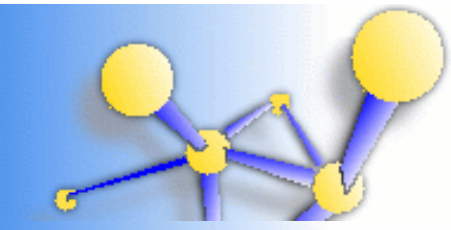
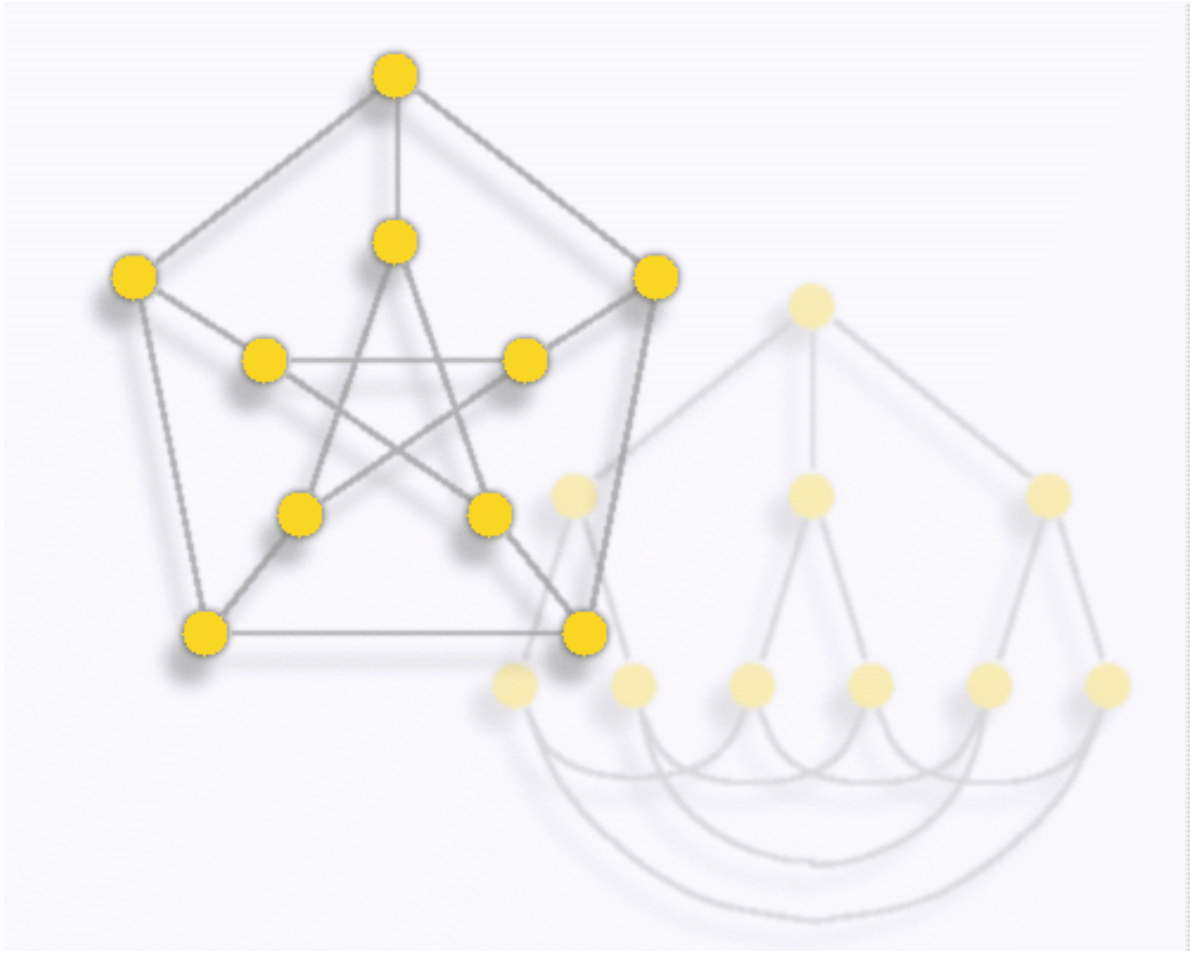


Théorie des Graphes



Les graphes modélisent de nombreuses situations concrêtes où interviennent des objets en interaction.

- Les interconnexions routière, ferrovière ou aériennes entre différentes agglomérations,
- Les liens entre les composants d'un circuit électronique,
- Le plan d'une ville et de ses rues en sens unique,...

Les graphes permettent de manipuler plus facilement des objets et leurs relations avec une représentation graphique naturelle. L'ensemble des techniques et outils mathématiques mis au point en Théorie des Graphes permettent de démontrer facilement des propriétés, d'en déduire des méthodes de résolution, des algorithmes, ...

- Quel est le plus court chemin (en distance ou en temps) pour se rendre d'une ville à une autre?
- Comment minimiser la longueur totale des connexions d'un circuit?
- Peut-on mettre une rue en sens unique sans rendre impossible la circulation en ville?

Un *graphe* permet de décrire un ensemble d'objets et leurs relations, c'est à dire les liens entre les objets.

- Les objets sont appelés les *noeuds*, ou encore les *sommets* du graphe.
- Un lien entre deux objets est appelé une *arête*

Définition

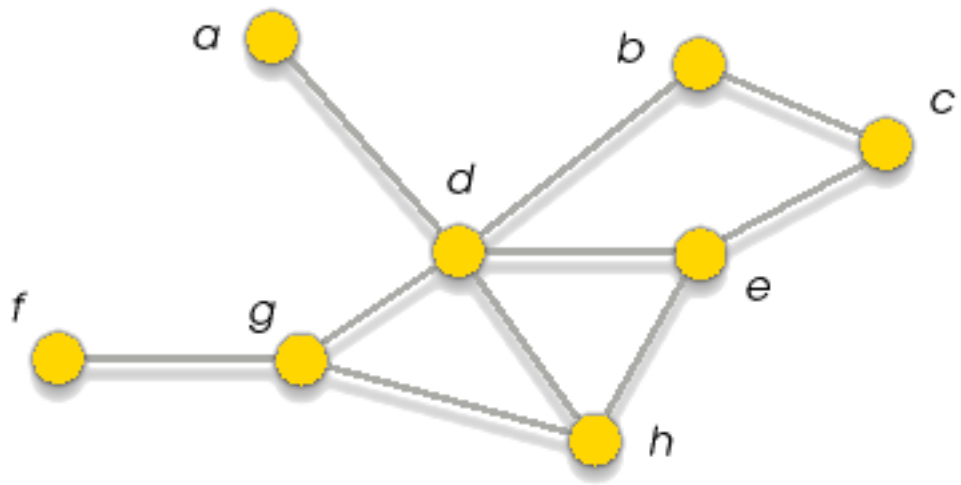
Un **graphe** ***G*** est un couple ***(V,E)*** où

- ***V*** est un ensemble (fini) d'objets. Les éléments de ***V*** sont appelés les **sommets** du graphe.
- ***E*** est sous-ensemble de ***V*x*V***. Les éléments de ***E*** sont appelés les **arêtes** du graphe.

