Chapitre 5

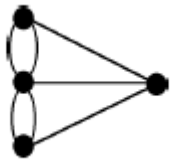
Les graphes et leurs algorithmes

1. Introduction :

La notion de graphe est une structure combinatoire permettant de représenter de nombreuses situations rencontrées dans des applications faisant intervenir des mathématiques discrètes et nécessitant une solution informatique. Circuits électriques, réseaux de transport (ferrés, routiers, aériens), réseaux d'ordinateurs, ordonnancement d'un ensemble de tâches sont les principaux domaines d'application où la structure de graphe intervient.

2. Quelques exemples d'application dans les graphes

Exemple 1. Le problème des ponts de Königsberg
La ville de Königsberg comprenait 4 quartiers, séparés par des ponts. Les habitants de Königsberg se demandaient s'il était possible, en partant d'un quartier quelconque de la ville, de traverser tous les ponts sans passer deux fois par le même et de revenir à leur point de départ.
Le plan de la ville peut se modéliser à l'aide d'un graphe ci-dessous, les quartiers sont représentés par les 4 sommets, les 7 ponts par des arêtes :

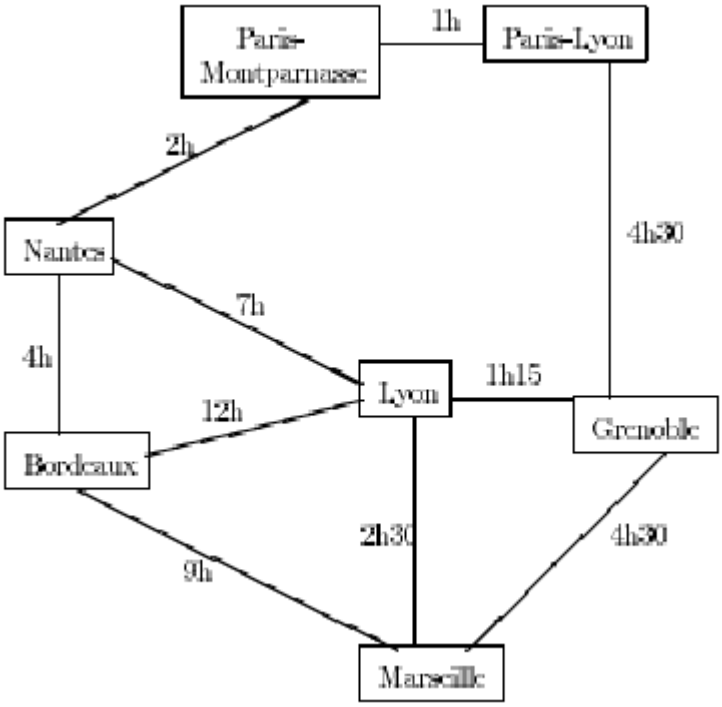


Exemple 2. Choix d'un itinéraire
Sachant qu'une manifestation d'étudiants bloque la gare de Poitiers, et connaissant la durée des trajets suivants :

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

Question : Comment faire pour aller le plus rapidement possible de Bordeaux à Grenoble ?

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



Il s'agit de déterminer, dans ce graphe, le plus court chemin entre Bordeaux et Grenoble.

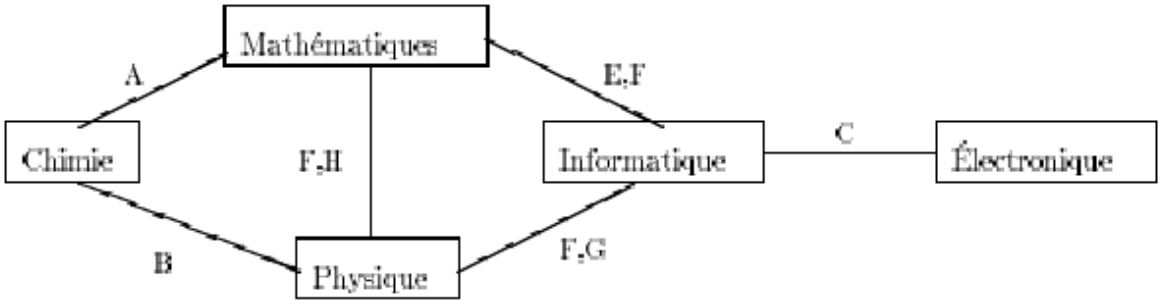
Exemple 3. Organisation d'une session d'examens
Des étudiants A, B, C, D, E et F doivent passer des examens dans différentes disciplines, chaque examen occupant une demi-journée :

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

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

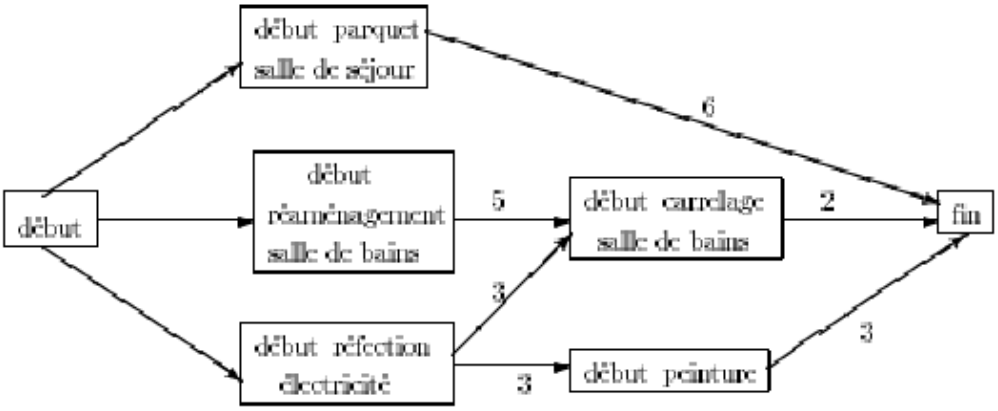
On peut représenter chacune des disciplines par un sommet, et relier par des arêtes les sommets correspondant aux examens incompatibles (ayant des étudiants en commun) :

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



Exemple 4. Planification de travaux

Pour rénover une maison, il est prévu de refaire l'installation électrique (3 jours), de réaménager (5 jours) et de carreler (2 jours) la salle de bains, de refaire le parquet de la salle de séjour (6 jours) et de repeindre les chambres (3 jours), la peinture et le carrelage ne devant être faits qu'après réfection de l'installation électrique. Si la rénovation est faite par une entreprise et que chacune des tâches est accomplie par un employé différent, quelle est la durée minimale des travaux ?
On peut représenter les différentes étapes de la rénovation sur un graphe dont les arcs sont étiquetés par la durée minimale séparant deux étapes



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

De manière générale, un graphe permet de représenter simplement la structure, les connexions, les cheminements possibles d'un ensemble complexe comprenant un grand nombre de situations, en exprimant les relations, les dépendances entre ses éléments. En plus de son existence purement mathématique, le graphe est aussi une structure de données puissante pour l'informatique.

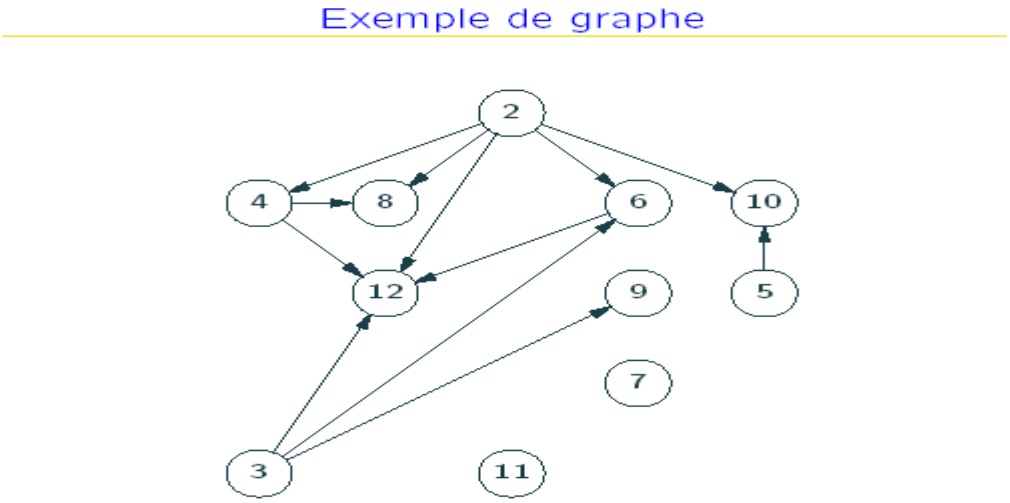
3. Définition

Un graphe G est constitué de 2 ensembles V et E. V est un ensemble non vide de sommets. E est un ensemble de paires de sommets de V. Ces paires sont appelés arêtes. $V(G)$ et $E(G)$

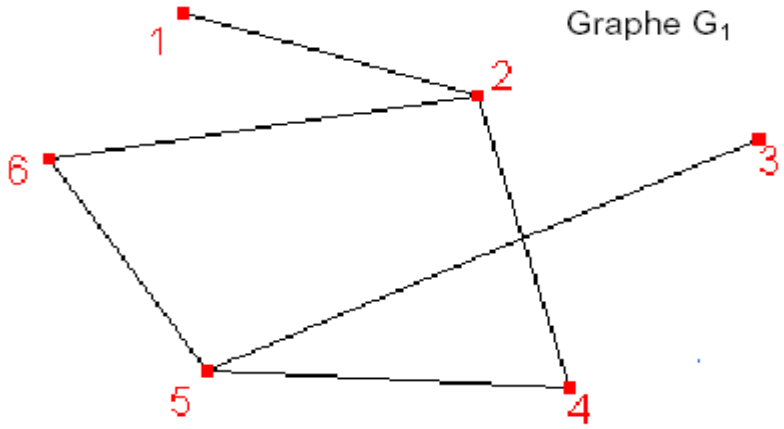
représentent, respectivement l'ensemble des sommets et d'arêtes du graphe *G*. On écrit aussi $G = (V,E)$ pour représenter le graphe.

Dans un graphe non orienté, la paire de sommets représentant une arête n'est pas ordonnées. Autrement dit, les (v_1, v_2) et (v_2, v_1) représentent la même arête.

Dans un graphe orienté (**diagraphe**), chaque arête est représentée par la paire orientée $<v_1, v_2>$; v_1 étant l'origine et v_2 l'extrémité de l'arête dans ce cas, on ne parle plus d'arête mais d'arc. Par conséquent, $<v_2, v_1>$ et $<v_1, v_2>$ représentent deux arcs différents. La figure ci-dessous représentent deux graphes.



graphe orienté



graphes non orienté

4. Terminologie : avant de continuer, il est utile de présenter la terminologie (même réduite) utilisée dans la théorie des graphes.

Terme	**Signification**
adjacence	deux arcs sont adjacents s'ils ont une extrémité commune; deux sommets sont adjacents s'il existe un arc, ou une arête, les reliant
arc	couple (x,y) dans un graphe orienté
arête	nom d'un arc, dans un graphe non orienté
boucle	arc reliant un sommet à lui-même
chaîne	Nom d'un chemin dans un graphe non orienté ; séquence d'arcs avec une extrémité commune dans un graphe orienté.
chemin	suite d'arcs connexes reliant un sommet à un autre. Par exemple (a;b) (b;c) (c;d) (d;b) (b;e) est un chemin reliant a à e ; on le note (a,b,c,d,b,e). Un chemin est une chaîne, la réciproque étant fausse
chemin eulérien	désigne un chemin simple passant une fois et une seule par toutes les arêtes du graphe ; il n'existe pas toujours…
chemin hamiltonien	désigne un chemin simple qui passe une fois et une seule par chaque sommet
chromatique (nombre)	c'est le nombre minimal de couleurs nécessaires pour colorier tous les sommets du graphe sans que deux sommets adjacents aient la même couleur ; pour les cartes de géographie, le nombre chromatique est 4 ; ce fut, en 1976, la première démonstration informatique d'un théorème mathématiques, sous réserve qu'il n'y ait pas eu de bug dans le programme… il aura fallu 700 pages de listing !
circuit	chemin dont l'origine et l'extrémité sont identiques
Connexité	Un graphe est connexe s'il existe toujours une chaîne, ou un chemin, entre deux sommets quelconques. Par exemple le plan d'une ville doit être connexe.
Complet	un graphe est complet si quels que soient deux sommets distincts, il existe un arc (ou une arête) les reliant dans un sens ou dans l'autre
Cycle	nom d'un circuit dans un graphe non orienté ; dans un graphe orienté, un cycle est une chaîne dont l'extrémité initiale coïncide avec l'extrémité finale
degré d'un sommet	nombre d'arête issues d'un sommet dans un graphe non orienté ; nombre d'arcs arrivant ou partant d'un sommet dans un arc orienté ; on peut vérifier facilement que la somme des degrés de tous les sommets, est donc le double du nombre des arêtes (puisque chacune est comptée deux fois).
Diamètre	le diamètre d'un graphe est la plus grande chaîne (chemin) de toutes reliant deux sommets quelconques du graphe
Distance	la distance entre deux sommets d'un graphe est la plus petite longueur des chaînes, ou des chemins, reliant ces deux sommets.
graphe orienté	désigne un graphe où le couple (x,y) n'implique pas l'existence du couple (y,x) ; sur le dessin, les liens entre les sommets sont des flèches
graphe simple	désigne un graphe non orienté n'ayant pas de boucle ni plus d'une arête reliant deux sommets. Sur le dessin, les liens entre les sommets sont des segments, et on ne parle alors plus d'arcs mais d'arêtes ; tout graphe orienté peut donc être transformé en graphe simple, en remplaçant les arcs par des arêtes
longueur d'un chemin (ou d'une chaîne)	nombre d'arcs du chemin (ou d'arêtes de la chaîne)
Ordre d'un graphe	nombre de sommets du graphe
prédécesseur	Dans l'arc (x;y), x est prédécesseur de y

Rang	le rang d'un sommet est la plus grande longueur des arcs se terminant à ce sommet
sous graphe	le graphe G' est un sous graphe de G si l'ensemble des sommets de G' est inclus dans l'ensemble des sommets de G, et si l'ensemble des arcs de G' est égal au sous-ensemble des arcs de G reliant entre eux tous les sommets de G' ; on a donc retiré de G certains sommets, et tous les arcs adjacents à ces sommets ;
Stable	soit un graphe G (E ; R), et F un sous-ensemble de sommets. On dit que F est un sous ensemble stable de E s'il n'existe aucun arc du graphe reliant deux sommets de F.
successeur	dans l'arc (x;y), y est successeur de x
Arcs(arêtes) incidents à un sommet	nombre d'arcs (arêtes) dont une extrémité part de ce sommet.
Arcs **incidents** à un sommet vers **l'intérieur**	nombre d'arcs (arêtes) dont l'extrémité finale arrive dans ce sommet
Arcs **incidents** à un sommet vers **l'extérieur**	nombre d'arcs (arêtes) dont l'extrémité d'origine part de ce sommet

Un sommet **pendant** est un sommet de degré **1**.

