'idée est d'utiliser la notion "cartésienne" de division de problèmes en sous-problèmes. On applique se raisonnement jusqu'à atteindre un sous-problème qu'est capable de résoudre, puis on synthétise l'ensemble des résultats de manière à donner une solution globale.
Pour simplifier les algorithmes de rechercher il est intéressant de pouvoir rapidement élager une partie de l'arborescence.

5.3.1 Algorithme séparation et évaluation *Branch and Bound*

Cet algorithme construit l'arborescence en évaluant a priori les chances de trouver la solution optimale dans une branche particulière. Pour cele, il est nécessaire d'introduire une **heuristique** (fonctionnement d'évaluation de situation), qui permet de déterminer si un solution est plus avantageuse qu'une autre.

L'évaluation empirique nous donne un parcours de l'arbre qui est *en moyenne* plus rapide qu'un algorithme de parcours exhaustif. En effet, on examine uniquement les sommets qui semblent a priori intéressant, ce qui permet souvent de trouver rapidement la solution, ou une solution satisfaisante.

Exemple des reines sur l'échiquier

On veut placer n reines sur un échiquer de dimension $n \times n$ de manière qu'aucune ne puisse attraper les autres.
Pour ce ramener à un problème de recherche de solution optimale dans un arbre, il suffit de considérer que l'on recherche dans l'ensemble des façons de placer les reines, la ou les solutions telles qu'aucune reine ne peut prendre les autres.
La division en sous-problème peut se faire en considérant qu'un sous-arbre est constitué de la position de la première reine sur le première colonne. Les sous-arbres sont obtenus en considérant la position de la $n$ieme reine sur la $n$ieme colonne (en posant le problème sous cette forme, on a déjà supprimé tous les cas pour lesquelles les reines pouvaient etre sur la même colonne, ce qui est un a priori sur le jeu).
En construisant l'arborescence en largeur ou profondeur d'abord, on peut trouver la solution s'il en existe.

chercher-solution-reines

```
1. visiter un sommet
      si p = n alors succès
      sinon
      p++
         pour i de 1 à n
            si case(p, i) libre
               placer une reine sur cette case
               rayer la colonne p et la ligne i
               rayer les diagonales passant par (p, i)
               visiter le sommet correspondant à cette configuration
            fin si
         fin pour
```

6 Algorithmes sur les graphes

6.1 Parcours des graphes

Il est tout a fait possible, moyennant quelques modifications, d'utiliser les parcours en largeur et profondeur d'abord. En fait, comme il y a des cycles dans le graphe, il ne faut pas reparcourir des sommets qui ont déjà été visités.

6.1.1 Parcours en largeur d'abord

On peut appliquer l'algorithme en considérant que l'on parcourt des arborescences associées aux sommets du graphe (l'arborescence est celle constituée des descendants qui n'ont pas encore été parcourus). Les arbres sous-jacents forment une foret recouvrante du graphe (contient tous les sommets du graphe).
L'algorithme est le suivant :

Largeur-D-abord-graphes

1. *Visiter tous les sommets*

2. VISITER-SOMMET(S)
3. *Si s n'est pas dans OUVERT, ni dans FERME*
   placer le sommet initial s dans OUVERT
   - *tant qu'OUVERT n'est pas vide*
     soit n le premier élément d'OUVERT
     mettre les successeurs de n qui ne sont pas déjà dans OUVERT en queue *d'OUVERT*
     {*effectuer le traitement pour n*}
     mettre n dans FERME
     - *fin tant que*

Remarque : le parcours en largeur d'abord est une stratégie COMPLÈTE.

6.1.2 Parcours en profondeur d'abord

L'algorithme est le suivant :

Profondeur-D-abord-graphes

1. *Visiter tous les sommets*

2. VISITER-SOMMET(s)
3. *Si s n'est pas dans OUVERT, ni dans FERME*
   placer le sommet initial s dans OUVERT
   - *tant qu'OUVERT n'est pas vide*
     soit n le premier élément d'OUVERT
     mettre les successeurs de n qui ne sont pas déjà dans OUVERT en tête *d'OUVERT*
     {*effectuer le traitement pour n*}
     mettre n dans FERME
     - *fin tant que*

Remarque : le parcours en largeur d'abord est une stratégie COMPLÈTE.

Exemple 10 *Recherche d'un élément situé entre 1 et 10 à partir des opérations* $+1$ *et* -1 *en partant de 5.*

Utilisation des parcours en largeur et profondeur d'abord

Accessibilité Les algorithmes de parcours de graphe peuvent être utilisés pour déterminer quels sont les sommets accessibles depuis un sommet donné. Pour cela, il suffit d'utiliser l'un des algorithmes de parcours à partir du sommet donné et de marquer les sommets visités.

Exemple 11 *Un passeur se trouve sur le bord d'un fleuve. Il doit passer de l'autre coté, un loup, un bouc et un chou. Cependant, il ne peut transporter qu'un seul client à la fois. Par ailleurs, il ne laisser ensemble ni le loup et le bouc, ni le bouc et le chou ? Existe t'il un chemin solution ?*

Composantes connexes On peut aussi utiliser les algorithmes de parcours pour déterminer les composantes connexes d'un graphe.

Rappel : les composantes connexes d'un graphe sont les ensembles de sommets accessibles les uns depuis les autres.
Pour cela, on utilise l'algorithme de parcours en profondeur d'abord et on colorie les sommets rencontrés avec une couleur correspondant à la composante connexe courante.