Une arête ***e*** du graphe est une paire ***e=(x,y)*** de sommets. Les sommets ***x*** et ***y*** sont les **extrémités** de l'arête.

Exemple

Un exemple de graphe à 8 sommets, nommés *a* à *h*, comportant 10 arêtes.



G=(V,E)

V={ a, b, c, d, e, f, g, h }

E={ (a,d), (b,c), (b,d), (d,e), (e,c), (e,h), (h,d), (f,g), (d,g), (g,h) }

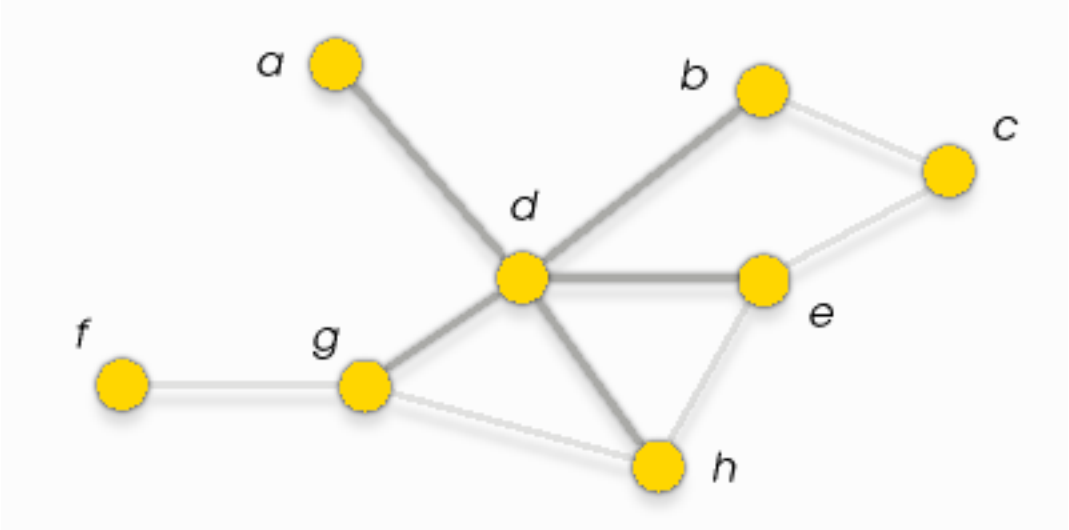
Notre définition d'un graphe correspond au cas des graphes *simples*, pour lesquels il existe au plus une arête liant deux sommets. Dans le cas contraire le graphe est dit multiple. Nous ne nous intéresserons ici qu'au cas des graphes simples sans boucle.

Les objets représentés par les sommets sont sans importance pour la manipulation du graphe. Nous dirons simplement qu'un graphe est d'*ordre* ***n*** si il comporte ***n*** sommets. Toute la richesse des graphes vient évidemment de la grande diversité que peut avoir l'ensemble de ses arêtes. Pour appréhender la structure d'un graphe, nous pouvons commencer par la caractériser *localement* en regardant pour chaque sommet les autres sommets auquels il est relié, le nombre d'arêtes dont il est extrémité,...

Définition **Degre**

- Deux sommets ***x*** et ***y*** sont **adjacents** si il existe l'arête ***(x,y)*** dans ***E***. Les sommets ***x*** et ***y*** sont alors dits **voisins**
- Une arête est **incidente** à un sommet ***x*** si ***x*** est l'une de ses extrémités.
- Le **degré** d'un sommet ***x*** de ***G*** est le nombre d'arêtes incidentes à ***x***. Il est noté **d*(x)***. Pour un graphe simple le degré de ***x*** correspond également au nombre de sommets adjacents à ***x***.

Exemple Dans le graphe le sommet ***d*** a un degré ***5***



d*(d)* = *5*

Les arêtes incidentes à ***d*** *sont :*
(d,a), (d,b) , (d,e), (d,h) *et* ***(d,g)***

Pour un graphe simple d'ordre ***n,*** le degré d'un sommet est un entier compris entre ***0*** et ***n-1***. Un sommet de degré ***0*** est dit *isolé* : il n'est relié à aucun autre sommet.
Citons ici deux propriétés très simples et souvent utiles.

Propriété

La somme des degrés des sommets d'un graphe est égal à 2 fois son nombre d'arêtes.

HINT
Une arête ***e=(x,y)*** du graphe est comptée exactement 2 fois dans la somme des degrés : une fois dans **d*(x)*** et une fois dans **d*(y)***

Propriété

Le nombre de sommets de degré impair d'un graphe est pair.

HINT
En écrivant la propriété sur la somme des degrés dans le corps ***Z/2Z*** (modulo 2), on obtient directement que le nombre de sommets de degré égal à 1 modulo 2 est nul.

Pour caractériser de manière moins locale la structure d'un graphe, il est possible de rechercher des parties remarquables du graphe, en restreingnant soit l'ensemble des sommets (sous-graphe), soit l'ensemble des arêtes (graphe partiel).

- Un *sous-graphe* de ***G*** consiste à considérer seulement une partie des sommets de ***V*** et les liens induits par ***E***. Par exemple si ***G*** représente les liaisons aériennes journalières entre les principales villes du monde, un sous-graphe possible est de se restraintre aux liaisons journalières entre les principales villes européennes.
- Un *graphe partiel* de ***G*** consiste à ne considérer qu'une partie des arêtes de ***E***. En reprenant le même exemple, un graphe partiel possible est de ne considérer que les liaisons journalières de moins de 3 heures entre les principales villes du monde.