Un **pont** est une arête telle que sa suppression disconnecte le graphe en question.

Demi-degré intérieur (degré entrant) d'un sommet *x*, $d^{-}(x)$, est le nombre d,arcs entrant *x*,

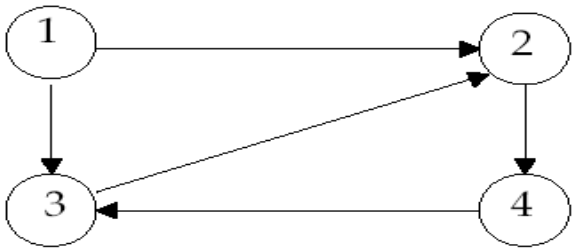
Demi-degré extérieur (degré sortant) d'un sommet *x*, $d^{+}(x)$, est le nombre d'arcs sortant de *x*.

Le degré d(x) est égal à $(d^{+}(x) + d^{-}(x))$

Un graphe valué (pondéré) est un graphe où les arcs (arêtes) possèdent un poids.

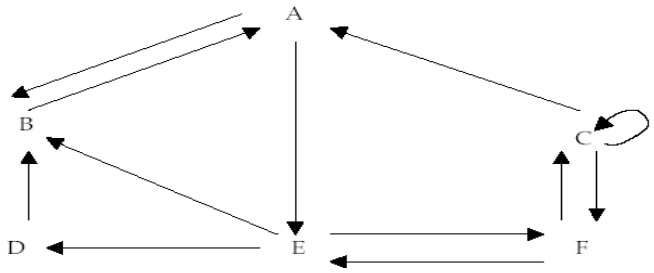
Graphe partiel
Soit G = (V, E) un graphe. Le graphe G' = (V, E') est un **graphe partiel** de G, si E' est inclus dans E. Autrement dit, on obtient G' en enlevant une ou plusieurs arête graphe G.

Exercice



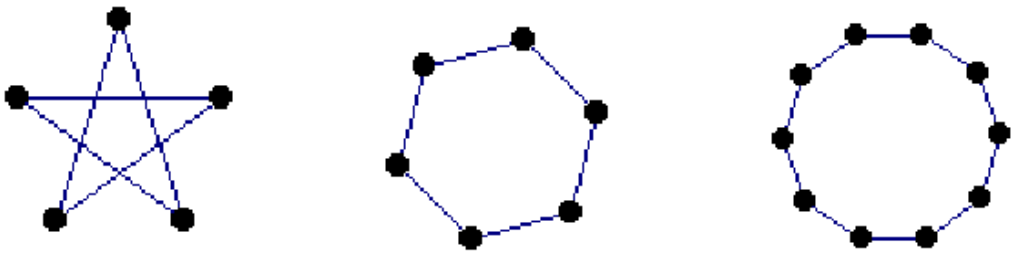
Quels sont les cycles et les circuits de ce graphe?

Exercice :



• successeurs (C)?
• prédécesseurs (B)?
• arcs incidents à E vers l'extérieur?

Exercice: Caractériser les graphes de diamètre 1. Trouver le diamètre des graphes ci-dessous.



5. Représentation d'un graphe

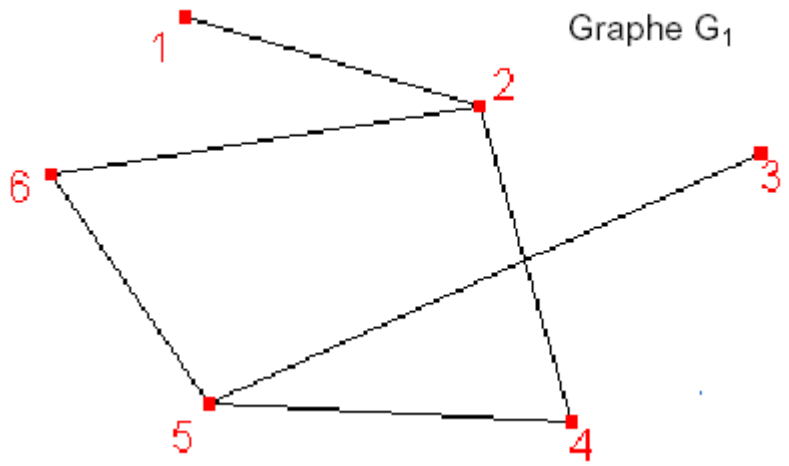
a) Utilisation de matrices

• L'ensemble des sommets du graphe n'évolue pas.
• On utilise un tableau de booléens, dite matrice d'adjacence, de dimension n x n où n= |V|.

```
⇒ type graphe = array [1..n, 1..n] of boolean;
```

L'élément d'indice i et j est vrai si et seulement si il existe un arc entre les sommets i et j.

Exemple: La matrice d'adjacence du graphe



est comme suit:

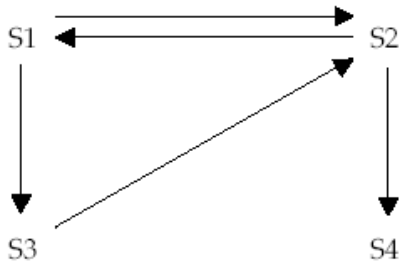
$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

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

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

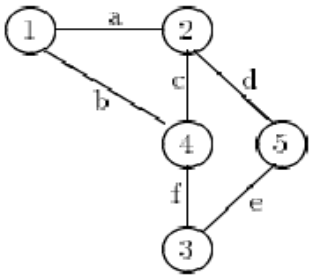
Exercice

a) quelle est la matrice d'adjacence du graphe suivant?
b) que se passe-t-il si le graphe est simple (non orienté)?
c) comment procéder dans un graphe valué?
d) comment traiter tous les successeurs du sommet S1?
e) écrire une fonction qui retourne le demi-degré extérieur de S3?



b. Matrice d'incidence

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



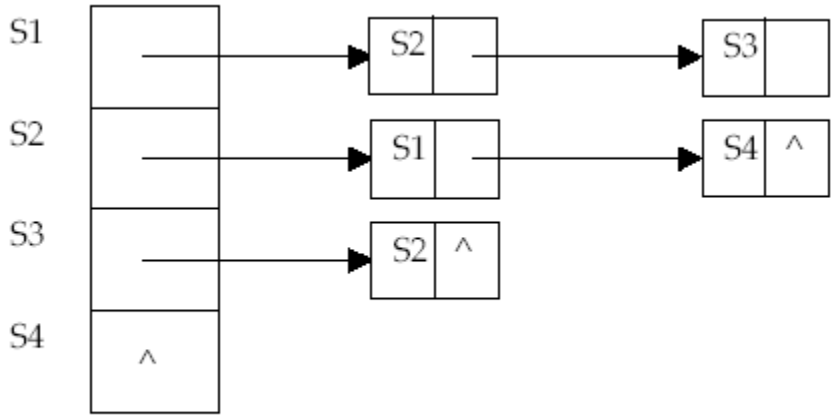
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

représentation par la matrice d'incidence.

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

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

b) Utilisation des listes d'adjacence



Exercice

a) écrire, en C-C++ ou en Java, les déclarations qui correspondent à cette liste
b) faire la distinction entre un graphe où le nombre de sommets évolue
c) comment faire si le graphe est valué
d) avantages et inconvénients de cette solution
e) écrire une fonction qui retourne le demi-degré extérieur de x.

Quelques Propriétés dans les graphes

Théorème : Un graphe ayant V sommets possède au plus n(n-1)/2 arêtes.

Preuve : en classe !!

Pour tout graphe :

(1) La somme des degrés des sommets est égale au double du nombre d'arêtes.

(2) La somme des degrés des sommets est paire

(3) Le nombre de sommets de degré impair est pair

Démonstration

1. Chaque arête du graphe relie exactement 2 sommets, une arête augmente donc de 2 la somme des degrés des sommets (1 pour chaque sommet).
2. Il est clair que, d'après ce qui précède, la somme des degrés est paire.
3. Soit N le nombre de sommets de degré impair. Ces degrés valent $2k_1+1$, …, $2k_N+1$. Les autres valent $2k'_1, \cdots, 2k'_m$. La somme des degrés vaut

$$S = 2(k_1 + \cdots + k_N + k'_1 + \cdots + k'_m) + N$$

et cette somme est paire d'après (1) d'où le résultat

Un exemple

Est-il possible de dessiner, dans le plan, 9 segments de telle manière que chacun en coupe exactement 3 autres ?

En représentant **chaque segment** par un sommet d'un graphe et en reliant deux sommets par une arête lorsque les deux segments se coupent, le problème se ramène à la recherche d'un graphe à 9 sommets où tous les sommets sont d'ordre 3.

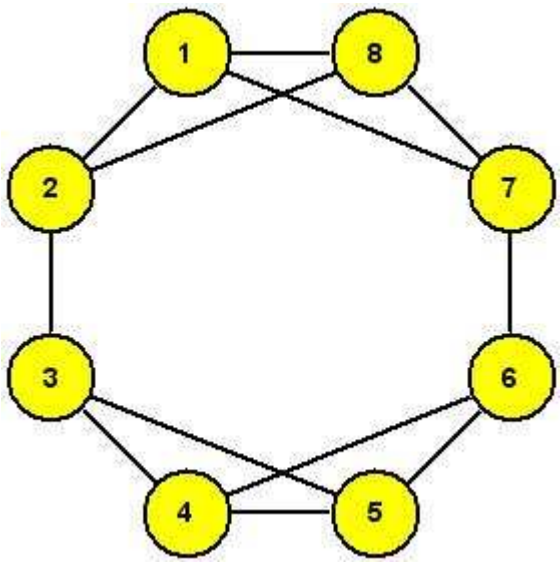
Or le nombre de sommets de degrés impairs est nécessairement pair, 9 étant impair, le problème n'a pas de solutions.

En modifiant l'énoncé comme suit:

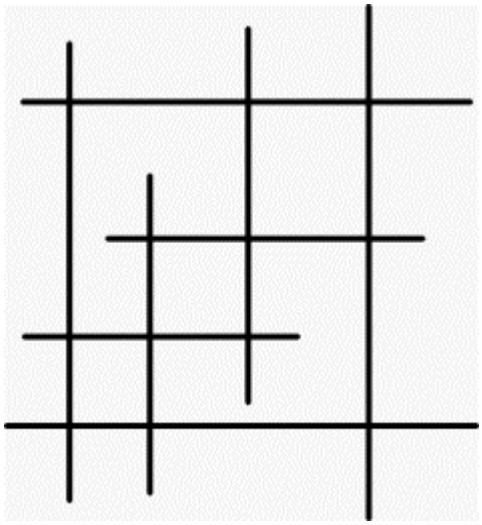
Est-il possible de dessiner, dans le plan, 8 segments de telle manière que chacun en coupe exactement 3 autres ?

Il n'y a plus de contradiction avec le théorème précédent mais il n'est pas sûr que le problème admette une solution. Quelques tâtonnements permettent cependant de répondre positivement à la question posée

Le graphe ci-contre répond à la question posée.



le graphe correspondant



Un dessin possible

Théorème Dans un graphe simple G, il y a au moins 2 sommets distincts ayant le même degré.

Preuve: Soit n le nombre de sommets de G et soit $d(x)$ le degré du sommet x dans G. On a toujours $0 \leq d(x) \leq n-1$. Si tous les degrés sont distincts, toutes les valeurs de $d(\cdot)$ sont atteintes, en particulier 0 et n-1. Soit a un sommet de degré $n-1$. Alors les n-1 sommets restants sont extrémités d'une arête ayant a comme autre extrémité. Les sommets ont donc tous un degré non nul, en contradiction avec l'hypothèse.

Soit A un graphe à n sommets.

Théorème - Les propriétés suivantes sont équivalentes.

(a) A est un arbre.

(b) A ne contient aucun cycle et possède n-1 arêtes.

(c) A est connexe et possède n-1 arêtes.

(d) A est connexe, et chaque arête est un pont.

(e) Deux sommets quelconques de A sont reliés par un unique chemin.

(f) A est acyclique mais l'addition d'une quelconque nouvelle arête crée exactement un cycle.

Démonstration – L'équivalence se fera comme suit : (a) ⇒ (b) ⇒ (c) ⇒ (d) ⇒ (e) ⇒ (f) ⇒ (a)

On raisonne par induction sur n. L'équivalence des six propriétés est évidente pour n= 1.

(a) ⇒ (b) : il faut montrer que A possède n-1 arêtes. Supprimons une arête de A. Cela le disconnecte (sinon A contiendrait un cycle) en deux sous-graphes B et C tous deux connexes et sans cycle, donc possédant, d'après l'hypothèse de récurrence, p -1 et q -1 arêtes (où $p+q=n$). Il en résulte que A possède (p -1) + (q -1) + 1 = n -1 arêtes.