Composantes-Connexes

1. *Partir d'un sommet s*
2. *S'il est colorié*
   ne rien faire
   sinon
   choisir une couleur
   appliquer profondeur d'abord en marquant avec la couleur

Exemple 12 *Utiliser cet algorithme pour déterminer le nombre de composantes connexes du graphe de la figure 6.*

Graphe orienté sans circuit On veut détecter si il y a un ou des circuits dans un graphe.
Pour cela, on utilise un algorithme de parcours. Si durant ce parcours on revient sur un sommet présent dans OUVERT ou FERME, c'est qu'il y a un circuit.

Exemple 13 *Utiliser cet algorithme pour prouver que le graphe 6 comporte des cycles.*

Tri topologique Cet algorithme ne s'applique que sur un graphe sans circuit. Le principe est de trouver une numérotation des sommets telle que les descendants d'un sommet ont toujours une numérotation supérieure.
Pour cela, on utilise le parcours en profondeur d'abord et on numérote à l'envers à partir du sommet le plus profond.

Tri-Topologique(G)

1. *appeler Profondeur-D-Abord(G) pour calculer les temps de fin de traitement pour chaque sommet*
2. *chaque fois que le traitement d'un sommet s'achève, l'insérer au début d'une liste chaînée*
3. *retourner la liste chaînée des sommets*

Exemple 14 *On cherche à déterminer dans quel ordre on enfiler ses vêtements pour s'habiller de la tête aux pieds. Sachant que :*
- *Pour mettre son caleçon il faut préalablement avoir mis ses chaussures, son pantalon et sa ceinture.*
- *Pour mettre ses chaussures il faut avoir mis ses chaussettes et son pantalon.*
- *Pour mettre sa ceinture il faut avoir enfile sa chemise.*
- *Pour mettre sa veste il faut avoir enfilé sa cravate et sa ceinture.*
- *Pour mettre sa cravate il faut avoir mis sa chemise.*
- *On peut mettre sa montre n'importe quand ?*

6.2 Fermeture transitive

Étant donné un graphe orienté $G = (S, A)$, on désire savoir s'il existe un chemin dans G de i vers j, $\forall(i, j)$. La *fermeture transitive* de G est définie par le graphe $G^* = (S, A^*)$ tel que :

$$A^* = \{(i, j) : \exists \text{ un chemin du sommet } i \text{ au sommet } j \text{ dans } G\}$$

Produit de matrices

Soient 2 matrices A et B de dimension (n,n). Leur produit P est de même dimension et défini par :

$$p_{ij} = \sum_{k=1}^{n} a_{i,k} bk, j$$

La complexité de cet algorithme est en $O(n^3)$.

Chemin d'une longueur donnée

Si M est la matrice d'adjacence d'un graphe G à n sommets, on peut montrer que la matrice M^p vérifie :

$$m_{ij}^p = 1 \Leftrightarrow \exists \text{ un chemin } p \text{ allant du sommet } i \text{ au sommet } j$$

La complexité d'un algorithme naïf est en $O(p \cdot n^3)$. On peut réduire à $O(\ln p \cdot n^3)$ grâce à l'algorithme suivant :

M-puissance-p

1. *si* $p = 1$
   alors $M^p \leftarrow M$
   sinon
   $J \leftarrow M^{Ent[\frac{p}{2}]}$
   $K \leftarrow J \times J$
   si p est pair
   alors $M^p \leftarrow K$
   sinon $M^p \leftarrow K \times M$

Algorithme de calcul de la fermeture transitive basé sur une approche mathématique S'il existe une chemin entre 2 sommets de G, il existe un chemin élémentaire entre ces 2 sommets de longueur $< n$. La matrice de fermeture transitive est donc la somme logique des matrice de puissance jusqu'à la taille n.

$$T = M + m^2 + M^3 + ... + M^{n-1}$$

La complexité d'un tel algorithme est en $O(n^4)$.

Algorithme de Roy-Warshall

On construit la fermeture transitive A en effectuant sur l'ensemble des sommets s le traitement $\Phi_s(A)$ qui consiste à ajouter les arcs (y, z) tels que (y, x) et (x, z) existent. L'algorithme est donc le suivant :

Fermeture-Transitive(G)

1. *Pour tout s faire*
2. *Pour tout* s'
   Si (s', s) *existe alors*
   Pour tout s''
   si (s, s'') *existe alors ajouter l'arc* (s', s'')

Exercice 19 *La complexité de cet algorithme est en* $O(n^3)$

6.3 Calcul du plus court chemin

Ce genre d'algorithme est utilisé pour calculer le meilleur chemin à prendre pour se rendre d'un lieu à un autre. Cependant, cette problématique peut s'étendre à tout système pour lequel on essaye d'optimiser un certain coût.

Intuitivement, si on se pose le problème, on s'aperçoit rapidement que la combinatoire est énorme.

Formalisation

Pour formaliser le problème, il faut considérer un graphe orienté pondéré $G = (S, A)$. Le poids du chemin $p =< v_0, v_1, ..., v_n >$ est la somme des poids des arcs qui le constituent. On se propose donc de minimiser :

$$w(p) = \sum_{i=1}^{n} w(v_{i-1}, v_i)$$

On définit le poids du plus court chemin entre u et v par :

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{s'il existe un chemin de } u \text{ à } v \\ \infty & \text{sinon} \end{cases}$$

L'algorithme de parcours en largeur d'abord est un algorithme de recherche de plus court chemin qui fonctionne sur des graphes non-pondérés (arc de poids unitaire)
On se place dans le cas où l'on souhaite calculer le plus court chemin à origine unique. Cependant, il existe des cas où l'on cherche plutôt :
- le plus court chemin à destination unique : en inversant le sens de chaque arc, on se ramène au problème précédent
- le plus court chemin pour un couple de sommets donné : si on résout le problème à origine unique, on résout aussi ce problème
- le plus court chemin pour tout couple de sommets : on peut explorer chaque sommet, mais il existe des algorithmes spécifiques que nous verrons ultérieurement.