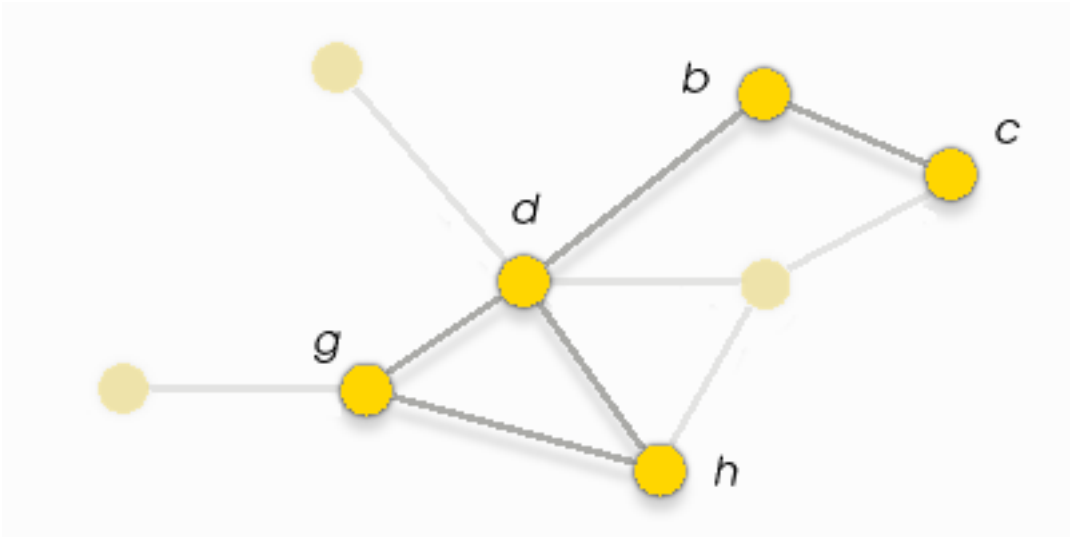
Définition

Pour un graphe ***G = (V,E)***

- Un **sous-graphe** de ***G*** est un graphe ***H=(W, E(W))*** tel que ***W*** est un sous-ensemble de ***V***, et ***E(W)*** sont les arêtes induites par ***E*** sur ***W***, c'est à dire les arêtes de ***E*** dont les 2 extrémité sont des sommets de ***W***. $E(W) = \{(x,y) \in E \mid x,y \in W\}$
- Un **graphe partiel** de ***G*** est un graphe ***I=(V,F)*** tel que ***F*** est un sous-ensemble de ***E***.

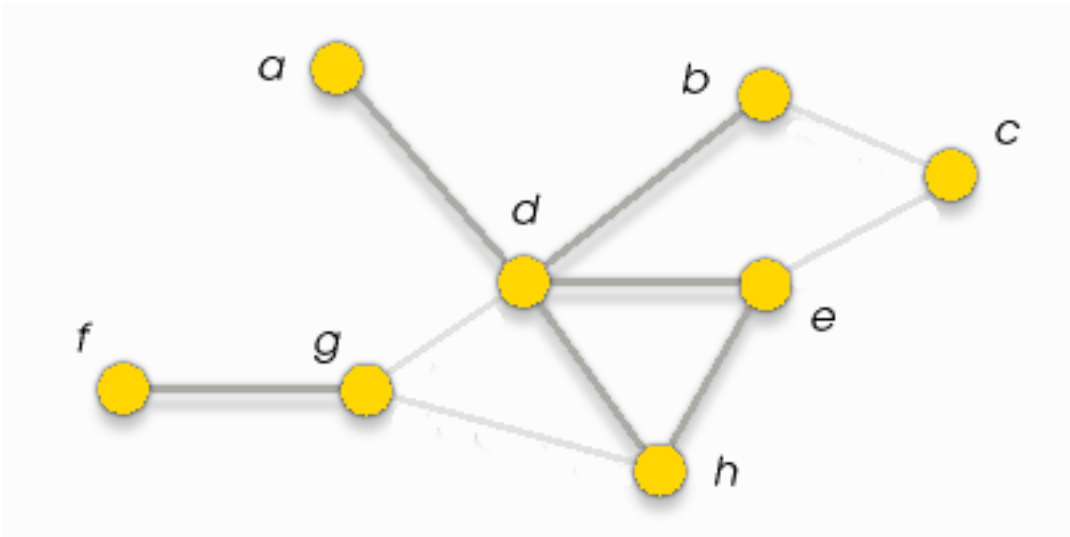
Un sous-graphe ***H*** de ***G*** est entièrement défini (induit) par ses sommets ***W***, et un graphe partiel ***I*** par ses arêtes ***F***.

Exemple Un exemple de sous-graphe et de graphe partiel sur le graphe de l' exemple introductif.



*Le sous-graphe **H** induit par l'ensemble*

***W={ b, c, d, g, h }** de sommets.*



*Le graphe partiel **I** défini par l'ensemble*

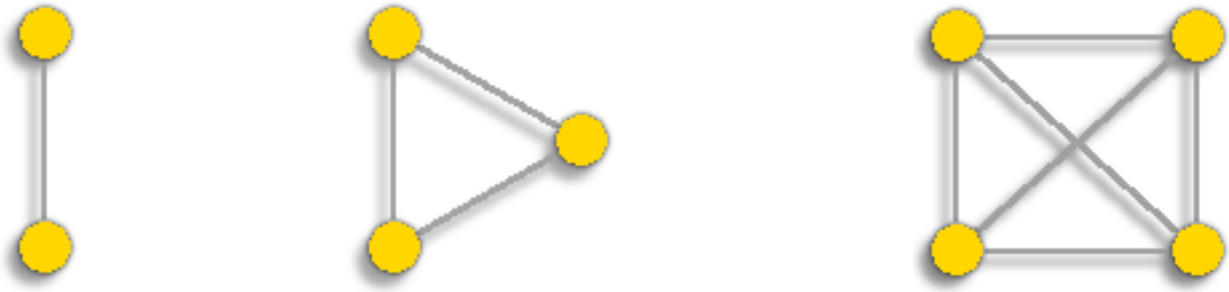
***F={ (a,d), (b,d), (d,e), (e,h), (h,d), (f,g) }** d'arêtes.*

Pour un graphe d'ordre ***n***, il existe 2 cas extrêmes pour l'ensemble de ses arêtes : soit le graphe n'a aucune arête, soit toutes les arêtes possibles pouvant relier les sommets 2 à 2 sont présentes. Dans ce dernier cas le graphe est dit *complet*. Pour un graphe général, il est souvent intéressant de rechercher de tels sous-graphes : on parle alors de *stable* et de *clique*.

Définition

Un **graphe complet** est un graphe où chaque sommet est relié à tous les autres. Le graphe complet d'ordre ***n*** est noté ***Kn***. Dans ce graphe chaque sommet est de degré ***n-1***.

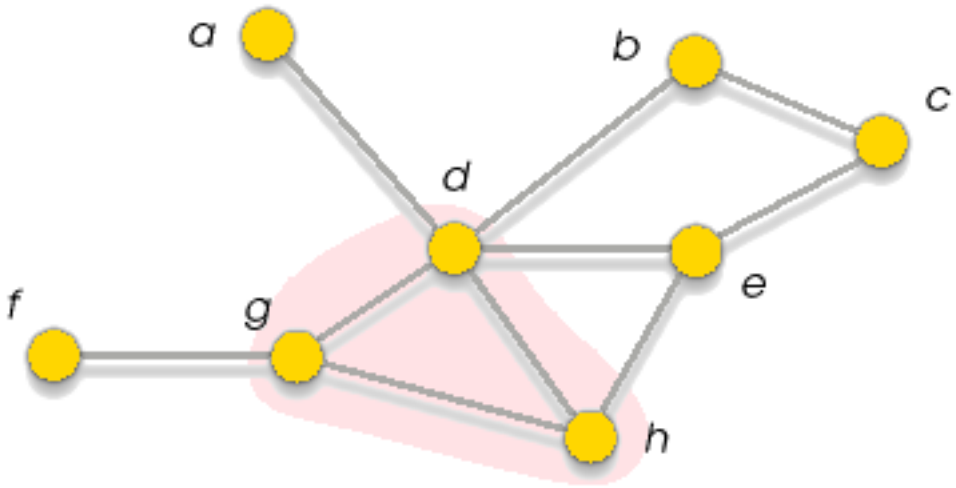
Exemple De gauche à droite sont représentés les graphes ***K2***, ***K3*** et ***K4***.