(b) ⇒ (c) : il faut montrer que A est connexe. S'il ne l'était pas, chacune de ses composantes serait connexe et sans circuit. Donc, par hypothèse de récurrence, elle possèderait q_{i-1} arêtes. On aurait donc

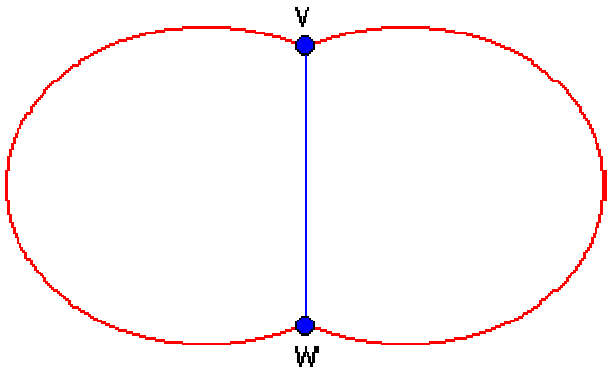
$$n-1=q_1-1+\cdots+q_k-1=n-k$$

(où k est le nombre de composantes). Par conséquent, k=1, d'où la contradiction.

(c) ⇒ (d) : il faut montrer que si l'on enlève une arête, le graphe n'est plus connexe, ou encore qu'un graphe possédant n sommets et n-2 arêtes n'est pas connexe. S'il l'était, il aurait forcément un sommet pendant (sommet de degré 1), puisque la somme des degrés est égale à 2n-4, alors qu'elle vaut au moins 2n pour un graphe sans sommet pendant. Retirant ce sommet et l'arête qui lui correspond, on aurait un graphe à n-1 sommets et n-3 arêtes qui serait connexe, ce qui est faux d'après l'hypothèse de récurrence (ou alors : qui entraîne, en recommençant le même raisonnement, l'existence d'un graphe connexe à deux sommets et zéro arête).

(d) ⇒ (e) : puisque A est connexe, ce chemin existe. S'il y en avait deux, A contiendrait un circuit et la suppression d'une arête quelconque de ce circuit laisserait A connexe, en contradiction avec l'hypothèse que toutes les arêtes sont des ponts.

(e) ⇒ (f) : si A contenait un circuit, deux sommets sur ce circuit seraient reliés par deux chemins. Si l'on ajoute une arête vw, puisque v et w étaient déjà reliés dans A, on crée un circuit. On ne peut en créer qu'un : si l'on en créait deux, on aurait la figure suivante



et A aurait déjà contenu auparavant le circuit mis en évidence en rouge.

(f) ⇒ (a) : si A n'était pas connexe, l'adjonction d'une arête entre deux composantes ne pourrait créer de circuit. □

Parcours dans les graphes

Étant donnée la racine d'un arbre, l'une des tâches les plus communes est la traversée de ce dernier en visitant tous ses noeuds dans un ordre bien défini. Dans le chapitre sur les arbres, nous avons défini trois procédures: *postfixe, infixe* et *prefixe*, pour effectuer cette visite. De la même manière, dans un graphe $G = (V,E)$ et un sommet v dans $V(G)$, nous sommes intéressés à visiter tous les sommets qu'on peut atteindre à partir de v. On peut rencontrer deux types de problèmes dans les parcours de graphes:

1) Il se peut que tous les autres sommets ne soient pas accessibles à partir du sommet de départ. Tel est le cas des graphes non connexes (voir tableau ci-dessous pour définition).
2) Le graphe peut contenir des cycles, et on doit s'assurer que l'algorithme ne rentre pas dans une boucle infinie.

Pour remédier aux deux problèmes cités, les algorithmes de parcours de graphes maintiennent en général un bit de marquage pour chaque sommet du graphe. Initialement, le bit est à 0 pour tous les sommets du graphe. Le bit de marquage est mis à 1 quand un sommet est visité pendant le parcours. Si un sommet marqué est rencontré à nouveau pendant le parcours, il n'est pas visité une deuxième fois. Ceci permet d'éviter les cycles.
Une fois l'algorithme de parcours terminé, on peut vérifier si tous les sommets ont été visité en consultant le tableau de marquage.

La stratégie de parcours d'un graphe est alors comme suit :

Nous allons discuter, dans ce qui suit, deux procédures: exploration en largeur d'abord et exploration en profondeur d'abord.

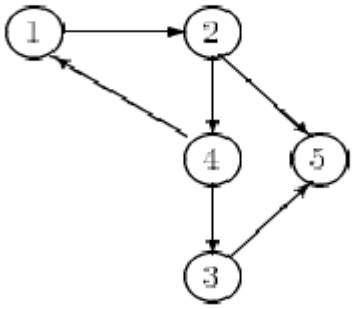
Parcours en largeur d'abord (Breadth First Search, BFS)

La stratégie de cette exploration est comme suit: on commence au sommet v et le marquer comme étant visité. L'ensemble des sommets adjacents à v sont visités. Ensuite, les sommets non visités de ces éléments sont à leur tour visités, et ainsi de suite, jusqu'à visiter tous les sommets.

```
void BFS (int debut) // debut étant le sommet de départ
{
```

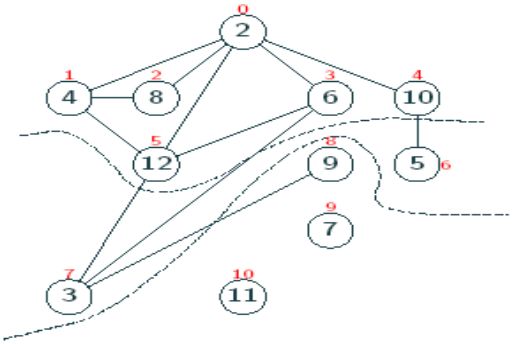
```
enfiler (debut) ;
marquer[debut] = 1;
tant que (file non vide)
{    s = defiler () ;
     // traitement nécessaire sur le sommet s
     pour tout w adjacent à s faire
          si (Marque[w] == 0 )
          {
               Marquer[w] = 1;
               enfiler (w);
          }
}}
```

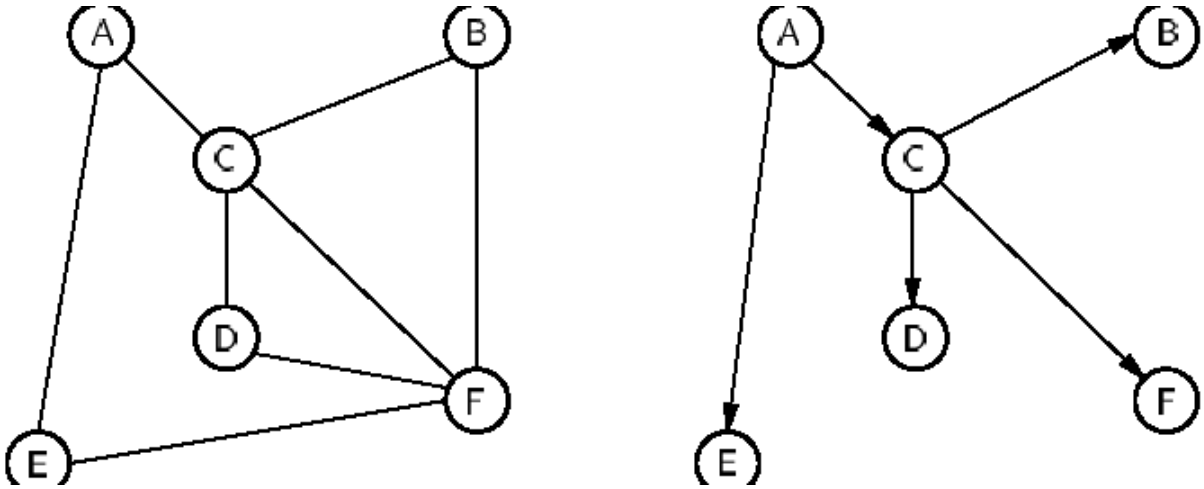
Exemple sur un graphe



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

Un autre exemple: le parcours en BFS est indiqué par les numéros rouges.





Ordre de parcours : A,C,E,D,F,B

Complexité : Il est clair que cette complexité est donnée par celle de la boucle while. Comme un sommet n'est jamais visité plus d'une fois, cette boucle est au plus répétée n fois. Le nombre d'itérations de la boucle for est proportionnel au degré de chaque sommet u+1 (le +1 car même si degre(u)=0, la boucle fera quand même le test) . En additionnant sur les sommets, on obtient :

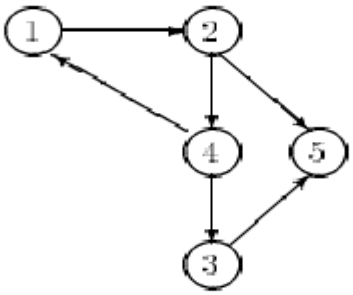
$$T(n) = O(n + \sum_{u \in V} \deg re(u) + 1) = O(2n + 2e) = O(n + e)$$

Parcours en profondeur d'abord (Depth First Search, DFS)

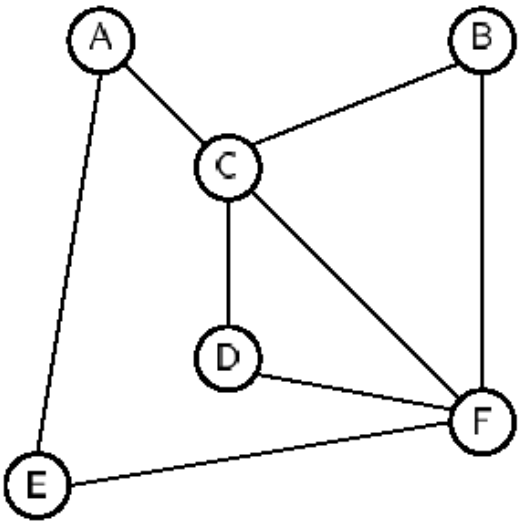
La stratégie de cette méthode est comme suit: On commence par *v* qu'on marque comme visitée. Ensuite, un sommet adjacent *w* adjacent à *v* est choisi et un parcours en profondeur d'abord est effectué à partir de *w*. Quand un sommet *u* est atteint tel que tous les sommets qu'ils lui sont adjacents sont visités, alors on choisit un sommet adjacent du dernier sommet visité et on recommence la procédure jusqu'à ne plus avoir de sommets adjacents non visité. En terme algorithmiques, on obtient ce qui suit :

void *DFS*(int *v*) // v étant le sommet de départ {
pour i allant de 1à n faire
initialiser marquer [i] = 0;
marquer [v] = 1
Pour chaque sommet *w adjacent à v faire*
  si marquer [*w***]** = 0 alors *DFS*(*w*)
finpour }

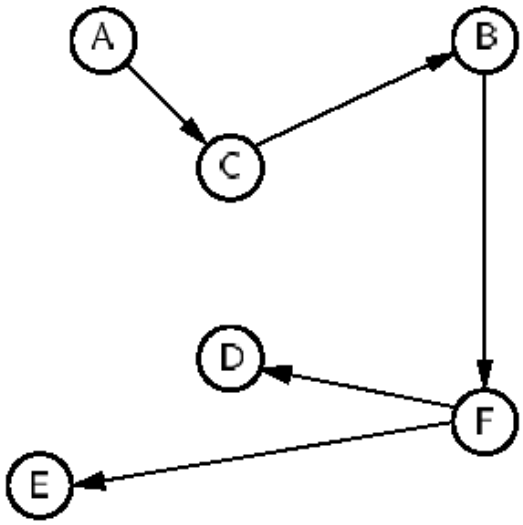
Exemple sur un graphe



Ordre de visite : 1, 2, 4, 3, 5;



(a)



(b)

Ordre de visite: A, C, B, F, D, E.

Complexité

Observons que DFS examine chaque sommet au plus une fois. Pour chaque sommet u, un temps proportionnel à (degreexterne(u)+1) est nécessaire pour traiter les sommets adjacents de u (pourquoi le +1 ?). Par conséquent, la complexité de DFS est :

$$T(n) = O(n + \sum_{u \in V} \deg reexeterne(u) + 1) = O(n + \sum_{u \in V} \deg reexterne + n) = O(2n + e) = O(n + e)$$

Quelques applications de parcours de graphes

Application 1 : Accessibilité

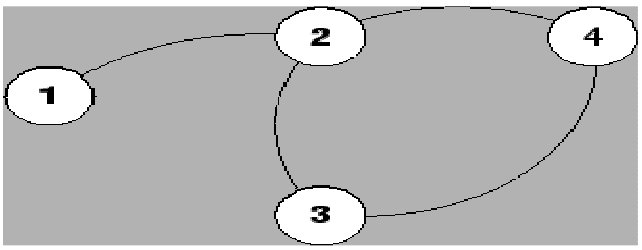
Pour connaître les sommets accessibles depuis un sommet donné d'un graphe (orienté ou non), il suffit de faire un parcours en profondeur à partir de ce sommet, en marquant les sommets visités :