Arcs de poids négatif Dans certains cas, il arrive que le poids associé à des arcs soit négatif. S'il n'existe aucun circuit de poids négatif accessible à partir de l'origine, alors le poids du chemin reste bien défini. Par contre, s'il existe un tel circuit, le poids du chemin n'est pas défini puisqu'il suffit de tourner dans ce circuit pour faire décroître à loisir le poids du chemin global.

Relâchement La plupart des algorithmes de calcul du plus court chemin sont basés sur de techniques dites *de relâchement*. C'est une méthode qui permet de faire diminuer progressivement une borne supérieure $d[v]$ sur les poids et de les faire tendre vers la valeur optimale.
L'initialisation s'effectue de la manière suivante :

Initialisation(G,s)

1. *Pour chaque sommet* $v \in S[G]$
   faire $d[v] \leftarrow \infty$
   $\pi \leftarrow Nil$
2. $d[s] \leftarrow 0$

La procédure de relâchement, à proprement parler consiste à estimer de plus en plus précisément le plus court chemin.

Relâcher(u,v,w)

1. *Si* $d[v] > d[u] + w(u, v)$
   alors $d[v] \leftarrow d[u] + w(u, v)$
   $\pi \leftarrow u$
2. $d[s] \leftarrow 0$

Algorithme de Ford Le principe de l'algorithme Ford consiste à utiliser directement la procédure d'initialisation suivie de la procédure de relâchement sur les sommets dont l'estimation est finie, jusqu'à convergence.

Ford(G,s)

1. *Initialisation(G,s)*
2. *Tant que modifications*
   Pour tout sommet t
   Si $d[t] < \infty$ *alors*
   Pour tout successeur u de t
   Relâcher(t,u,w)
3. *Fin Tant Que*

La complexité de l'algorithme est en $O(S \times A)$:
– Initialisation en $\Theta(S)$
– Il y a $|S|-1$ passage dans la boucle qui est de complexité $O(A)$ (au maximum $|A|$ arcs).

Algorithme de Dijkstra Le principe de cet algorithme est de tenir à jour une liste de sommets "fixés", c'est à dire, dont les poids finaux de plus court chemin à partir de l'origine ont déjà été calculés. A chaque itération, l'algorithme choisit dans les sommets non-fixés le sommet dont l'estimation de plus court chemin est la plus faible, le fixe et relâche les arcs partant de ce sommet.

Dijkstra(G,s)

1. *Initialisation(G,s)*
2. $E \leftarrow \emptyset$
3. $F \leftarrow S[G]$
4. *Tant que* $F \neq \emptyset$
   Faire
   $u \leftarrow$ *Extraire-Min(F)*
   $E \leftarrow E \cup \{u\}$
   pour tout successeur v *de* u
   Relâcher(u,v,w)
5. *Fin Tant Que*

La complexité de l'algorithme de Dijkstra dépend de la manière dont a été implémentée l'extraction du minimum.
Si on considère que les files de priorités sont stockées sous la forme de tableaux linéaires, le temps d'exécution de Extraire-Min(X) est en $O(S)$. Cette extraction est réalisée $|S|$ fois. Chaque sommet et chaque arc est examiné une et une seule fois. Il y a donc $|A|$ itération de la boucle pour, consommant uniquement $O(1)$. Le temps d'exécution est donc en $O(S^2 + A) = O(S^2)$.
Si le tas est peu dense, il est plus rapide d'implémenter la file l'attente sous forme d'un TAS. Dans ce cas, chaque opération d'extraction est en $O(\ln S)$. La procédure Relâcher est modifiée pour permettre un temps de calcul en $O(\ln S)$. Le temps de calcul est alors en $O((S + A) \ln S)$.
On peut encore diminuer ce temps de calcul en implémentant une file de Fibonacci. L'algorithme est alors en $O(S \ln S + A)$

Exemple

Voici un exemple du fonctionnement de l'algorithme de Dijkstra.