Définition

- Une **clique** est un sous-graphe complet.
- Un **stable** est un sous-graphe sans arête.

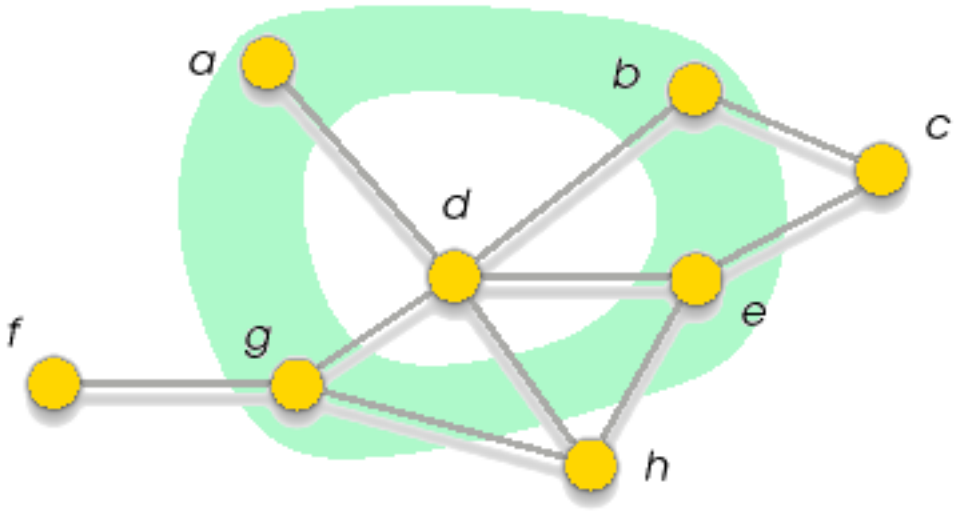
Exemple Reprenons le graphe de l'exemple introductif.



Le graphe ***G*** *admet 2 cliques d'ordre 3 définies par les ensemble de sommets*

{ d, g, h }* et *{ d, e, h }

La clique ***{ d, g, h }*** *est représentée en surimpression. Le graphe n'admet pas de clique d'ordre 4.*



Le graphe ***G*** *admet 4 stables d'ordre 4 définis par les ensemble de sommets*

{ a, b, e, g } *et* ***{ a, b, h, f }***
{ a, c, h, f } *et* ***{ a, b, e, f }***

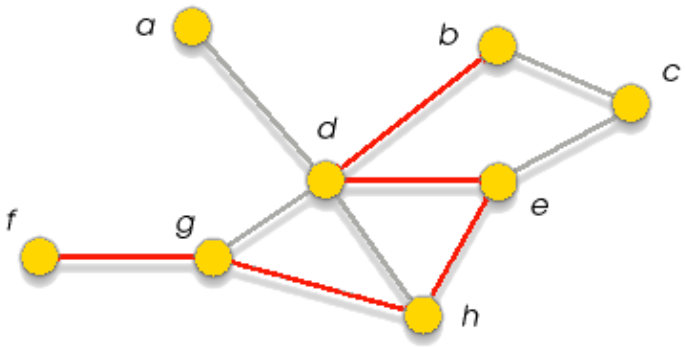
Le stable ***{ a, b, e, g }*** *est représenté en surimpression. Le graphe n'admet pas de stable d'ordre 5.*

Dans un graphe il est naturel de vouloir se déplacer de sommet en sommet en suivant les arêtes. Une telle marche est appelée une *chaine* ou un *chemin*. Un certain nombre de questions peuvent alors se poser : pour 2 sommets du graphe, existe-t-il un chemin pour aller de l'un à l'autre? Quel est l'ensemble des sommets que l'on peut atteindre depuis un sommet donné? Comment trouver le plus court chemin pour aller d'un sommet à un autre?

Définition **Chemin**

- Un **chemin** est une liste $p = (x_1, \ldots, x_k)$ de sommets telle qu'il existe dans le graphe une arête entre chaque paire de sommets successifs. $\forall i = 1, \ldots, k-1 \quad (x_i, x_{i+1}) \in E$
- La **longueur** du chemin correspond au nombre d'arêtes parcourues : ***k-1***.

Exemple Un chemin de longueur ***5*** dans le graphe reliant les sommets ***f*** à ***b***.



p=(f, g, h, e, d, b)

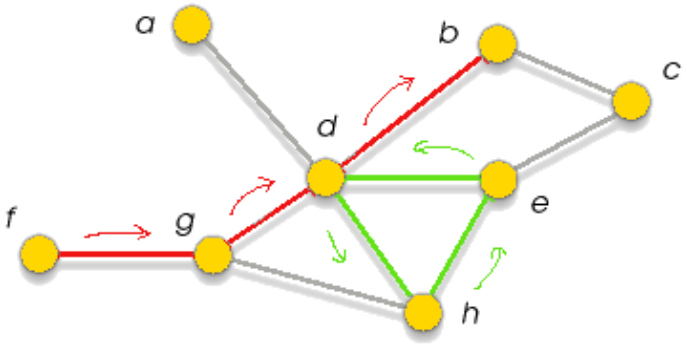
Remarque Il existe bien d'autres chemins pour aller de ***f*** à ***b*** : par exemple ***(f, g, d, b)*** de longueur 3, le chemin ***(f, g, d, h, e, d, b)*** de longueur 6, ou encore ***(f, g, d, h, e, d, h, e, d, b)*** de longueur 9, ... ***(d, h, e, d)*** est appelé un cycle. Ce cycle pouvant être emprunté autant de fois que l'on veut, il y a un nombre infini de chemins de ***f*** à ***b***

Définition **Chemin Simple et Cycle**

- Un chemin ***p*** est **simple** si chaque arête du chemin est empruntée une seule fois.
- Un **cycle** $c = (x_1, \ldots, x_k, x_{k+1})$ est un chemin simple finissant à son point de départ : $x_1 = x_{k+1}$

Exemple

Les chemins ***(f, g, d, b)*** et ***(f, g, d, h, e, d, b)*** sont simples. Le chemin ***(f, g, d, h, e, d, h, e, d , b)*** ne l'est pas : le cycle ***(d, h, e, d)*** est emprunté 2 fois.