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

visiter un sommet k:
  si k n'a pas encore été visité
     alors
        marquer k;
        visiter tous les sommets adjacents de k
    finsi
```

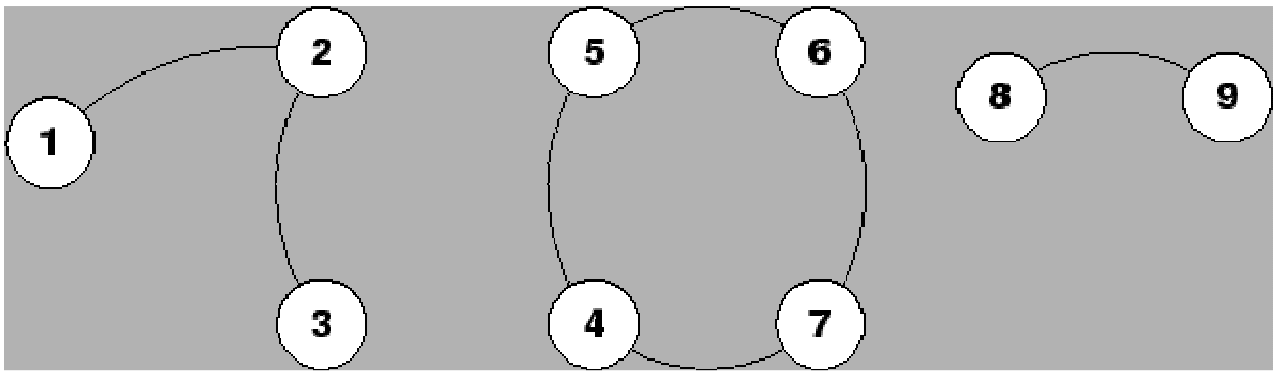
Application 2 : composantes connexes

Définition : Un graphe est connexe si et seulement si, quelque soit deux sommets, ils sont reliés par une chaîne (chemin).



Définition: Une composante connexe est un sous graphe connexe de taille maximale.

Remarque : il est toujours possible de partitionner un graphe en composantes connexes. Par exemple, sur la figure ci-dessous, le graphe est scindé en 3 composantes connexes.



Théorème Soit G un graphe simple à n sommets. Si G possède k composantes connexes, le nombre m de ses arêtes vérifie

$$n-k \leq m \leq (n-k)(n-k+1)/2$$

Preuve:

n-k ≤m ?

Chaque composante connexe de G est un graphe simple connexe. Il est clair que le plus petit (en nombre d'arêtes) graphe connexe est un arbre (pourquoi ?). Il en résulte que chacune des k composantes connexes possède au moins n_1-1 arêtes. Autrement dit :

$$m \geq n_1 - 1 + n_2 - 1 + ... + n_k - 1 = n - k$$

***m* ≤(*n*-*k*)(*n* - *k* +1)/2?**

Nous pouvons supposer que chaque composante est un graphe complet. Supposons que nous ayons deux composantes C_i et C_j de G qui soient des graphes complets possédant respectivement n_i et n_j sommets, avec $n_i \geq n_j > 1$. Remplaçons C_i et C_j par des graphes complets à n_i+1 et n_j -1 sommets respectivement (en d'autres termes. Dans une telle opération, le nombre de composantes et celui de sommets reste inchangé. En revanche, le nombre d'arêtes augmente de:

$$n_i(n_i+1)/2 - n_i(n_i-1)/2 + (n_j-1)(n_j-2)/2 - n_j(n_j-1)/2 = n_i - n_j + 1 > 0$$

Le nombre m sera donc maximal quand il n'y aura plus qu'une seule composante ayant plus d'un sommet, c'est-à-dire quand on aura k-1 sommets isolés et un graphe complet à n - k + 1 sommets et, par conséquent, (n - k)(n - k + 1)/2 arêtes.

Comment déterminer les composantes connexes d'un graphe?

En appliquant d'une manière répétitive DFS ou BFS sur tous les sommets non encore visités. Il est clair qu'une composante connexe est constituée du sous graphe dont les sommets sont visités par un seul appel à DFS ou BFS

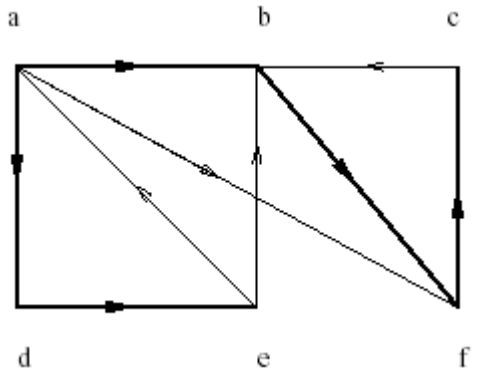
for i = 1 **to** *n* **do**
  marquer(i) = 0
fin pour

pour i = 1 à n
  si *marquer*(*i*) = 0 **alors** *DFS*(*i*) (ou alors BFS);
  restituer le sous graphe visité par DFS (formant une composante connexe)
fin pour

Complexité : Si G est représenté par sa liste adjacente, alors le temps total pris par DFS est en $O(e)$, e étant le nombre d'arêtes. La restitution peut être complétée en temps $O(e)$ si DFS garde une liste de

tous les sommets nouvellement visités. Comme la boucle **pour** est en $O(n)$, la complexité totale, pour générer toutes les composantes connexes est en $O(n + e)$.

Exercice : quelles sont les composantes connexes du graphe ci-dessous



Application 4 : Graphe orienté sans circuit

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