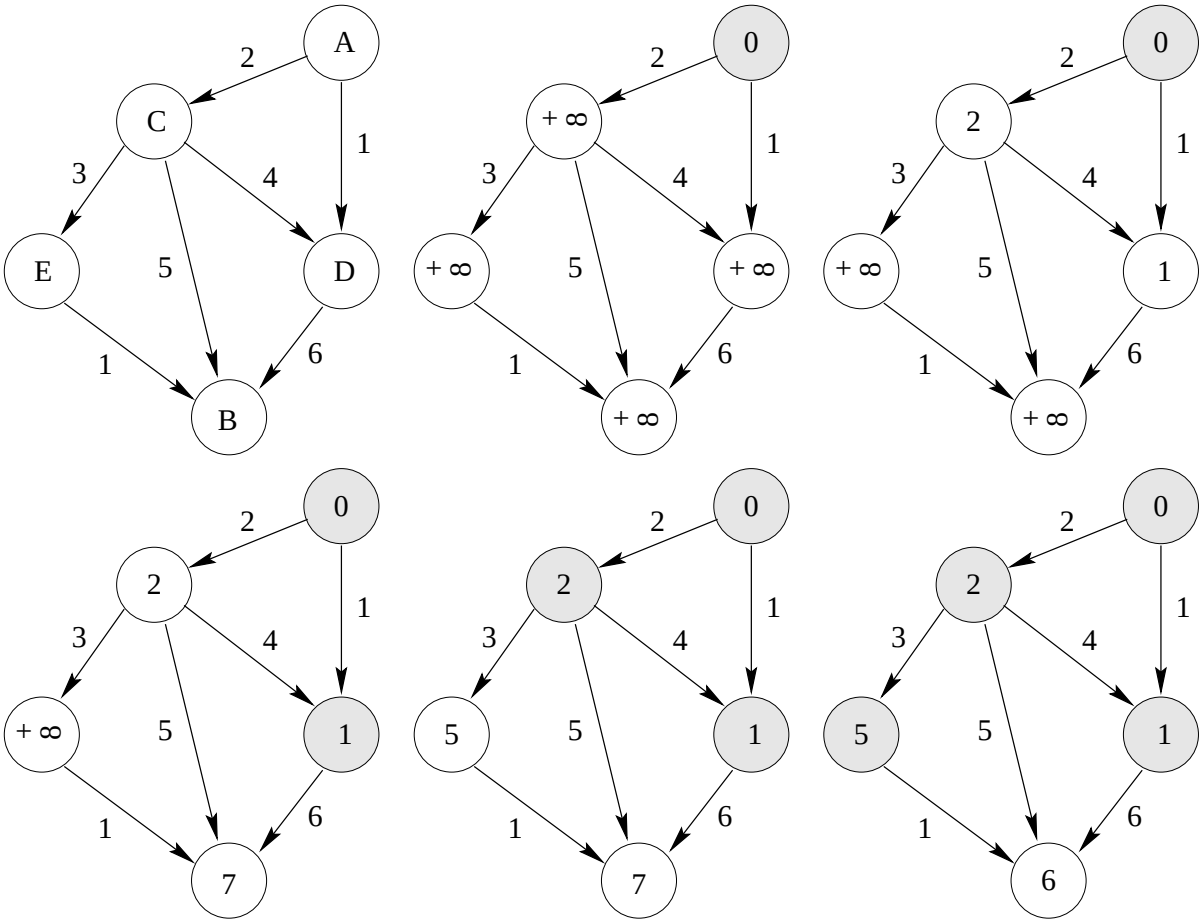

Fig. 9. *Exemple de fonctionnement de l'algorithme de Dijkstra*

Algorithme de Bellmann Comme pour l'algorithme de Dijkstra, le principe de cet algorithme est de tenir à jour une liste de sommets "fixés". A la différence de Dijkstra, par contre, l'algorithme ne choisit pas dans les sommets non-fixés le sommet dont l'estimation de plus court chemin est la plus faible, mais *un* sommet dont les prédécesseurs sont fixé ; le fixe et relâche les arcs partant de ce sommet.

Bellmann(G,s)

1. *Initialisation(G,s)*
2. $E \leftarrow \emptyset$
3. $F \leftarrow S[G]$
4. *Tant que* $F \neq \emptyset$
   Faire
   u *est un sommet* $\in F$ *dont tous les successeurs* $\in E$
   $E \leftarrow E \cup \{u\}$
   pour tout successeur v *de* u
   Relâcher(u,v,w)
5. *Fin Tant Que*

La complexité de l'algorithme est en $O(S \times A)$.

6.4 Calcul du plus long chemin

Il n'est pas possible d'adapter directement les algorithmes de plus court chemin dans le cas d'un graphe possédant des cycles. Par contre, on peut adapter les algorithmes de calcul du plus court chemin pour calculer le plus long chemin pour des graphes acycliques. Dans ce cas, on estime le plus long chemin plutôt que le plus court.

Ford(G,s)

1. *Initialisation(G,s)*
2. *Tant que modifications*
   Pour tout sommet t
   Si $d[t] < \infty$ *alors*
   Pour tout successeur u de t
   Si $d(u) < d(t) + w(t,u)$ *ou* $d(u) = +\infty$ *alors*
   $d(u) \leftarrow d(t) + w(t,u)$
3. *Fin Tant Que*

6.5 Ordonnancement

Les algorithmes de calculs de plus court chemin, ont été utilisés pour résoudre des problèmes d'ordonnancement.
A la base, un problème d'ordonnancement consiste à trouver comment exécuter un ensemble de taches de durée donnée en respectant des contraintes de précédence entre taches.

En pratique, les durées d'exécution peuvent être imprécises (données par des lois de probabilité) et soumises à d'autres contraintes temporelles (commencer cette tache à telle date, il est impossible d'effectuer ces taches en parallèle...).

Définitions

Une tache est **critique** si son exécution ne peut être différée ou allongée sans retarder la fin des travaux

la **marge totale** est le temps maximal donc cette tache peut être différée ou allongée sans retarder la fin des travaux.

Il existe 2 principales méthodes d'ordonnancement qui diffère principalement par leur mode de représentation : la méthode MPM et la méthode PERT.

Méthode des potentiels/taches (MPM)

Dans la méthode **MPM** (Method of Project Management) chaque tache est associée à un sommet, et chaque contrainte temporelle à un arc, valué par la durée de la tache source. Par ailleurs, on ajoute un tache **début** et un arc allant de début à chaque sommet sans prédécesseur et une tache **fin** et un arc allant de chaque sommet sans successeur à fin.

NB : le problème n'a de solution que si le graphe est sans circuit !

On peut calculer la date **de début au plus tot** pour chaque tache comme le poids du plus long chemin allant de **début** à cette tache. La durée minimale des travaux est le poids maximal d'un chemin allant de début à fin.

De même, le calcul de la date de **début au plus tard** (différence entre la durée minimale de la tache et la durée minimale des travaux à accomplir) s'effectue en calculant le plus long chemin allant de la tache à fin.

Une solution de l'ordonnancement consiste à calculer au plut tot. Dans ce cas, chaque tache est planifiée pour sa date de début au plus tot ; la marge libre de chaque tache est alors la différence entre la plus petite des dates de début au plus tot de ses successeurs, et sa date de début au plus tot augmentée de sa durée.