Les termes de **chemin** et de **circuit** s'emploient en propre pour les graphes orientés. Pour les graphes non orientés que nous manipulons ici, on parle de **chaine** et de **cycle**. Cependant la définition formelle est exactement la même dans les 2 cas, seule change la structure (graphe orienté ou non) sur laquelle ils sont définis.

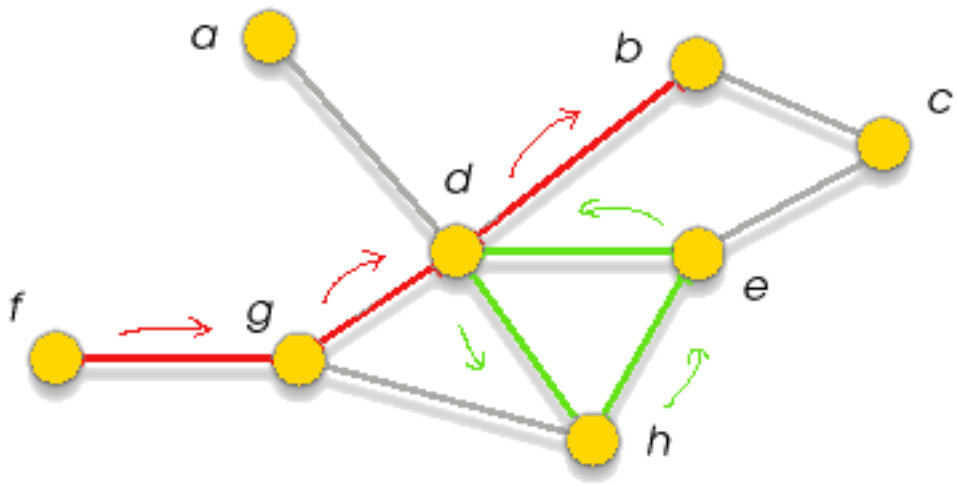
Pour aller d'un sommet à un autre à travers un graphe, un cycle constitue un détour peu naturel sur la route. Pour se restreindre à des chemins sans cycle, considérer les chemins simples ne suffit pas : il nous faut la notion de chemin élémentaire.

Définition — Chemin élémentaire

- Un chemin $p = (x_1, \ldots, x_k)$ est **élémentaire** si chacun des sommets du parcours est visité une seule fois : $\forall i,j = 1,\ldots,k,\ i \neq j, \quad x_i \neq x_j$
- Un chemin élémentaire est donc un chemin simple et sans cycle.

Exemple

Le chemin ***(f, g, d, b)*** est élémentaire, le chemin ***(f, g, d, h, e, d, b)*** ne l'est pas : le sommet ***d*** est visité 2 fois, ce qui crée le cycle ***(d, h, e, d)*.**



Propriété — Dans un graphe ***G*** d'ordre ***n***

- Tout chemin élémentaire est de longueur au plus ***n*-1**
- Le nombre de chemins élémentaires dans le graphe est fini.

Preuve. Un chemin élémentaire visitant au plus 1 fois chaque sommet du graphe, sa longueur (nombre d'arêtes) ne peut effectivement excéder ***n*-1**. Le nombre de chemins de longueur ***k*** (***k*=0, ..., *n*-1**) est au plus la combinatoire du choix d'une suite de ***k*+1** sommets parmi ***n*** : il y en a ***(k+1)!* C*(n,k+1)*** .

Les chemins élémentaires sont la restriction naturelle que nous recherchons à la notion de chemin. La question qui se pose est de savoir si nous "perdons" quelque chose en ne considérant que les chemins élémentaires dans un graphe : peut on toujours remplacer un chemin du graphe par un chemin élémentaire? Le *lemme de Koenig* répond affirmativement à cette question : de tout chemin on peut extraire un sous-chemin élémentaire.

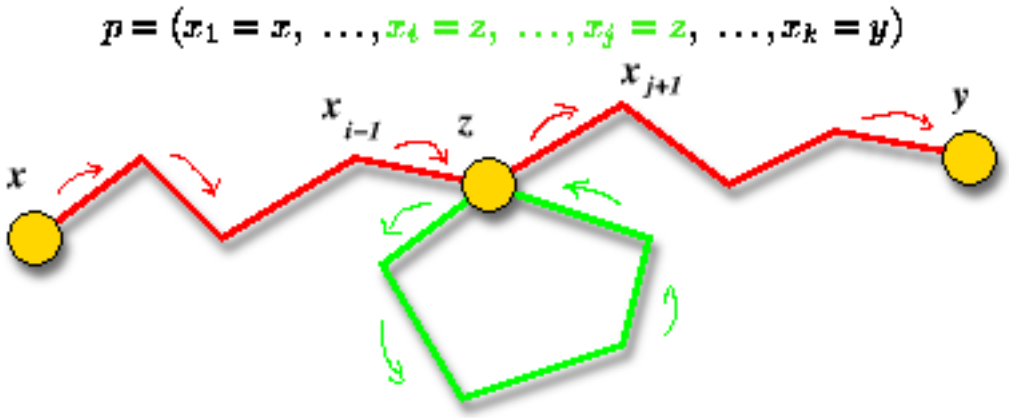
Propriété Lemme de Koenig

Si il existe un chemin entre 2 sommets ***x*** et ***y***, alors il existe un chemin élémentaire entre ***x*** et ***y***

HINT
Il suffit de choisir un plus court chemin

Preuve. L'idée de la preuve est de choisir un chemin particulier entre ***x*** et ***y*** et de montrer qu'il est élémentaire. Quel chemin choisir? Si un chemin comporte un circuit, ce circuit est un détour sur la route menant de ***x*** et ***y***. Un bon candidat à être un chemin élémentaire semble donc être un plus court chemin.

Parmi tous les chemins reliant ***x*** à ***y***, choisissons ainsi un chemin $p = (x_1 = x, \ldots, x_k = y)$ comportant le moins d'arêtes. Supposons par l'absurde que ***p*** n'est pas élémentaire. Il existe alors un sommet ***z*** apparaissant au moins 2 fois le long du chemin. Soient ***i*** et ***j*** les 2 premiers indices tels que $x_i = z$ et $x_j = z$.