```
Pour tout sommet s
        Si s n'a pas encore été visité
           alors
                visiter s
        finsi
finpour;
répondre "il n'a pas de cycle"
visiter s :
   si s est dans la liste critique
      alors
          répondre ``il y a un cycle ";
         retour
     sinon
          ajouter s à la liste critique;
          visiter tous les successeurs de s;
          enlever s de la liste critique
  finsi
```

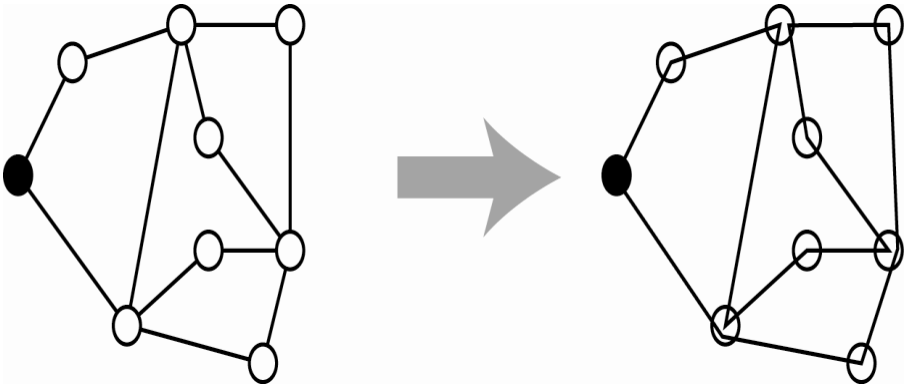
Applications dans les graphes simples

Application 1 : chemins eulériens et hamiltoniens

Définition: une chaîne eulérienne est une chaîne passant une et une seule fois par toutes les arêtes du graphe. Si le sommet initial et le sommet final sont confondus, on parle alors de cycle eulérien.

Définition : Un graphe est dit eulérien s'il admet un chemin ou un cycle eulérien.

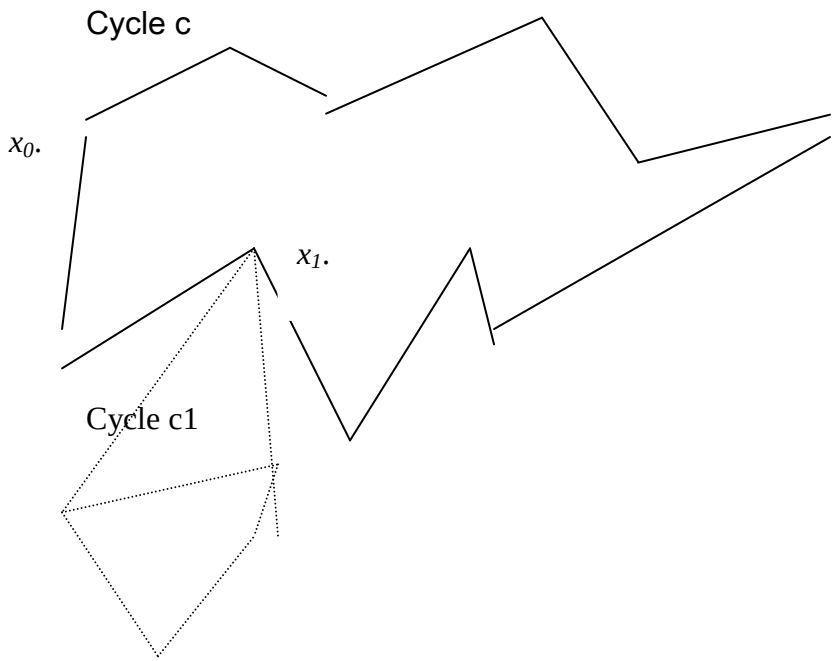
Exemple : Un camion de ramassage des ordures part du dépôt représenté en noir ci-dessous et doit passer sur chaque route du réseau ci-dessous pour effectuer la collecte des déchets. Comment doit-il s'y prendre ? On cherche ici un cycle eulérien.



Théorème d'Euler : Un graphe simple connexe est eulérien si et seulement si pour tout sommet *x*, *d(x)* est pair.

Preuve : Supposons G admet un cycle eulérien, soit alors *c* un cycle eulérien et *x* un sommet de G. Le cycle *c* contient toutes les arêtes de *G*, donc toutes les *d(x)* arêtes ayant *x* comme extrémité. Lors d'un parcourt de *c* on arrive en *x* autant de fois qu'on en repart, chaque arête de *G* étant présente une et seule fois dans *c*, *d(x)* est nécessairement un nombre pair.

Réciproquement, choisissons un sommet arbitraire x_0, puis considérons l'autre extrémité de cette arête : ce deuxième sommet étant de degré pair, on peut en repartir par une autre arête et atteindre ainsi un autre sommet à nouveau. Et ainsi de suite, comme le graphe admet un nombre fini d'arêtes, jusqu'a ce que la chaîne formée se referme sur elle pour former un cycle c (si elle ne se referme pas, alors l'extrémité aura un degré impair). Si ce cycle est eulérien, on arrête. Sinon, chacune des composantes du graphe restant vérifie les propriétés du théorème: elle est finie, connexe et chacun des sommets est de degré pair.



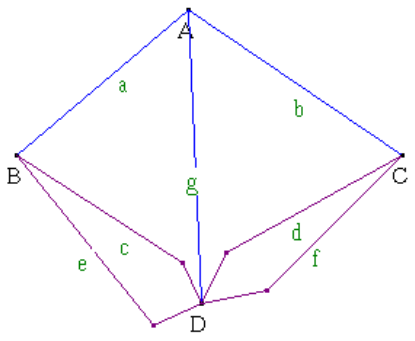
Comme *G* est connexe, chacune des composantes restantes possède au moins un sommet commun avec le cycle c. Soit x_1 un tel sommet. Construisons alors, de la même manière que précédemment, un cycle *c1* dans cette composante à partir de x_1. Rallongeons le cycle c_1 en insérant à partir du sommet *x1* le cycle *c1* pour former un cycle c'_1 de x_0 à x_0.

Si ce cycle c'_1 possède toutes les arêtes de *G*, c'_1 est le cycle eulérien cherché. Sinon, on continue ce processus, qui se terminera car les sommets du graphe *G* sont en nombre fini.

Exercice : démontrer le théorème suivant :

Un graphe connexe G admet une chaîne eulérienne si et seulement le nombre de sommet de degré impair est soit 0 ou 2.

Exemple d'application le plus connu: le problème dit « des 7 ponts de Königsberg » mentionné en introduction:



Comment partir de A et revenir à A, en passant par chaque pont une fois et une seule ? Le problème revient à chercher un cycle eulérien de A à A.

Ce graphe est bien connexe (tout sommet peut être relié à un autre) et on a

Sommet	A	B	C	D
Degré	3	3	3	5

Il y a quatre sommets de degré impair : il n'existe donc pas de chemin Eulérien. Autrement dit, l**e problème des 7 ponts de Königsberg n'a donc pas de solution**.

Exercice Lequel des deux graphes ci-dessous admet une cycle eulérien**.**

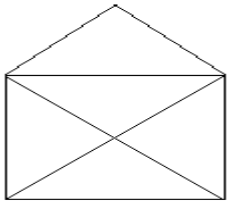


figure 1

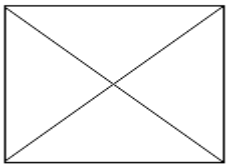


figure 2

L'algorithme d'Euler permet de trouver explicitement, lorsque cela est possible, une chaîne eulérienne dans un graphe. Les conditions d'existence d'un parcours eulérien sont données comme suit :

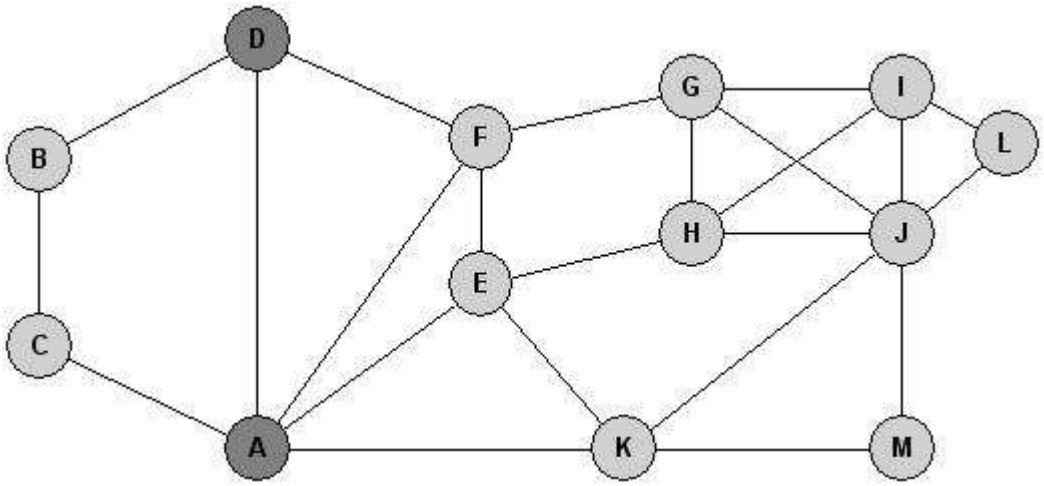
- Un graphe connexe admet une **chaîne eulérienne** si le nombre de sommets de degré impair est 0 ou 2.
- Un graphe connexe admet un **cycle eulérien** si tous ses sommets sont de degré pair.

Algorithme d'Euler

1	• S'il y a exactement **deux** sommets de degré impair, on construit une chaîne quelconque joignant ces deux sommets. • Si tous les sommets sont de degré pair, on construit un cycle à partir d'un sommet. Dans les deux cas, on **marque** toutes les arêtes utilisées. **Si** toutes les arêtes sont marquées, la chaîne est eulérienne. **Sinon** on passe à l'étape 2.
2	On **insère** dans cette chaîne, une **chaîne fermée** ayant pour origine l'un des sommets déjà utilisés, ne contenant pas deux fois la même arête et ne contenant aucune arête déjà parcourue. Les arêtes utilisées sont marquées. L'insertion consiste à remplacer un sommet X par une chaîne fermée X…X
3	S'il reste des arêtes non marquées, revenir à l'étape 2.

Exemple de mise en œuvre sur le graphe ci-dessous : *départ en* D *et arrivée en* A

Bien entendu, il faut s,assurer au préalable que ce graphe admet une chaîne eulérienne : calculer pour cela l'ordre de chaque sommet.



	Chaînes fermées	Chaîne eulérienne
Étape 1		**D** B C **A**
Étape 2	A E K A	D B C **A E K A**
Étape 3	D F A D	**D F A D** B C A E K A
Étape 4	F G H E F	D **F G H E F** A D B C A E K A
Étape 5	G I L J I H J G	D F **G I L J I H J G** H E F A D B C A E K A
Étape 6	K J M K	D F G I L J I H J G H E F A D B C A E **K J M K** A

Note : Cet algorithme ne conduit pas à une solution unique: le choix de la chaîne de départ et des chaînes fermées est arbitraire.

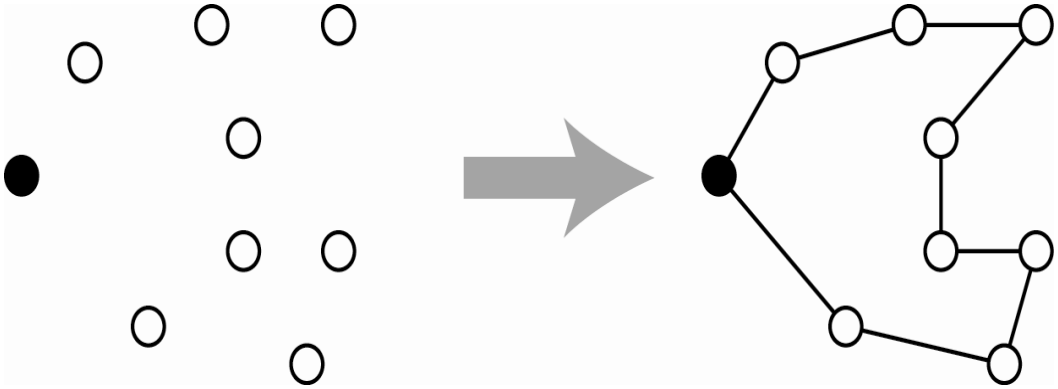
Exercice : déduire un algorithme déterminant un cycle eulérien (en supposant que tous les degrés des sommet soient pairs). Quelle est sa complexité, en représentant le graphe comme :

1. matrice d'adjacence
2. matrice d'incidence
3. liste d'adjacence.

Graphes hamiltoniens

Définition : une chaîne hamiltonienne est une chaîne passant une et une seule fois par tous les sommets du graphe. Si le sommet initial et le sommet final sont confondus, on parle alors de cycle eulérien.

Exemple : Un voyageur de commerce part tous les matins de son domicile représenté en noir ci-dessous et doit rendre visite à un ensemble de clients représentés en blanc, puis retourner à son domicile. Comment doit-il s'y prendre pour minimiser la distance totale parcourue. (On suppose que les distances entre toutes les paires de clients ainsi qu'entre les clients et le domicile du voyageur de commerce sont connues). On cherche ici un cycle hamiltonien de longueur minimale.



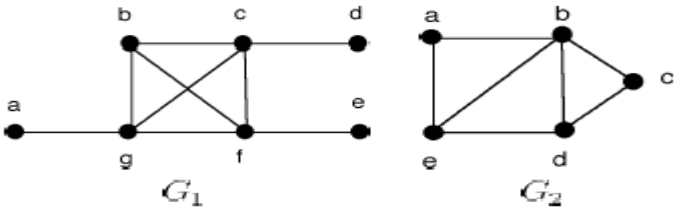
Contrairement aux graphes eulériens, il n'existe pas de caractérisation simple des graphes hamiltoniens. On peut cependant énoncer quelques propriétés et conditions suffisantes.

1. Un graphe possédant un sommet de degré 1 ne peut être hamiltonien.
2. Si un sommet dans un graphe est de degré 2, alors les deux arêtes incidentes àce sommet doivent faire partie du cycle hamiltonien.
3. Les graphes Kn sont hamiltoniens. (Kn étant un graphe complet de n sommets)

Théorème (Ore) Soit G = (X,A) un graphe simple d'ordre $n \geq 3$. Si pour toute paire de sommets $\{x, y\}$ non adjacents, on a $d(x) + d(y) > |X|$ alors G est hamiltonien.

Corollaire (Dirac) Soit G = (X,A) un graphe simple d'ordre $n \geq 3$ Si pour tout sommet x de G on a $d(x) \geq \frac{n}{2}$ alors G est hamiltonien.

Exemple : Lesquels des graphes suivants est hamiltonien ?

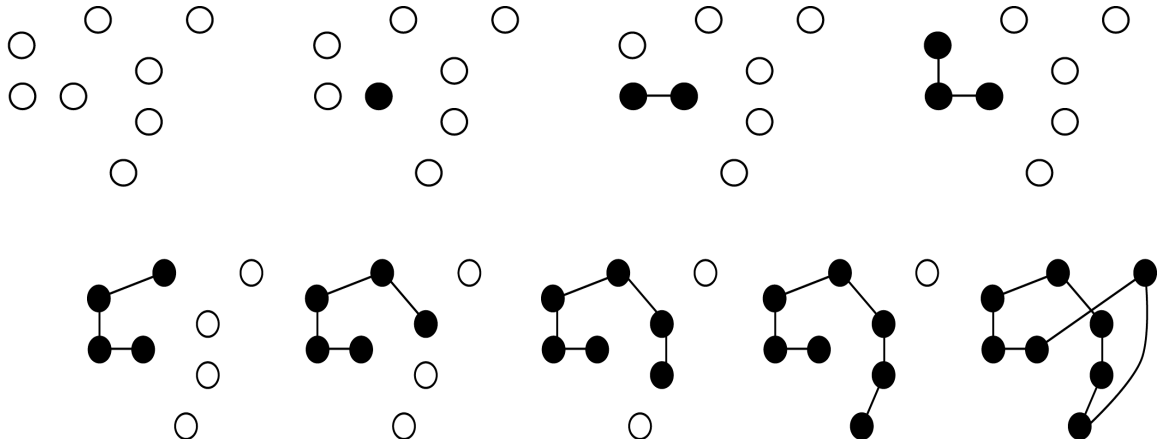


Lorsqu'on a affaire à des gros graphes ou qu'on désire obtenir des solutions très rapidement, on utilise ce qu'on appelle des *heuristiques* qui fournissent des solutions de
qualité raisonnable en des temps raisonnables.

Heuristique du plus proche voisin

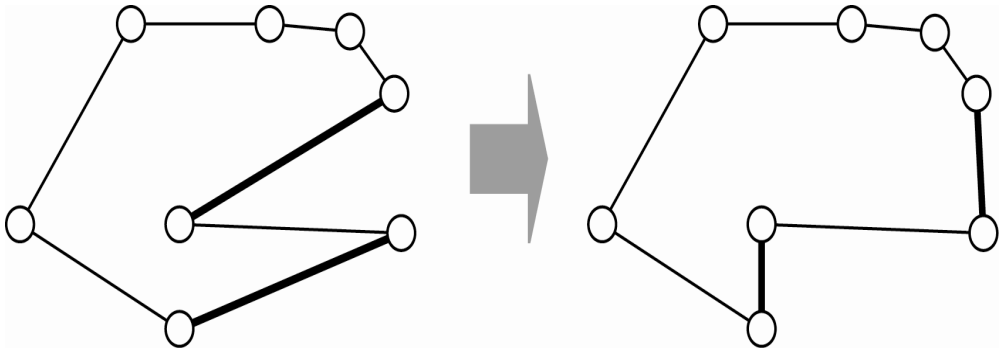
1) Choisir un sommet de départ x
2) Tant que tous les sommets ne sont pas encore visités faire l'opération suivante
se rendre au sommet le plus proche pas encore visité

Exemple



Toute heuristique, y compris celle du plus proche voisin, peut facilement être améliorée en rajoutant à la fin une procédure de *post-optimisation*. Celle-ci consiste à vérifier s'il existe une paire d'arêtes sur le cycle qui peut être remplacée par une autre paire d'arêtes (il n'y a qu'un remplacement possible par paire d'arêtes) tout en diminuant la distance totale parcourue. Tant que de telles paires d'arêtes existent, les échanges sont effectués.

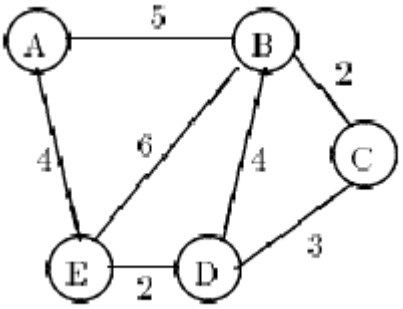
Illustration



Cette procédure de post-optimisation permet de réduire considérablement l'écart à l'optimum. Par exemple, on a pu constater que la combinaison de l'heuristique du plus proche voisin suivie d'une post-optimisation a amélioré la performance de l'algorithme en moyenne d'environ 2 à 3%. Il existe des méthodes qui permettent de réduire cet écart à quelques dixièmes de pourcent, mais ces méthodes sont beaucoup plus complexes à mettre en place.

Application 2: Arbres couvrant de poids minimum

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



Question : Comment câbler le réseau à moindre coût ?

Réponse : Il s'agit d'enlever des arêtes au graphe de façon qu'il reste connexe, et que la somme des pondérations des arêtes soit la plus petite possible. Remarquons que le graphe partiel recherché est sans cycle (pourquoi ?)

Ce problème se pose, par exemple, lorsqu'on désire relier n villes par un réseau routier de coût minimum. Les sommets du graphe représentent les villes, les arêtes, les tronçons qu'il est possible de construire et les poids des arêtes correspondent aux coûts de construction du tronçon correspondant.

Le problème est ramené au problème suivant :

Définition : Le problème de l'arbre recouvrant de poids minimal est celui qui consiste à déterminer un arbre qui soit un graphe partiel d'un graphe G simple connexe et dont le poids total est minimal

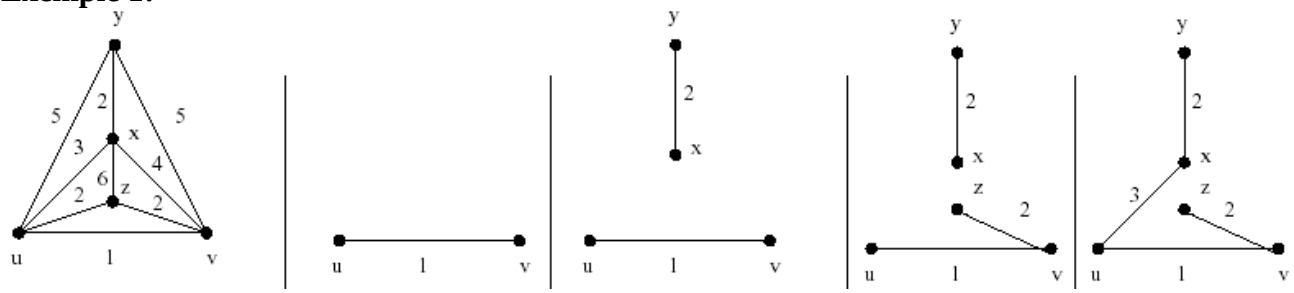
Exemple de solutions sur le graphe ci-dessus

Algorithme de Kruskal

La stratégie de cet algorithme consiste à construire l'arbre en question comme suit : on part d'une solution vide. On choisit, à chaque fois, une arête de G de poids minimum et qui ne crée pas de cycle. Soit E l'ensemble des sommets de G. On utilisera un ensemble d'arêtes T qui sera en sortie l'arbre couvrant minimal et un ensemble d'arêtes F qui représentera les arêtes qui peuvent être choisies.