Une autre solution consiste à planifier chaque tache pour sa date de début au plus tard.

La représentation se fait souvent sous la forme d'un diagramme de Gantt.

La méthode des potentiels/étapes (PERT)

Dans la méthode **PERT** (Project Evaluation and Review Technique), chaque tache est représentée par un arc valué par la durée de la tache. Les sommets correspondent alors aux étapes de la réalisation du projet. Chaque tache à un sommet début et un sommet fin. Les contraintes de précédence sont appliquées en ajoutant des taches fictives de durée nulle reliant la tache contraignante au début de la tache contrainte. La aussi, on crée un sommet début et un sommet fin.

On calcule la date **de début au plus tot** 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 tache située sur ce chemin est une tache

critique.
Le calcul de la date de **début au plus tard** s'effectue en calculant le plus long chemin allant de l'étape à fin.
La marge totale d'une tache $s \to t$ est la différence entre la date de fin au plus tard de t et la somme de la date de début au plus tot de s et de la durée de la tache.

6.6 Problèmes de flots

Un flot est un modèle de réseau de communication qui peut modéliser aussi bien les écoulements dans les conduites d'eaux que les réseaux électriques. Pour qu'un graphe orienté soit un flot, il faut qu'il existe 2 sommets particuliers appelés *source* et *puits* pour lesquels il existe un arc unique (arc de retour) qui part du puits et arrive à la source et tel que tout sommet respecte la loi de conservation des flux (appelée *loi de Kirchhoff* dans les circuits électriques.
En fait, les problèmes qui se posent dans les réseaux est de savoir comment, pour un réseau à capacité connue, régler le flot pour qu'il soit maximum.

6.7 Définitions

On se donne un graphe **borné**, c'est-à-dire, orienté, simple et antisymétrique, dont les arcs sont étiquetés par des intervalles à bornes entières voire infinie.
On appelle **capacité** la borne supérieure de l'intervalle. On suppose que le graphe possède une source et un puits (appelés aussi *entrée* et *sortie).*
On appelle **flot compatible** un flot pour lequel le poids de l'arc est dans l'intervalle définit (voir figure 10).
Le problème du **flot maximal** consiste à trouver un flot compatible de **valeur** maximale (somme des flots sur tous les sommets ; c'est).
On dit qu'un arc est **saturé** par un flot compatible si la valeur du flot sur l'arc est sa capacité. On dit qu'un arc est **antisaturé** la valeur du flot sur l'arc est la borne inférieure de l'intervalle des valeurs possibles.

6.8 Algorithme de Ford-Fulkerson

Le principe de cet algorithme est de trouver un chemin menant de la source au puits capable d'améliorer le flot, de l'augmenter et de recommencer jusqu'à ce qu'il n'existe plus de chemin pouvant être maximisé.

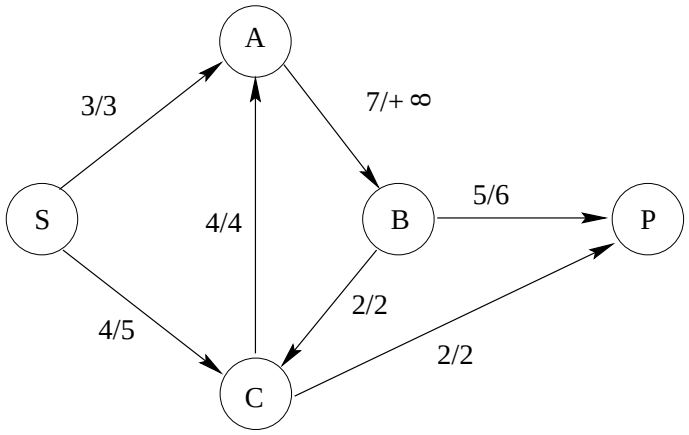

Fig. 10. *Exemple de flot*

Ford-Fulkerson(G)

tant que le flot n'est pas maximal
  marquer + *le sommet entrée*
  tant qu'on marque des sommets
    pour tout sommet marqué s *non encore traité*
      pour tout arc $s \rightarrow t$
        si t *n'est pas marqué et* $s \rightarrow t$ *n'est pas saturé alors*
          marquer t *par* $(+, s)$
      fin pour
      pour tout arc $t \rightarrow s$
        si t *n'est pas marqué et* $t \rightarrow s$ *n'est pas antisaturé alors*
          marquer t *par* $(-, s)$
      fin pour
  tant que
  si le puits n'est pas marqué alors le flot est maximal
  sinon augmenter le flot
fin tant que

Pour augmenter le flot on calcule d'abord de combien il peut être augmenté avec l'algorithme suivant :

calcule-augmentation-flot(G)

$a \leftarrow +\infty$
$t \leftarrow puits$
  tant que $t \neq source$
    si t *est marqué* $(+, s)$
      $a \leftarrow \min(a, \max(s \rightarrow t) - f(s \rightarrow t))$
    fin si
    si t *est marqué* $(-, s)$
      $a \leftarrow \min(a, f(t \rightarrow s) - \min(t \rightarrow s))$
    fin si
  $t \leftarrow s$
  fin tant que

On augmente ensuite le flot de la valeur calculé sur tout le parcours :

augmentation-flot(G)

$t \leftarrow puits$
  tant que $t \neq source$
    si t *est marqué* $(+, s)$
      augmenter de a *le flot sur* $s \rightarrow t$
    fin si
    si t *est marqué* $(-, s)$
      diminuer de a *le flot sur* $t \rightarrow s$
    fin si
  $t \leftarrow s$
  fin tant que

Marquage de Ford-Fulkerson sur le flot donné en exemple.