Pour obtenir une contradiction, il suffit de "supprimer" le cycle entre $x_i = z$ et

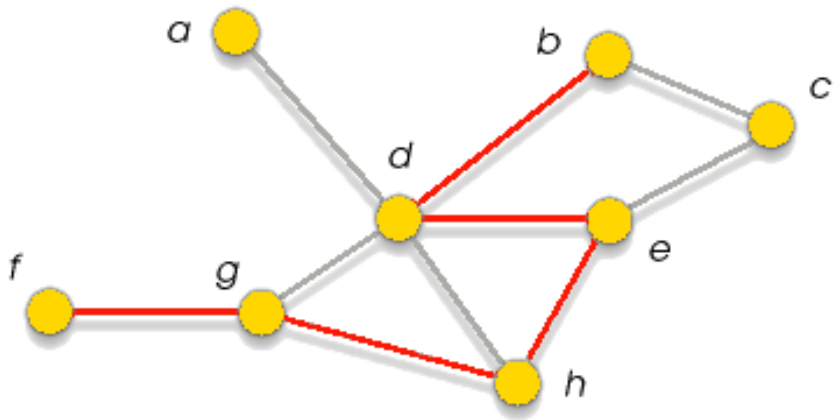
$x_j = z$. Alors $p' = (x_1 = x, \ldots, x_{i-1}, z, x_{j+1}, \ldots, x_k = y)$ est un chemin, reliant x à y. Sa longueur est strictement inférieure à celle de ***p***, ce qui contredit notre choix de ***p*** comme étant un plus court chemin.

Dans un graphe il est naturel de vouloir se déplacer de sommet en sommet en suivant les arêtes. Une telle marche est appelée une *chaine* ou un *chemin*. Un certain nombre de questions peuvent alors se poser : pour 2 sommets du graphe, existe-t-il un chemin pour aller de l'un à l'autre? Quel est l'ensemble des sommets que l'on peut atteindre depuis un sommet donné? Comment trouver le plus court chemin pour aller d'un sommet à un autre?

Définition **Chemin**

- Un **chemin** est une liste $p = (x_1, \ldots, x_k)$ de sommets telle qu'il existe dans le graphe une arête entre chaque paire de sommets successifs. $\forall i = 1, \ldots, k-1 \quad (x_i, x_{i+1}) \in E$
- La **longueur** du chemin correspond au nombre d'arêtes parcourues : ***k-1***.

Exemple Un chemin de longueur ***5*** dans le graphe reliant les sommets ***f*** à ***b***.



p=(f, g, h, e, d, b)

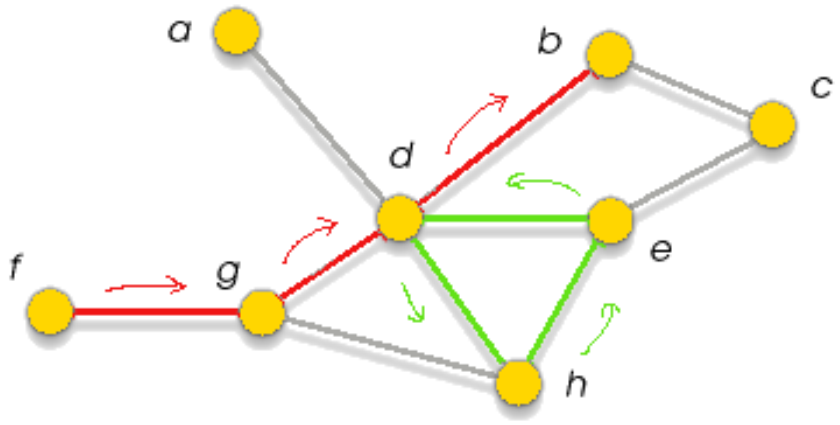
Remarque Il existe bien d'autres chemins pour aller de ***f*** à ***b*** : par exemple ***(f, g, d, b)*** de longueur 3, le chemin ***(f, g, d, h, e, d, b)*** de longueur 6, ou encore ***(f, g, d, h, e, d, h, e, d, b)*** de longueur 9, ... ***(d, h, e, d)*** est appelé un cycle. Ce cycle pouvant être emprunté autant de fois que l'on veut, il y a un nombre infini de chemins de ***f*** à ***b***

Définition Chemin Simple et Cycle

- Un chemin ***p*** est **simple** si chaque arête du chemin est empruntée une seule fois.
- Un **cycle** $c = (x_1, \ldots, x_k, x_{k+1})$ est un chemin simple finissant à son point de départ : $x_1 = x_{k+1}$

Exemple

Les chemins ***(f, g, d, b)*** et ***(f, g, d, h, e, d, b)*** sont simples. Le chemin ***(f, g, d, h, e, d, h, e, d , b)*** ne l'est pas : le cycle ***(d, h, e, d)*** est emprunté 2 fois.



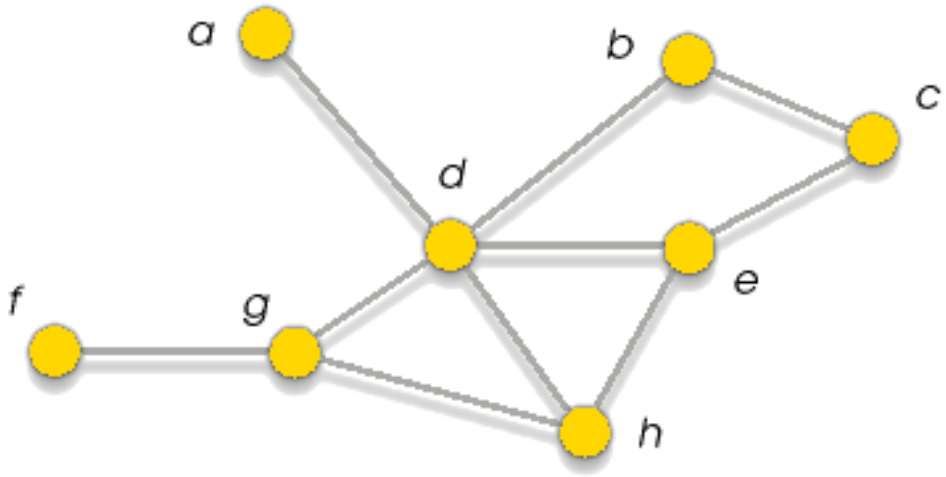
Les termes de **chemin** et de **circuit** s'emploient en propre pour les graphes orientés. Pour les graphes non orientés que nous manipulons ici, on parle de **chaine** et de **cycle**. Cependant la définition formelle est exactement la même dans les 2 cas, seule change la structure (graphe orienté ou non) sur laquelle ils sont définis.

La notion de connexité est liée à l'existence de chemins dans un graphe : depuis un sommet, existe-t-il un chemin pour atteindre tout autre sommet? Les graphes connexes correspondent à la représentation naturelle que l'on se fait d'un graphe. Les graphes non connexes apparaissent comme la juxtaposition d'un ensemble de graphes : ses composantes connexes.

Définition **Connexité**

Un graphe est **connexe** ssi il existe un chemin entre chaque paire de sommets.