```
T  = { };
F =  E ;
tant que |T| < n - 1 faire
   trouver une arête e de F de poids minimal
   F = F - e
   si T + e est acyclique
      alors T = T + e
   finsi
fin tant que
```

Exemple 1:



Les différentes étape pour construire l'arbre T

Exemple 2 : Reprenons le graphe ci-dessus, et construisons l'arbre recouvrant de poids minimum à l'aide de l'algorithme de Kruskal.

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

Preuve d'optimalité: Soit n l'ordre du graphe. L'algorithme produit bien un arbre recouvrant du graphe puisqu'il termine lorsque les $n-1$ arêtes sont choisies et T est acyclique. Supposons maintenant que l'arbre recouvrant T ne soit pas minimal. Si $e_1, e_2, ..., e_{n-1}$ alors $p(e_1) \leq p(e_2) \leq ... p(e_{n-1})$ par construction ($p(e_i)$ étant le poids de l'arête e_i). Soit alors A un arbre couvrant minimal tel que l'indice de la première arête de T, qui ne soit pas une arête de A, soit maximum. Soit donc k cet indice. On a alors $e_1, e_2, ..., e_k \in T$, $e_1, e_2, ..., e_{k-1} \in A$ et $e_k \notin A$. Alors $A + e_k$ contient un unique cycle C. C ne peut pas être constitué uniquement d'arêtes de T (hormis e_k) car sinon T contiendrait un cycle. Soit alors e' une arête de C appartenant à A mais non à T. Nous avons alors $A + e_k - e'$ est donc un arbre

couvrant du graphe, et $p(A+e_k-e') = p(A)+p(e_k)-p(e')$. Dans l'exécution de l'algorithme de Kruskal, l'arête e_k a été choisie de poids minimal tel que le graphe contenant le sous graphe contenant les arêtes $e_1, e_2, ..., e_k$ soit acyclique. Mais le sous graphe contenant les arêtes $e_1, e_2, ..., e_{k-1}, e'$ est aussi acyclique, puisqu'il est sou graphe de A. Par conséquent, $p(e') \geq p(e_k)$ et $p(A+e_k-e') \leq p(A)$. On en déduit que $A+e_k-e'$ est optimal et diffère de T par une arête strictement supérieur à k : contradiction.

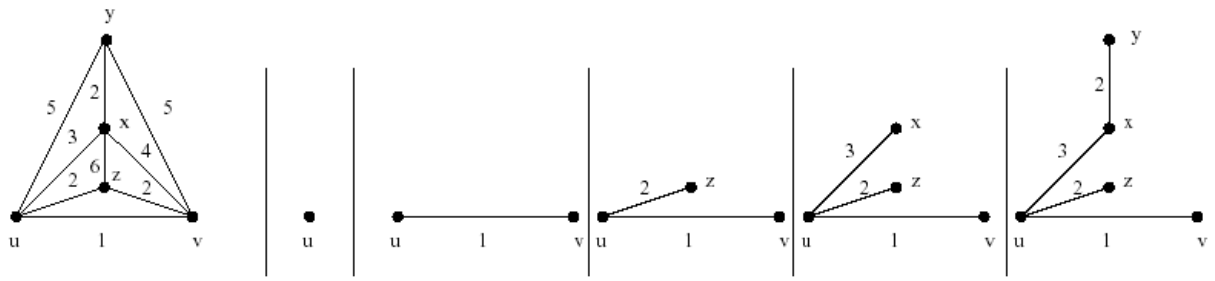
Complexité : Si les arêtes sont classées par ordre de coût croissant, ce qui peut se faire en $O(e\log e)$ **;** e étant le nombre d'arêtes dans le graphe, il reste à évaluer la complexité de test d'acyclicité. Si les composantes connexes du graphe T sont connues, ce test se réduit à vérifier que les extrémités de l'arête choisie sont dans eux composantes connexes distinctes. Il reste alors à gérer ces composantes connexes de manière à fusionner deux composantes connexes reliées par l'arête choisie. Cela peut se faire par la technique de fusion rapide en $O(e\log n)$ **- comment? –** L'algorithme de Kruskal a une complexité de l'ordre $O(e\times\max(\log n, \log e))$

Algorithme de Prim

L'algorithme de Kruskal veille à maintenir la propriété d'acyclicité d'un arbre alors que l'algorithme de Prim se base sur la connexité d'un arbre. L'algorithme de Prim fait pousser un arbre couvrant minimal en ajoutant au sous-arbre T déjà construit une nouvelle branche parmi les arêtes de poids minimal joignant un sommet de T à un sommet qui n'est pas dans ce dernier. L'algorithme s'arrête lorsque tus les sommets du graphe sont dans T.

Soit X l'ensemble de sommets du graphe de départ G. On utilisera un ensemble d'arêtes qui sera en sortie l'arbre recouvrant en question, et S un ensemble qui contiendra les sommets de T.

T $= \emptyset$
S= S $\cup\, x$
tant que S $\neq$ X faire
  trouver une arête e = {y,s} de poids minimal tel que y$\in$ X-S et s$\in$ S
  T = T $\cup$ {y,s}
  S = S $\cup$ y
findutantque



Les différentes étape pour construire l'arbre T

Preuve de l'algorithme : Soit *n* l'ordre du graphe G. L'algorithme produit bien un arbre couvrant du graphe puisqu'il termine lorsque tous les sommets de X sont dans S, et à chaque itération, une arête est choisie. Donc T contient les sommets de G et a exactement $n-1$ arêtes.

Supposons maintenant que l'arbre couvrant T ne soit pas minimal. Soit A un arbre couvrant minimal qui a le maximum d'arêtes en commun avec T. Si $e_1, e_2, ..., e_{n-1}$ sont les arêtes de T dans l'ordre de choix de l'algorithme, alors $p(T) = \sum_{i=1}^{n-1} p(e_i)$. Soit $j = \inf\{i, 1 \le i \le n, e_i \notin A\}$ (e_j est la première arête de T qui ne soit pas dans A). On notera S_i l'ensemble des sommets du sous-graphe induit par les arêtes $e_1, e_2, ..., e_i$. Le graphe $A + e_j$ n'est pas un arbre donc contient un unique cycle C. Or, par construction, e_j relie nécessairement un sommet de S_{j-1} à un sommet de X-S_{j-1}. Donc C doit contenir une arête $e \neq e_j$ qui elle aussi relie un sommet de S_{j-1} à un sommet de X-S_{j-1}. Le graphe $A + e_j - e$ est alors un arbre couvrant du graphe de départ G. Mais par construction de T, $p(e_j) \le p(e)$, d'où $p(A + e_j - e) \le p(A) + p(e_j) - p(e)$: $A + e_j - e$ est alors un arbre couvrant minimal qui a plus d'arêtes en commun avec T ce qui contredit l'hypothèse sur A.

Complexité : Si S possède *k* sommets, pour choisir une arête de poids minimal reliant un sommet de S à un sommet de X-S, on doit trouver le poids minimum parmi au plus $k(n-k)$ arêtes puisque chaque sommet de S est adjacent à au plus $(n-k)$ sommets de X-S. Or *k* varie entre 1 et *n*, donc $k(n-k) < (n-1)^2$. Ce choix est effectuée *n* fois dans la boule tant que. Donc la complexité est en $O(n^3)$.

Exercice : Proposez une structure de données pour le graphe G de telle manière que la complexité temporelle de l'algorithme de Prim soit en $O(e \log n)$.

Références : d'autres solutions de meilleures complexités existent dans la littérature. Ainsi, ce problème peut résolu en un temps linéaire si les poids sont de petits entiers (Freedman and Willard : trans-dichotomous algorithms for minimum spanning trees and shorets paths, 31st IEEE Symp. Fournadations of Compu. Sci., 1990, pp. 719-725).

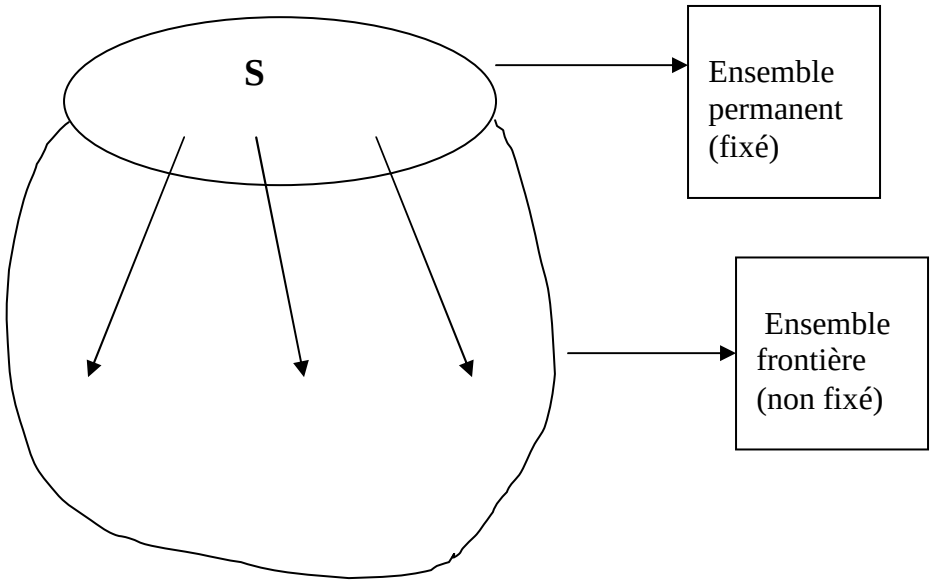
Autrement, la meilleure solution existant dans la littérature est proche de linéaire. La borne exacte est $O(e \log \beta(e,n))$ où la fonction $\beta(e,n)$ est le plus petit indice *i* tel que $\log(\log(\log(...\log(n)...)))$ plus petit que $\frac{e}{n}$ (Gabow, Galil, Spencer et Tarjan : efficient algorithms for finding minimu spanning trees in undirected and directed grahs, Combinatorica, Vol. 6, 1986, 109-122.

Application 3: chemin le plus court

Soit un graphe (orienté ou non) pondéré, c'est-à-dire chaque arc de ce graphe est muni d'un poids (nombre réel). Le poids d'un chemin est défini comme étant la somme des poids des arcs qui le constituent.

Le problème du plus court chemin, dans ce chapitre, consiste à déterminer le poids minimal d'un chemin d'un sommet à tous les autres, en supposant des poids positifs.

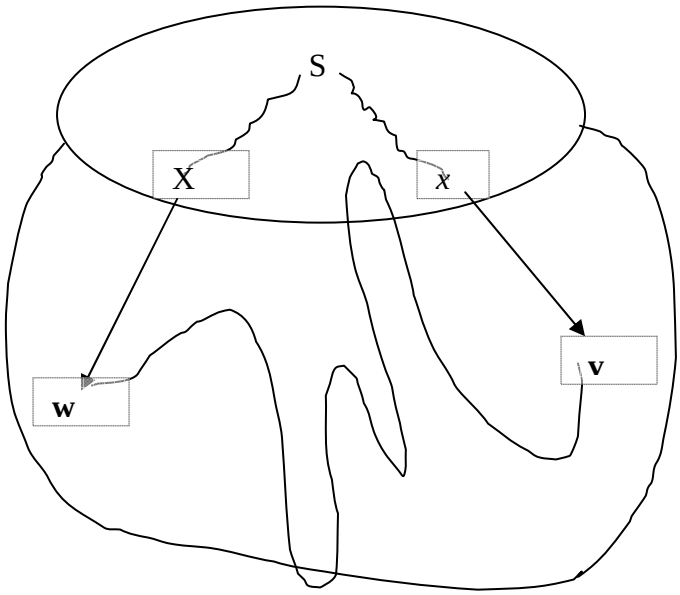
L'algorithme présenté ci-dessous est celui de Dijkstra. La stratégie de cet algorithme est comme suit : On construit un ensemble fixé (permanent) de sommets *x* ayant le plus court chemin T_x. Initialement, cet ensemble contient juste le sommet de départ S, et son plus court chemin est $T_s = 0$. À chaque étape, on considère l'ensemble frontière des sommets qui ne sont pas encore fixés et sont adjacents à ceux qui sont permanents. Si le sommet *y* qui est dans frontière est adjacent à $x_1, x_2, ..., x_k$ appartenant à l'ensemble permanent, alors nous posons temporairement T_y égal au minimum de (T_{x_1} + poids de (x_1, y),(T_{x_2} + poids de (x_2, y), …,(T_{x_k} + poids de (x_k, y)), comme illustré par la figure ci-dessous. Parmi les sommets adjacents à cet ensemble permanent, nous prenons *w* dont T_w est minimale. Il est alors mis dans l'ensemble fixé et supprimé de l'ensemble frontière. Cette procédure est répétée jusqu'à ce que l'ensemble frontière devient vide.



Théorème : Si un sommet *w* dans frontière possède le plus petit chemin temporaires T_w parmi les sommets de cet ensemble, alors ce poids devient permanent.

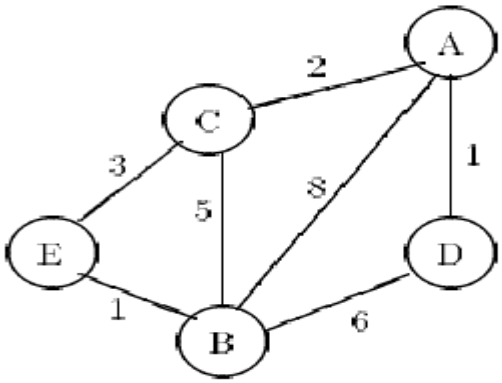
Preuve: La preuve est par contradiction. Supposons que T_w ne peut pas être mis comme étant le plus petit chemin pour le sommet *w*. Alors, il doit exister un autre chemin de poids plus petit du sommet S au sommet *w* (voir figure ci-dessous).

Ce nouveau chemin contient le plus petit chemin de S à *x*, ensuite, ça continue vers *v*. Ensuite, une autre suite de sommets (représenté dans la figure par des lignes courbées), avant d'arriver au sommet *w*. Autrement dit, T_v + poids du chemin en ligne courbée $< T_w$. Mais cette inégalité ne peut pas être vraie pour la simple raison que le poids du chemin en ligne courbée est positive. D'où le résultat du théorème.