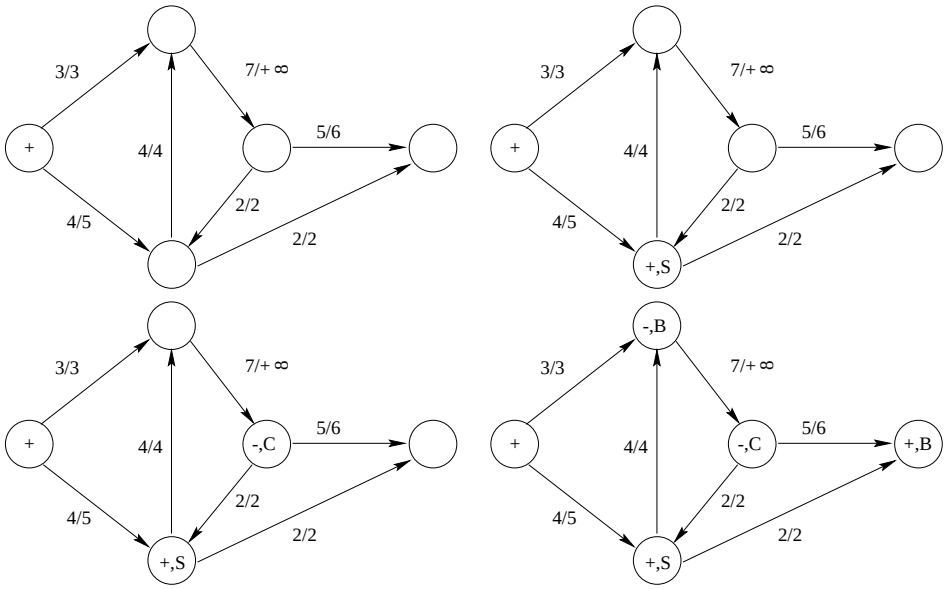

Fig. 11. *Marquage de Ford-Fulkerson*

En utilisant le calcul de l'augmentation on a :

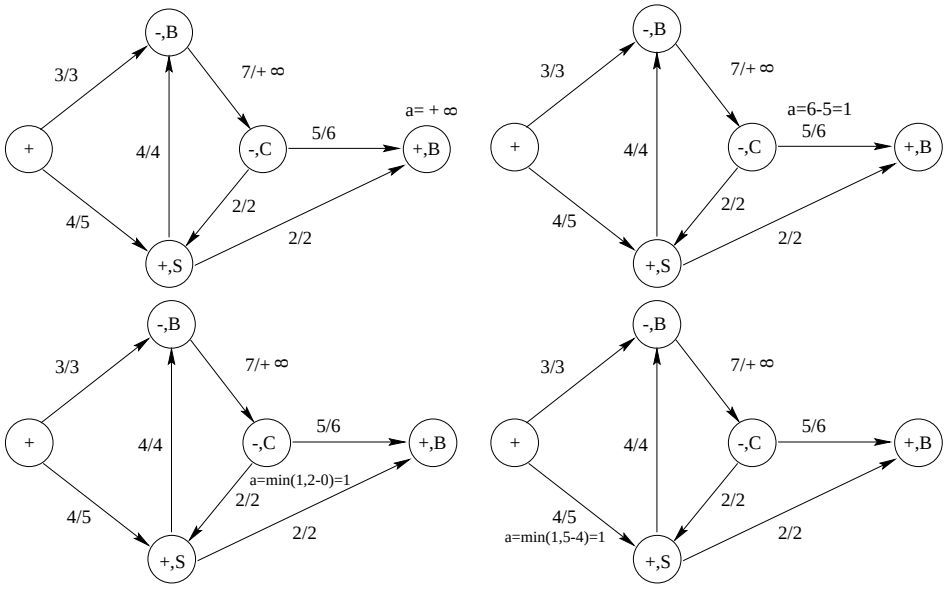

Fig. 12. *Calcul de l'augmentation*

On augmente ensuite de 1 le flot sur le chemin marqué et on obtient le graphe suivant :

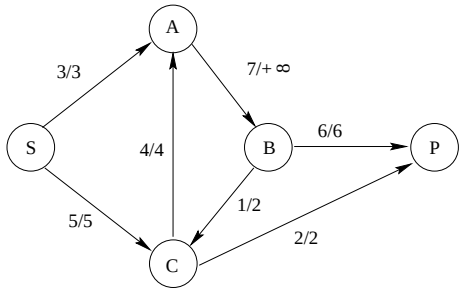

Fig. 13. *Flot augmenté*

Pour vérifier que le flot est maximal, appliquons de nouveau le marquage de Ford-Fulkerson.

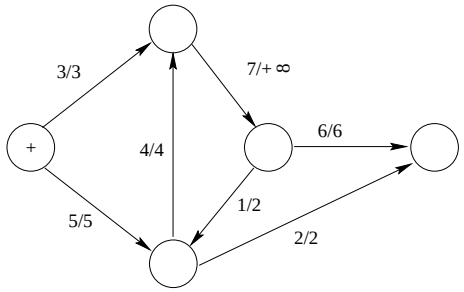

Fig. 14. *Flot maximal ?*

On voit alors que les 2 flots partant de la source sont saturés. Le flot est donc maximal.

Complexité

L'algorithme de Ford-Fulkerson se termine au plus tard quand on a marqué les $|S|$ sommets ; par ailleurs, on explore au plus 2 fois chaque arc (pour son origine et son extrémité). La complexité est donc en $O(\max(|S|, |A|))$.
La complexité des algorithmes de calcule et d'augmentation du flot est en $O(|S|)$ puisque on traite au maximum 2 fois de suite un chemin comportant au plus $|S|$ sommets.