Exemple



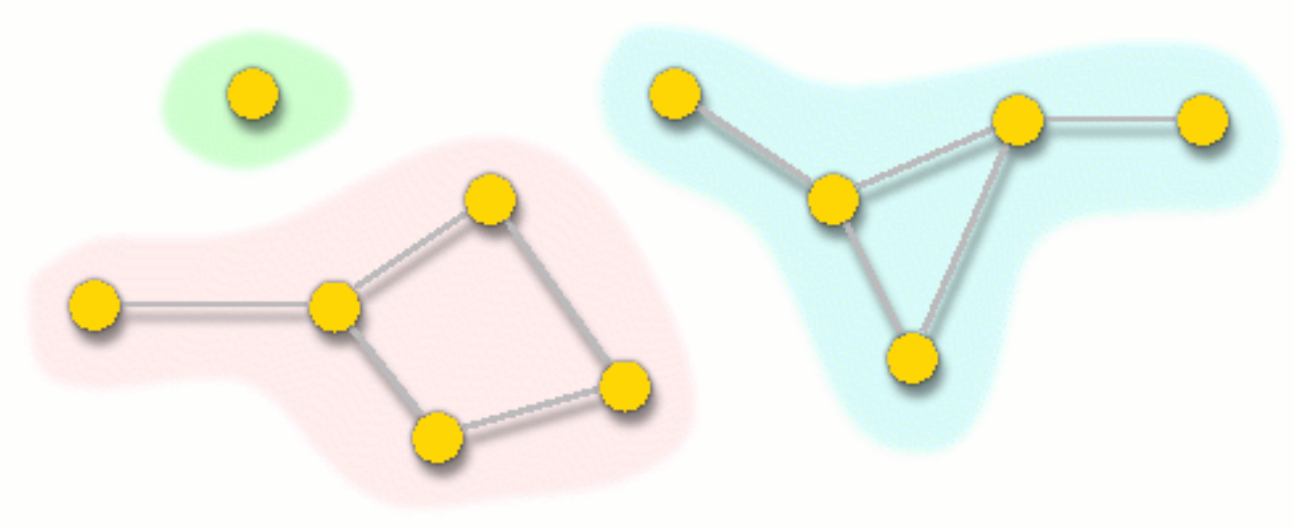
Le graphe nous servant d'illustration depuis le début est un graphe connexe.

Que se passe-t-il si le graphe ***G*** n'est pas connexe? Il apparaît alors comme un ensemble de graphes connexes "mis" les uns à coté des autres. Chacun de ces graphes est un sous-graphe particulier de ***G***, appelé *composante connexe*. Il est souvent utile de se placer sur les composantes connexes d'un graphe pour se ramener au cas d'un graphe connexe.

Définition — Composante connexe

Une composante connexe d'un graphe ***G*** est un sous-graphe ***G'=(V',E')*** connexe maximal (pour l'inclusion) : il n'est pas possible d'ajouter à ***V'*** d'autres sommets en conservant la connexité du sous-graphe.

Exemple



Le graphe ci-contre possède 3 composantes connexes, dont un sommet isolé.

Remarque

Un graphe ne possédant qu'une seule composante connexe est simplement un graphe connexe.
Un sommet isolé (de degré 0) constitue toujours une composante connexe à lui seul.
La relation sur les sommets "il existe un chemin entre ..." est une relation d'équivalence (réflexive, symétrique et transitive). Les composantes connexes d'un graphe correspondent aux classes d'équivalences de cette relation.

Existe-t-il une relation entre le nombre d'arêtes d'un graphe et sa connexité? On sent que pour connecter un graphe il faut qu'un minimum d'arêtes soient présentes pour qu'existent suffisamment de chemins. En fait, pour qu'un graphe ***G=(V,E)*** soit connexe, il faut qu'il ait au moins ***|V|-1*** arêtes.

Propriété

Un graphe ***G*** d'ordre ***n*** connexe comporte au moins ***n*-1** arêtes.

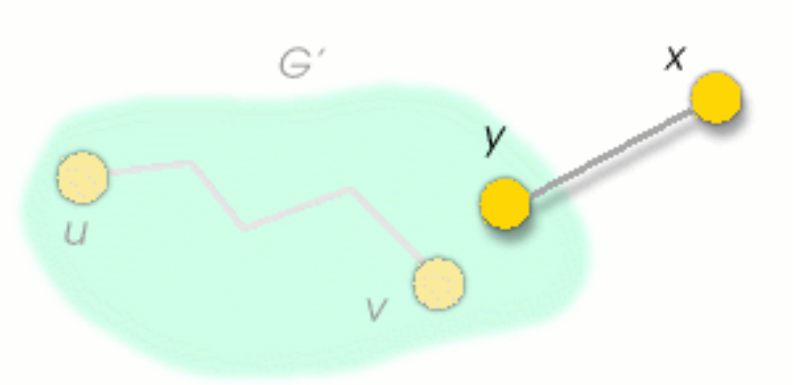
HINT
Preuve par induction sur le nombre de sommets en utilisant la somme des degrés

Preuve. La propriété peut se montrer par induction sur l'ordre du graphe. Les preuves par induction sont un outil courant en théorie des graphes, pour la bonne raison que les graphes sont des structures discrètes : le nombre de sommets et d'arêtes sont des entiers.

n=1. La propriété est clairement vraie.
n+1. Supposons la propriété prouvée sur les graphes d'ordre inférieur à n. Considérons un graphe ***G=(V,E)*** connexe d'ordre ***n+1***. Nous allons distinguer 2 cas : soit il existe dans ***G*** un sommet de degré 1, et alors l'hypothèse d'induction nous permettra de conclure, soit tout sommet est de degré au moins 2. Dans ce dernier cas un argument direct nous prouvera la propriété.

- **1er cas :** Il existe dans ***G*** un sommet ***x*** de degré 1.
  Une seule arête, appelons la ***e=(x,y)***, est alors incidente à ***x***. Considérons le sous-graphe ***G'=(V\{x},E')*** induit par tous les sommets à l'exception de ***x***. En particulier l'ensemble ***E'*** des arêtes de ***G'*** est exactement ***E*** moins l'arête ***e***. Le graphe ***G'*** reste connexe : si l'on considère 2 sommets ***u*** et ***v***, il existe dans ***G*** un chemin ***p***, que nous pouvons choisir élémentaire (Lemme de Koenig), reliant ces 2 sommets. Le chemin ***p*** étant élémentaire, il ne peut passer par le sommet ***x***, à moins fatalement de visiter au moins 2 fois le sommet ***y*** ! Par suite ***p*** est également un chemin de ***G'*** reliant ***u*** et ***v***.

  L'hypothèse d'induction impose que ***G'*** comporte au moins ***n*-1** arêtes.
  Cependant ***G'*** possède exactement un sommet et une arête de plus que ***G***.