```
POUR tout sommet t
    d(s; t) := +infini
FINPOUR;
T[s] = 0;
TANT QU'il reste des sommets non fixés
     choisir un sommet w non fixé tel que T[t] soit minimal;
     supprimer  w des sommets non fixés
     POUR tout (w,x) dans E
          SI T[x] > T[w] + poids (w,x)
              ALORS
                T[x] = T[w] + poids (w,x)
                Si  x n'est pas membre de frontiere
                     Alors ajouter x à frontiere
         FIN SI
     FIN POUR
FIN TANT QUE
```

Exemple : Soit à chercher les plus courts chemins depuis le sommet A vers tous les autre sommets dans le graphe suivant :



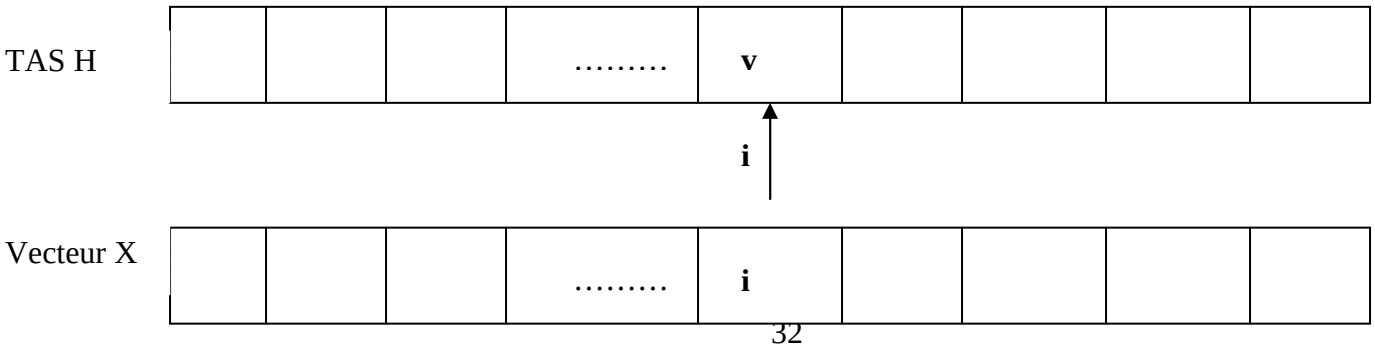
Le tableau suivant donne les distances depuis A et les arcs marqués au cours des différentes étapes de l'algorithme :

Parmanent	**frontiere**	**poids temporaire du chemin**	**poids fixe du chemin**
{}	{A}	Ta = 0	Ta =0
{A}	{B,C,D}	Tb =8 ;Tc=2 ;Td=1	Td=1
{A,D}	{B,C}	Tb = 7, Tc = 2	Tc =2
{A,D,C}	{E,B}	Te = 5, Tb =7	Te = 5
{A,D,C,E}	{B}	Tb = 6	Tb =6
{A,D,C,E,B}	{}		

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

Exercice : comment changer l'algorithme pour qu'il donne le plus court chemin à un sommet donné d'une manière explicite.

Complexité et structure de données: Nous avons besoin d'une structure de données pour supporter l'ensemble frontiere. Nous pouvons le représenter comme un TAS H et un vecteur d'indices X dans le TAS. Utilisant ce vecteur, étant donné un sommet, sa location dans le TAS H est déterminé en O(1).



X[v] = 0 si le sommet **v** n'est pas dans le TAS. Si maintenant le sommet v est stocké dans la location H[i], alors X[v] prendra la valeur *i*. Les opérations sur l'ensemble frontiere peuvent être implantées comme suit :

v

- Ajouter (x, frontiere) : utiliser T comme un TAS et insérer x . À chaque fois qu'un élément de T est mis à jour (T[i] =j) mettre à jour aussi le vecteur X (X[j] = i).

- Supprimer minimum (x, frontiere) : utiliser T comme un TAS et y supprimer x. De plus, à chaque fois qu'une location du TAS est effacée par un indice j, mettre à jour le vecteur X en faisant X[j] =i. et aussi, comme x est supprimé de l'ensemble frontiere, son indice doit aussi être supprimé (X[x] = 0).

- Modification de poids (x,frontiere): le nouveau poids de x est donné par T[x]. Par ailleurs, T doit être maintenu comme un TAS en le remontant à sa bonne location vers la racine. Chaque fois qu'un élément de T est mise à jour (T[i] =j), on doit aussi mettre à jour aussi le vecteur X (X[j] = i).

La complexité de ces opérations est comme suit :

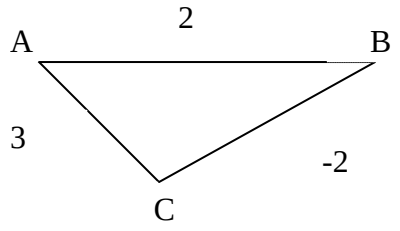
Ajouter(x,frontiere) :	$O(\log n)$
Supprimer-minimum(x,frontiere) :	$O(\log n)$
Membre(x,frontiere) :	$O(1)$
Modification de poids(x, frontiere) :	$O(\log n)$

Par conséquent, la complexité de l'algorithme de Dijkstra est comme suit :

$$O(n + \quad n\log n + \quad e + \quad \max(n\log n, e\log n))$$

initialisation **supprimer_min** **Membre** **ajouter frontiere** **modifier poids**

L'algorithme de Dijkstra et les distances négatives : Cet algorithme ne génère pas la bonne solution si les poids des arêtes sont négatifs. Par exemple, considérons le graphe ci-dessous :



L’algorithme ne va jamais trouver que le plus court chemin allant de A vers B est 1(passant par C).

Application 4 : chemin le plus long

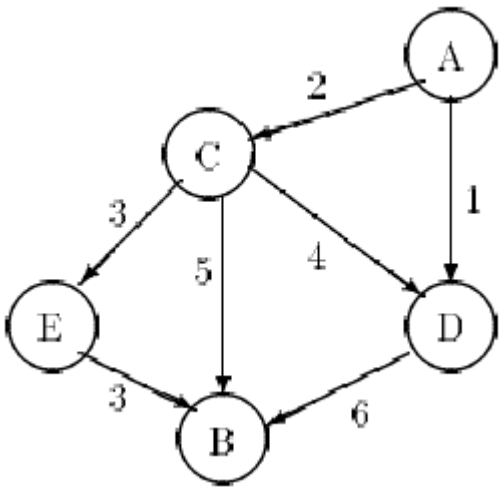
Le problème à résoudre est à peu prés l’opposé du problème précédent. Deux cas peuvent se présenter :

Cas 1 : le graphe est acyclique

Dans le cas particulier d'un graphe acyclique, on peut adapter efficacement les algorithmes de plus court chemin à la recherche des poids maximaux. La modification de l'algorithme de Dijkstra est comme suit (algorithme de Bellmann) :

```
POUR tout sommet t
   T[t] = +infini
FINPOUR;
T[s] = 0;
Fixer s;
TANT QU'il reste des sommets non fixés
  choisir un sommet t non fixé dont tous les prédécesseurs sont fixés;
  POUR tout successeur u de t
     SI T[u] > T[t] + poids de (t,u) ALORS
        T[u] = T[t] + poids de (t,u)
    FIN SI
 FIN POUR;
 Fixer t
FIN TANT QUE
```

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



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

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

<u>Cas 2</u> : **<u>le graphe contient des cycles</u>**

L'idée est de parcourir l'arbre des chemins acycliques partant d'un sommet s, au moyen d'un algorithme de parcours en profondeur utilisant une liste :

```
parcourir le graphe :
    POUR tout sommet t différent de s
       T[t] := +infini
    FINPOUR;
   T[s] = 0;
   visiter s ;