6.9 Coupe minimale

On appelle **coupe minimale** du graphe associée à un flot, une partition des sommets en 2 ensembles E et F tels que tout arc du premier ensemble vers le deuxième soit saturé et tout arc en sens inverse soit antisaturé.

Théorème 3 ***Théorème de caractérisation :*** *si un flot f admet une coupe minimale (E, F), alors ce flot est maximal*

démonstration

Pour tout flot compatible, loi de conservation est vérifiée quelque soit le sommet, et en particulier, pour tous les sommets de M. On a donc :

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

Or, source appartient à M alors que puits n'en fait pas partie. D'où :

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

Donc :

$$c(arcretour) \leq \sum_{s\to t}^{s\notin M, t\in M} \max(s \to t) - \sum_{s\to t\neq arcretour}^{s\in M, t\notin M} \min(s \to t)$$

Par définition de la coupe minimale, les arcs de M vers N sont saturés, et les arcs de N vers M sont antisaturés par le flot f. On atteint donc l'égalité :

$$f(arcretour) = \sum_{s\to t}^{s\notin M, t\in M} \max(s \to t) - \sum_{s\to t\neq arcretour}^{s\in M, t\notin M} \min(s \to t)$$

On en déduit donc que le flot f est maximal parmi les flots compatibles. ■

Dans l'exemple précédent, on peut déterminer la coupe minimale à partir de la figure 14. Celle-ci est composée de la source, d'une part, et de l'ensemble des autres sommets d'autre part.

6.10 Graphe d'écart associé à un flot compatible

Le graphe d'écart G_f associé à un flot compatible f sur un graphe G, est un graphe dont les sommets sont ceux de G mais dont les arcs sont à double sens :

à tout arc $a = s \to s'$ de G on associe

si $f(a) < \max(a)$ (arc non-saturé), un arc $s \to s'$ de capacité $\max(a) - f(a)$

si $f(a) > \min(a)$ (arc non-antisaturé), un arc $s' \to s$ de capacité $f(a)$

Pour notre exemple, le flot d'écart associé est :

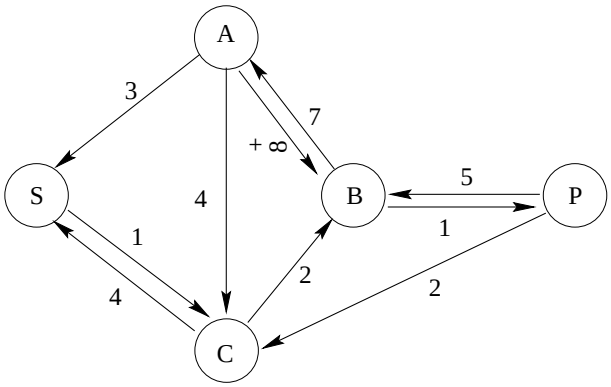

Fig. 15. *Graphe d'écart*

6.11 Preuve de l'algorithme de Ford-Fulkerson

Si la sortie est marquée à la fin de l'algorithme, on augmente le flux d'une quantité a, en respectant à la fois la conservation de flux (on fait varier conjointement flux entrant et flux sortant) et la compatibilité du flot (l'augmentation est calculée comme valeur minimale des variations possibles sur les arcs du chemin).
Par ailleurs, si la sortie n'est pas marquée, alors, si M est l'ensemble des sommets marqués et N l'ensemble des sommets non marqués, (M, N) est une coupe minimale et donc le flot est maximal d'après le théorème de caractérisation.

6.12 Interprétation de l'algorithme de Ford-Fulkerson

En fait, l'algorithme de marquage consiste à trouver un chemin possible de la source vers le puits dans le graphe d'écart associé au flot. L'optimisation revient à diminuer le plus possible les capacités des arcs du chemin marqué en augmentant les arcs réciproques.

6.13 Flot maximal à coût minimal

Une étape supplémentaire dans le calcul de flots maximaux, consiste à considérer que chaque utilisation d'un arc de transmission correspond à un certain coût de transfert. Les problèmes qui sont maintenant traités consistent à trouver, parmi les flots maximaux, celui dont le coût est minimal.

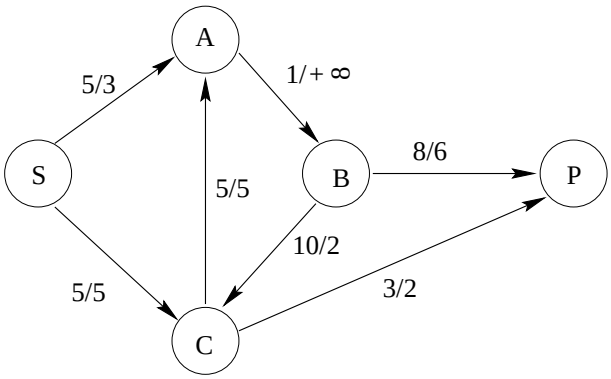

Fig. 16. *Exemple de graphe étiqueté à la fois avec des coûts et des capacités*

Définitions

On se donne un graphe G borné, dont les arcs sont pondérés par des capacités et par des coûts (réels positifs). On note $\max(a)$ la capacité et $c(a)$ le coût associé à l'arc a.
Le problème consiste à trouver, parmi les flots maximaux, celui dont le coût total $c(f) = \sum_{a \in A} f(a) \cdot c(a)$ est minimal.
Tel quel, ce problème ne connaît pas de solution algorithmique de complexité polynomiale. Le principe est donc d'optimiser progressivement une solution initiale.

Algorithme de Roy

Le principe de cette méthode consiste à partir d'un flot nul et de choisir à chaque étape la modification dont le coût est minimal, tout en augmentant le flot. L'arrêt se produit lorsque le flot est maximal.

Graphe d'écart associé à un flot à coûts Le principe est de construire un graphe d'écart de flot compatible, puis d'associer à chaque arc $s \to s'$.
A chaque arc $s \to s'$ on associe : $c(a)$ si $a = s \to s'$ et $-c(a)$ si $a = s' \to s$.

Théorème 4 ***Critère d'optimisation :*** *Soit f, un flot compatible. Il existe un flot, de même valeur de f sur l'arc de retour, et de coût strictement inférieur à celui de f ;* ssi *il existe dans le graphe d'écart, un circuit dont la somme des coûts des arcs est strictement négative.*

L'algorithme Le principe de l'algorithme est de partir du graphe d'écart, et de choisir, parmi tous les chemins possible de la source au puits, celui dont le coût est minimal On augmente ensuite le flot sur ce chemin et on recommence jusqu'à ce que le flot soit maximal.

Exemple Exemple de fonctionnement de l'algorithme de Roy.

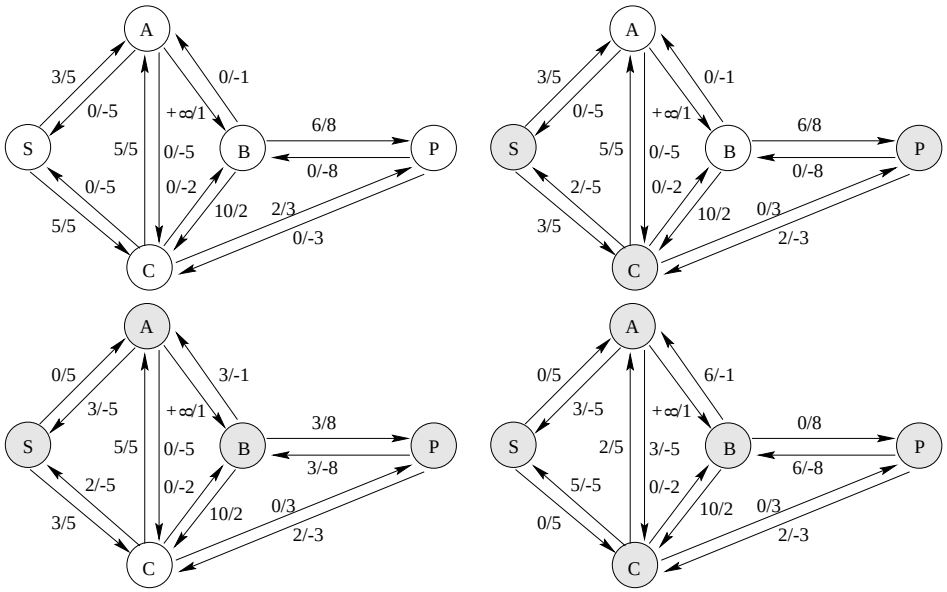

Fig. 17. *Exemple de fonctionnement de l'algorithme de Roy*

6.13.1 Arbre recouvrant de poids minimal

Ce type de problème consiste à construire un arbre à partir d'un graphe en supprimant progressivement les arétes de manière à ce que la somme des poids soit minimale.
Cette technique est utilisée pour simplifier le routage, que ce soit celui d'un circuit électronique ou d'un réseau informatique.

Algorithme de Kruskal

Kruskal

1. *soit G' le graphe partiel de G sans arete*
2. *pour i de 1 à n (n nombre d'aretes)*

   si l'arete a_i ne crée pas de cycle dans G'
   alors ajouter a_i à G'
   finsi
3. *fin pour*

Il faut cependant se poser le problème du test de la présence d'un cycle. En fait, on part de la constatation du fait que pour que (x, y) forme un cycle, il faut que ses extrémités soient reliées par une chaine.
Le principe de l'algorithme de test est donc de tenir à jour une liste des composantes connexes et de tester si (x, y) appatiennent bien à 2 composantes différentes.

test-cycle

1. *Initialisation*
   pour i de 1 à n (n nombre de sommets)
   $I_i \leftarrow i$
   fin pour
2. $Test(x, y)$
   si $I_x = I_y$
   alors on ne garde pas l'arete
   sinon
   on garde l'arete
   $I \leftarrow I_y$
   pour i de 1 à n (nombre de sommets)
   si $I_i = I$ alors $I_i \leftarrow I_x$
   fin pour

Complexité Le test de cycle est en $|S|$. Le coeur de l'algorithme est constitué d'une itération sur les aretes. L'algorithme est donc en $O(|S| \times |A|)$.

Le principe de cet algorithme est de partir d'un graphe sans arete constitué des sommets de G : $G' = (S, \emptyset)$.

Sollin

tant que G' *comporte plus d'une composante connexe*

choisir une composante connexe C *de* G'

ajouter à G' *une arete de poids minimal reliant* C *à une autre composante connexe de* G'

fin tant que

Complexité La complexité de l'algorithme est due à :
– la boucle interne en $|S|$
– le maintient de la liste des composantes connexes (voir plus haut)
– la recherche de l'arete de poids minimal en $|A|$
Au total, l'algorithme est en $O(|S| \times \max(|S|, |A|)$.

Bibliographie

[Cormen et al., 1994] Cormen, T., Leiserson, C., and Rivest, R. (1994). *Introduction à l'algorithmique*. Dunod.

[M.Gondran and M.Minoux, 1979] M.Gondran and M.Minoux (1979). *Graphes et algorithmes*. Eyrolles.