visiter t :
 ajouter t à la liste;
 POUR tout successeur (sommet adjacent) u de t qui n'est pas dans la liste
    SI T[u] = +infini  ou T[u] < T[t] + poids de (t, u) ALORS
       T[u] := T[t] + poids de (t,u)
    FIN SI;
  visiter u
FIN POUR;
enlever t de la liste
```

Optimalité de l'algorithme: on montre récursivement que lorsqu'on visite un sommet t qui n'est pas dans la liste, la liste comporte la suite des sommets d'un chemin acyclique allant de s à t; d'autre part, les états de la liste sont distincts à chacune des entrées dans la procédure visiter. Par conséquent l'algorithme termine (il ne peut y avoir qu'un nombre fini d'états de la liste, correspondant aux chemins acycliques depuis le sommet s). Pour tout sommet t, à tout instant, si T[t] < +infini, T[t] représente le poids d'un chemin acyclique allant de s à t. Comme tous les chemins sont examinés, on obtiendra bien le poids maximal.

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

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

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

Un mot sur la représentation matricielle d'un graphe

Dans certaines situations, nous pourrions être intéressés par le calcul du nombre de chemins ou de cycles d'une certaine longueur. La structure de données de la matrice adjacente peut être directement utilisée pou répondre à cette interrogation.

Théorème
Soit $G = (X,A)$ un graphe orienté, avec $X = \{x_1, x_2 ,\ldots,x_n \}$; de matrice d'adjacence $M = (m_{i,j})$. Pour tout entier naturel k, non nul notons M^k la $k$ème puissance de la matrice. Alors m^k_{ij} est égal au nombre de chemins de longueur k du sommet x_i au sommet x_j .

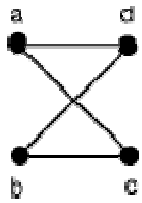
Démonstration
Effectuons une récurrence sur k : $m^k_{ij} = m_{i,j}$ désigne bien le nombre de chemins allant de x_i à x_j.
Supposons le résultat vrai pour l'entier $k - 1$; comme $M^k = M^{k-1} \times M$, on a par définition :

$$m_{ij}^{\ k} = \sum_{l=1}^{n} m_{il}^{\ k-1} \times m_{lj}$$

Par hypothèse de récurrence, $m_{il}^{\ k-1}$ est le nombre de chemins de longueur $k - 1$ allant de x_i à x_l et m_{lj} est égal à 1 si (x_l, x_j) est une arête de G et à 0 sinon. Observons que $m_{il}^{\ k-1} \times m_{lj}$ est le nombre de chemins de longueur k allant de x_i à x_j dont la dernière arête est (x_l, x_j), la somme de ces termes est donc bien le nombre de chemins de longueur k allant de x_i à x_j

Exemple

- Déterminons le nombre de chemins de longueur 4 allant de *a* à *b* dans le graphe G_1



G_1

La matrice d'adjacence de G_1 est : $M = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$

Le nombre de chemins cherché est le terme (1, 2) de la matrice $M^4 = \begin{pmatrix} 8 & 8 & 0 & 0 \\ 8 & 8 & 0 & 0 \\ 0 & 0 & 8 & 8 \\ 0 & 0 & 8 & 8 \end{pmatrix}$ c'est-à-dire 8.

Techniques générales
de résolution de problèmes

Introduction: La question fondamentale que l'on se pose est le suivante : comment approcher un problème pour le résoudre. Il n'existe pas de recette pour résoudre un problème. Néanmoins, certaines techniques générales sont à notre disposition pour nous accompagner dans ce processus de résolution.

Ce dernier chapitre n'a pas pour but de détailler toutes ces techniques; cela est l'objet d'autres cours. Toutefois, il est utile d'avoir une vue d'ensemble sur ces techniques. Non seulement pour avoir un avant goût sur les différentes stratégies d'approche de résolution, mais aussi le choix d'une stratégie peut influer sur le choix d'une structure de données.

Après une brève descriptions de quelques techniques, nous allons nous nous arrêter un peu plus sur deux méthodes à savoir la méthode de branch and bound et la méthode heuristique.

Les méthodes de résolution

Elles peuvent être répertoriées en quatre catégories :

1. **Concevoir des solutions en étant vorace (glouton):** Cette approche consiste, à partir d'un solution vide, à construire une solution à partir de règles simples qui, souvent obéissent au bon sens. Le plus simple exemple que nous puissions donner est celui de l'algorithme de Kruskal. Comme nous l'avons vu, à chaque itération, cet algorithme « expand » sa solution partielle courante en choisissant l'arête de poids minimal qui ne crée pas de cycle. Dans certaine situations, être glouton localement peut conduire à une solution optimale globalement.

2. **Concevoir des solutions en divisant et régnant :** Cette approche consiste à subdiviser le problème de départ en plusieurs sous-problèmes de tailles plus petites. D'une manière récursive, ce processus va être continué en des sous problèmes de taille de plus en plus petite jusqu'à arriver à des problème de taille dont la résolution est plutôt facile. Ensuite, on remonte cette récursivité en combinant les solutions de ces sous-problèmes en des solutions des problèmes qui les a subdivisé. Ainsi de suite, jusqu'à arriver à la solution du problème de départ. Pour obtenir des solutions de meilleures complexités, il est conseillé d'avoir des sous-problèmes de tailles identiques ou presque.
   L'exemple que nous pouvons mentionner est celui du tri par fusion. Cet algorithme consiste à diviser le tableau en deux partie égales. D'une manière récursive, ces deux parties sont ensuite divisées de la même manière jusqu'à avoir deux tableau d'un élément chacun. Ensuite, on remonte cette récursivité en fusionnant deux parties d'un tableau triées en un seul tableau trié jusqu'à arriver au tableau de départ.

3. **Concevoir des solutions en faisant attention à la manière de diviser et régner *:*** Dans certaines situations, dans la méthode de diviser et régner, la subdivision de haut en bas, telle que mentionnée plus haut, est telle que les sous problèmes générés peuvent être résolus plusieurs fois. Cela engendre des solutions de mauvaise complexité. Afin d'éviter ces répétitions, une autre approche consiste à commencer à résoudre les problèmes de taille plus petite et ensuite d'aller à des tailles de plus en plus grande. Cette approche est connue sous le nom de programmation dynamique. Quand le nombre de sous-problèmes **différents** n'est pas très élevé,

cette approche génère des algorithmes de complexité efficace. Un bon exemple à mentionner est celui des nombres de Fibonacci. On a vu que l'implantation directe de cette fonction conduisait à une complexité exponentielle. La raison est que des sous-problèmes sont exécutés plusieurs fois (dessinez le schéma d'exécution des différents appels récursifs pour, par exemple, n =5) .

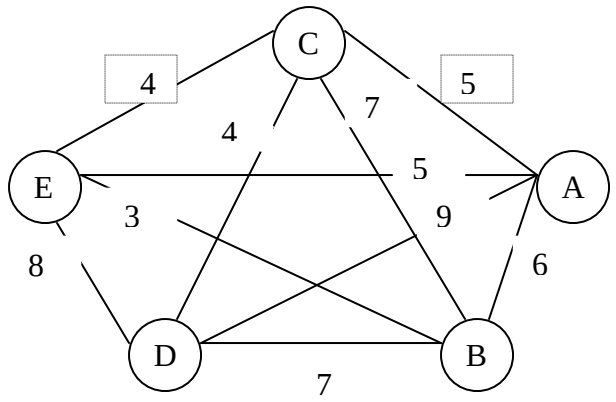
4. **Concevoir des solutions en faisant de l'énumération intelligente**: Dans certains situations, il est possible d'énumérer toutes les solutions du problèmes et prendre celle qui nous arrange. L'inconvénient majeur de cette approche est le nombre prohibitif du nombre de solutions. La méthode de *branch and bound* (*procédure par évaluation et séparation progressive*) consiste à énumérer ces solutions d'un manière intelligente en ce sens, en utilisant certaines propiétés du problème en question, elle arrive à éliminer des solutions partielles qui ne mènent pas à la solution que l'on recherche. Bien entendu, dans le pire cas, on retombe toujours sur l'élimination explicite de toutes les solutions du problème.

Pour ce faire, cette méthode se dote d'une fonction qui permet de mettre une borne sur certaine solutions pour les exclure de les maintenir comme des solutions potentielles. Bien entendu, la performance d'une méthode de branch and bound depend, entre autres, de la qualité de cette fonction (de sa capacité d'exclure des solutions partielles tôt). Illustrons ce point, prenons l'exemple du voyageur de commerce.

Définition : Soit un graphe G=(V,E). Un cycle est **hamiltonien** si et seulement si tous les sommets de G apparaissent une et seule fois dans ce cycle.

Définition : Soit un graphe G=(V,E) valué. Le problème du voyageur de commerce consiste à trouver un cycle hamiltonien dont la somme des poids est minimale.

Soit donc le graphe de la figure ci-dessous :



Nous pouvons résoudre ce problème en commençant par exemple à partir du sommet **E**.

Soit la borne v pour ce sommet (on verra plus loin comment la trouver !) c'est-à-dire la valeur de toutes les solutions incluant le sommet E vont avoir un coût $\geq v$. Le prochain sommet du cycle que nous recherchons est soit A, B, C ou D. Pour chacune de ces solutions partielles, nous calculons une nouvelle borne (signifiant que toutes les solutions comprenant cette solution partielle va avoir un coût $\geq$ à cette nouvelle borne).

Initialement, nous pouvons mettre le coût de la meilleure solution trouvée à un très grand nombre. L'idée de cette méthode la suivante : toute solution partielle dont le coût est plus grand que celui de la meilleure solution trouvée jusqu'à`présent va être exclue de la recherche. Ce processus est continué jusqu'à avoir une solution complète. Si le coût de cette solution complète est inférieure à celui de la meilleure solution que nous avons, alors nous remplaçons ce coût comme étant celui de la meilleure solution courante.

Il existe plusieurs manières de décider quelle solution partielle à explorer en premier. Celle que nous allons décrire ci-dessous est celle qui consiste à choisir toujours la solution ayant la plus petite borne. Mais il existe d'autres manières de procéder. Les unes se valent que les autres.

La partie cruciale de branch and bound est la qualité de la fonction F. Si elle est trop simple, probablement que l'exclusion des solutions partielles se fera à des moment avancés. Cela a pour effet de rallonger encore les temps d'exécution des ces algorithmes.

Une fonction F peut être comme suit :

Soit le cycle hamiltonien suivant : $v_1, v_{2,}, ..., v_n, v_{n+1} = v_1$. Il est facile de voir que, quelque soit la solution, son coût est $\geq \frac{1}{2}\sum_{i=1}^{n}(arc - v_{i_1} + arc - v_{i_2})$ où $arc - v_{i_1}$ et $arc - v_{i_2}$ désignent deux arcs adjacents au sommet *i* ayant **<u>le plus petit poids</u>**.

En commençant à partir de E, on aura :

Le coût est ≥ ½{(3+4)+(4+4)+(5+5)+(3+6)+(4+7)}= 22.5

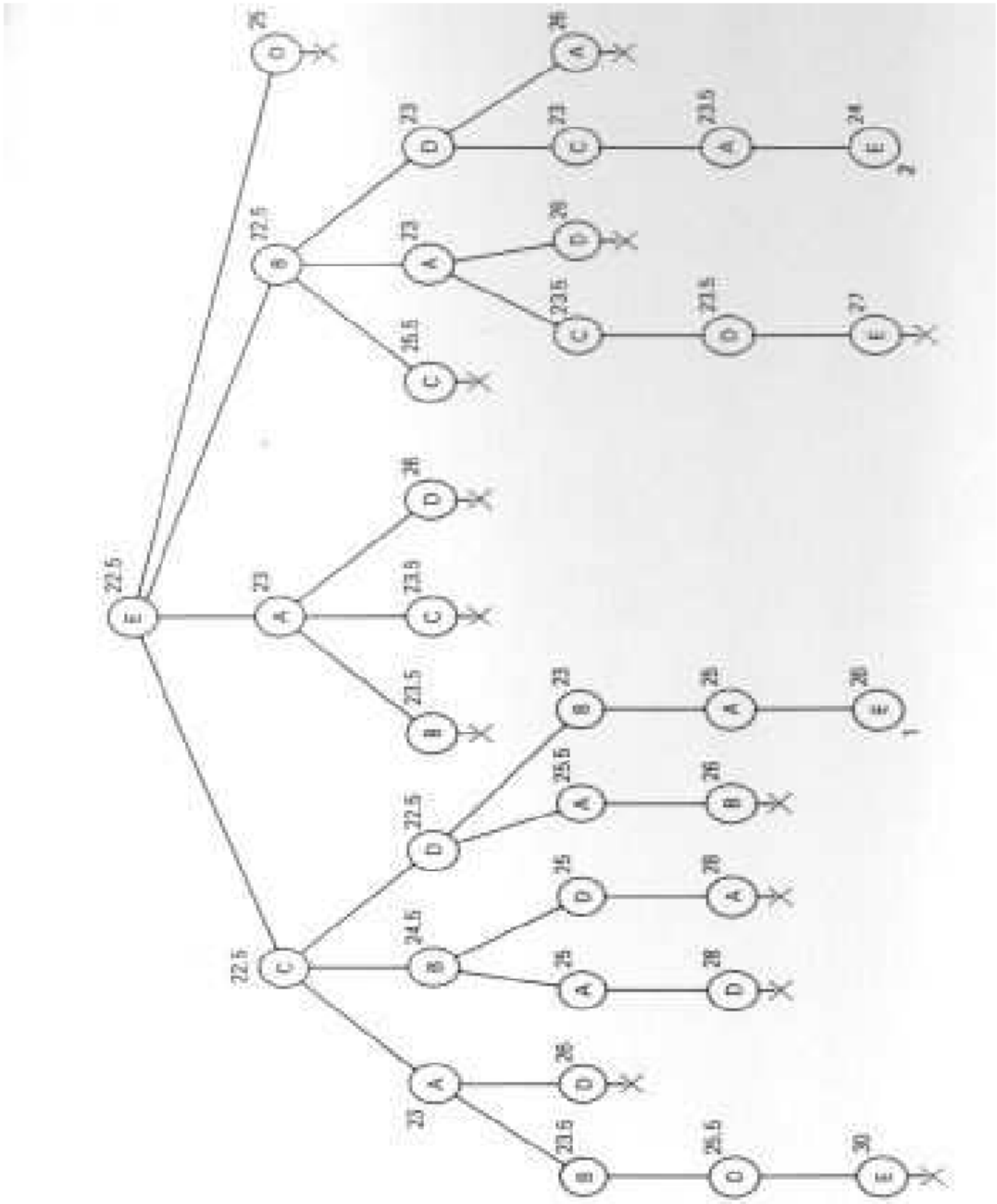
Les prochain sommets dans le cycle peuvent être : A, B, C ou D. Pour chacune des ces solutions partielles, une borne est calculée. Par exemple pour D, la nouvelle borne est :

½{(8+3)+(4+4)+(5+5)+(3+6)+(4+8)}= 25.

Ceci est fait pour chacun des sommets. On obtient successivement : 22.5 pour le sommet **C,** 23 pour le sommet **A** et 22.5 pour le sommet **B**. Parmi ces quatre valeurs, nous allons explorer le sommet ayant la plus petite valeur, soit le sommet (remarquez qu'il existe d'autre manières de procéder dans le choix de la prochain solution partielle à explorer; chacune a ses propres avantages et inconvénients. À titre d'exemple, nous pouvons citer DFS et BFS.

Ce processus est ensuite répété. Quand une solution complète est trouvée, la meilleure solution courante est modifiée si cela est nécessaire. Par exemple, dans notre exemple, la première solution trouvée a pour coût 26 remplaçant ainsi l'ancienne valeur que nous avons initialisée à un grand nombre. L'idée de Branch and Bound est résumée dans ce qui suit : Quand un sommet a une borne plus grande que cette valeur, il n'y a plus lieu de le considérer étant donnée que cette solution partielle ne mènera pas à la solution minimale.

Ce processus de génération de solutions partielles, avec le calcul de leur borne, génère une arborescence. Pour exemple ci-dessus, l'arborescence obtenue est la suivante :


22.5
E
22.5
C
23
A
23.6
B
26
D
24.5
B
25
A
26
D
22.5
D
25.5
A
23
B
26
A
26
B
26
E
23
A
26
D
23.6
B
25.5
D
30
E
23
A
23.5
C
21.5
B
26
D
72.5
B
26.5
C
23
D
23
A
23.6
C
26
D
23
C
23.5
A
26
A
26
C

Exercice : proposez une autre fonction F et la comparer à celle présentée ci-dessus.

Exercice : discuter la structure de données la mieux appropriée pour implanter cette méthode, dans le contexte du problème de voyageur de commerce. Quels problèmes, selon vous, cette méthode risque de poser lors de son exécution ?

5. **Concevoir des solutions en étant moins exigeant :** Souvent on est en face d'un dilemme pour résoudre un problème:

   1. obtenir la solution optimale mais, souvent, à un prix extrêmement élevé en temps de calcul, comme c'est le cas de la méthode de branch & bound.
   2. ou alors obtenir une solution à un prix raisonnable (en temps de calcul et espace). Seulement, souvent, cette solution produit des solutions qui ne sont pas optimales (solutions approchées). D'où le nom d'approche heuristique.

Une fois le choix de l'approche heuristique accepté (pour certains problèmes, ce choix est imposé), l'étape suivante est de savoir comment en concevoir une.

Il existe plusieurs approches heuristiques. Les plus utilisées sont :

1. méthodes voraces (gloutonnes)
2. méthodes itératives
3. méthodes lagrangiennes

Celle qui nous intéresse dans ce chapitre est la première citée.

Méthode vorace (gloutonne)

Cette approche nous l'avons déjà présentée précédemment. Comme cela a été déjà mentionné, dans certains cas (l'algorithme de l'arbre de couvrement de poids minimum de Kruskal en est un bel exemple), cette approche produit une solution optimale. Cependant, cela n'est, en général, pas le cas, même si leur temps d'exécution est en général raisonnable.

Bien entendu, pour un problème donné, plusieurs heuristiques peuvent être conçues pour sa résolution. L'appréciation de la qualité de la solution générée par une heuristique donnée se fait de deux manières. Il est utile de mentionner que ces deux approches sont plutôt complémentaires.

1. **Expérimentation (simulation):** à travers une batterie de données, générées d'une manière aléatoire, ces différentes heuristiques sont exécutées. Les solutions générées sont soumises par la suite à une comparaison statistique, pour déterminer la meilleure d'entre-elles.
2. **Analyse mathématique** : cette approche consiste à calculer la distance qui sépare la valeur de la solution générée à celle de la solution optimale. Dans ce qui suit, nous allons voir juste le cas de l'évaluation de la distance maximale. Mais, il est clair que cette distance peut aussi être évaluée d'une manière probabiliste (c'est-à-dire en moyenne).

Exemple : Ordonnancement de tâches sur des machines parallèles

Soient *m* machines identiques mises en parallèle pour exécuter *n* tâches. Chaque tâche *i* doit passer un temps p_i sur une seule des machines pour être réalisée. Le problème consiste à répartir les *n* tâches sur les *m* machines de telle manière à les terminer le plus tôt possible. On suppose que l'ensemble des tâche et machines sont disponibles continuellement dès l'instant 0.

Solution : Une des solutions approchées utilisées consiste à mettre les *n* tâches dans une *liste*. Ensuite, dès qu'une machine devient libre, on lui affecte la première tâche de la liste.

Illustration:

Complexité : à discuter en classe !

Qualité de la solution : Une manière d'évaluer la qualité de la solution obtenue est de calculer l'écart maximal qui sépare le résultat obtenu par cette heuristique et la résultat obtenu par la solution optimale (qu'on ne connaît pas). En général, l'approche utilisé pour arriver à ce genre de résultats est l'utilisation de borne inférieures (supérieures pour les problèmes de maximisation).

En termes formels, si opt(I) est la valeur de la solution optimale et H(I) la valeur solution générée par l'heuristique H pour une instance quelconque I du problème, l'analyse dans le pire de l'heuristique H consiste à trouver deux valeurs ρ et *r* tels que:

$$H(I) \leq \rho \,\mathrm{opt(I)} + r$$

Références et sources

1. Anne Dicky : Graphes et algorithmes,
Conservatoire National des Arts et Métiers
Informatique - Cycle A - 1997/1998.

2. Joäelle Cohen, Théorie des graphes et algorithmes
MIAGE, 2002

3. Manoochehr Azmoodeh, abstract data types and algorithms, MacMillan, 1988.

4. **T. H. Cormen et al. (1990)**: Algorithms, MCGraw Hill.

5. Anany Levitin (2003): Introduction to the design and analysis of algorithms, Addison Wesley.

6. O. Carton (2003): Notes de cours d'algorithmique avancée en master de Bio-Info, Université Paris

7. **E. Sigward :** Introduction à la théorie des graphes, Université de Metz.