- **2ème cas :** Il n'existe pas dans ***G*** de sommet de degré 1.
  Puisque ***G*** est connexe d'ordre au moins 2, il ne peut exister de sommet isolé. Tout sommet de ***G*** est donc de degré au moins 2. L'hypothèse d'induction est

alors plus délicate à utiliser : il nous faut choisir un sommet x tel que le sous-graphe G' reste connexe. A la place nous pouvons utiliser un argument direct avec la somme des degrés : $2|E|$ doit alors être supérieur à $2|V|$, ce qui nous permet de conclure.

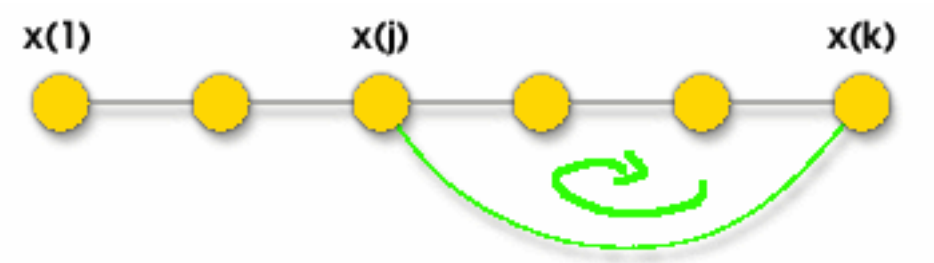
Un cycle est un chemin simple rebouclant sur lui-même. Un graphe dans lequel il n'existe aucun cycle est dit *acyclique*. Les graphes acycliques constituent une classe intéressante de graphes, avec des propriétés remarquables et un nom : les forêts. Il existe des relations fortes entre l'existence d'un cycle dans un graphe, le degré des sommets, et le nombre d'arêtes.

Propriété Existence d'un cycle

Si dans un graphe ***G*** tout sommet est de degré supérieur ou égal à 2, alors ***G*** possède au moins un cycle.

HINT
Algorithme de marquage

Preuve. La preuve utilise un algorithme de marquage. Initialement tous les sommets sont non marqués. Un sommet ***x*(1)** est marqué arbitrairement. L'algorithme construit alors une séquence ***x*(1),...,*x*(k)** de sommets marqués en choisissant arbitrairement pour ***x*(i+1)** un sommet non marqué adjacent à ***x*(i)**. L'algorithme s'arrête lorsque ***x*(k)** ne possède plus de voisin non marqué. Puisque ce sommet est de degré au moins 2, il possède, outre ***x*(k-1)**, un autre voisin ***x*(j)** dans la séquence, **j<k-1**. Alors **(*x*(k), *x*(j), *x*(j+1), ..., *x*(k-1), *x*(k))** est un cycle.

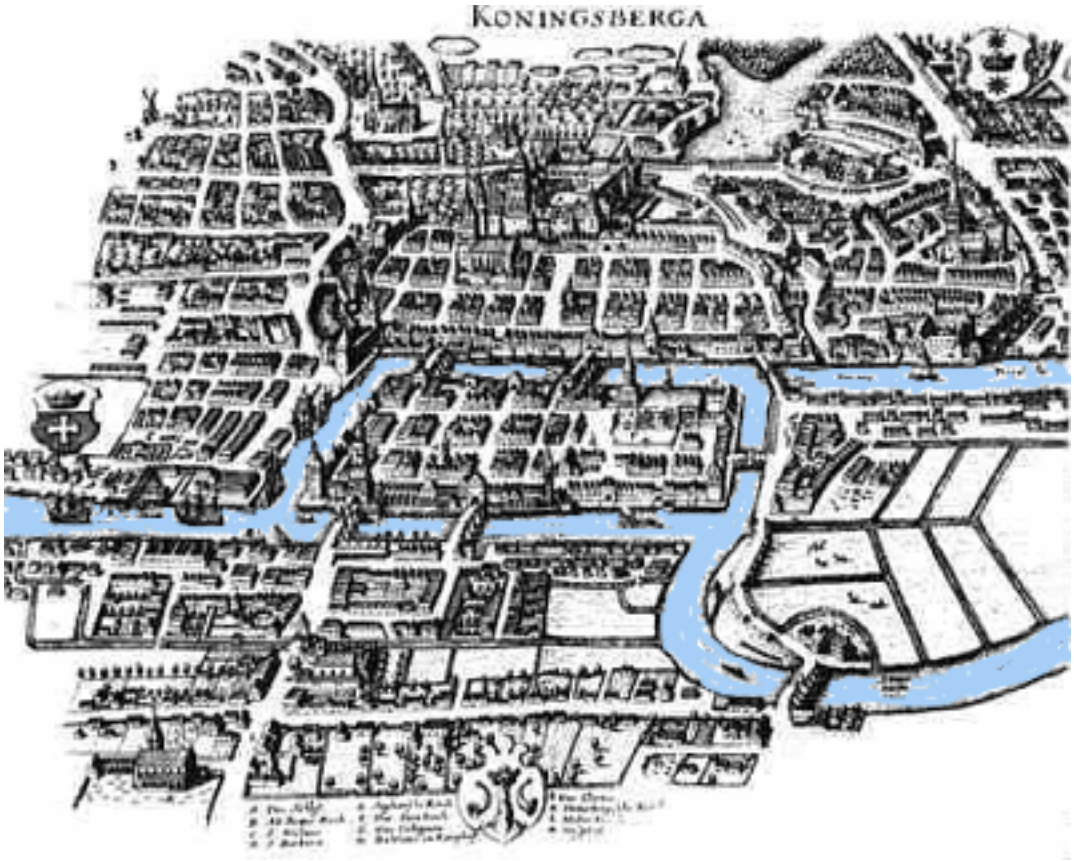


Cette propriété simple implique qu'un graphe sans cycle possède au moins un sommet de degré 0 ou 1.
A l'inverse, nous pouvons lier cette fois l'absence de cycle dans un graphe avec le nombre d'arêtes.

Propriété Graphe acycle

Un graphe acyclique ***G*** à ***n*** sommets possède au plus ***n*-1** arêtes.

Preuve. Nous pouvons montrer ce résultat par induction sur le nombre de sommets du graphe. Si ***G*** est d'ordre **1**, il ne possède aucune arête et la propriété est évidemment vérifiée. Supposons la propriété vraie à l'ordre ***n*** et établissons la à l'ordre ***n*+1**. Considérons donc un graphe ***G=(V,E)*** sans cycle à ***n*+1** sommets. Il existe un sommet ***x*** de degré au plus **1**. Soit ***G'=(V',E')*** le sous-graphe d'ordre ***n*** induit par les sommets ***V'=V\{x}***. Le graphe ***G'*** est clairement sans cycle, ce qui implique par l'hypothèse d'induction qu'il possède au plus ***n*-1** arêtes. Or **d(*x*)<2** impose que ***E*** différe de ***E'*** par au plus une arête, de la forme ***(x,y)***. Par suite |***E***| est inférieure à ***n***..

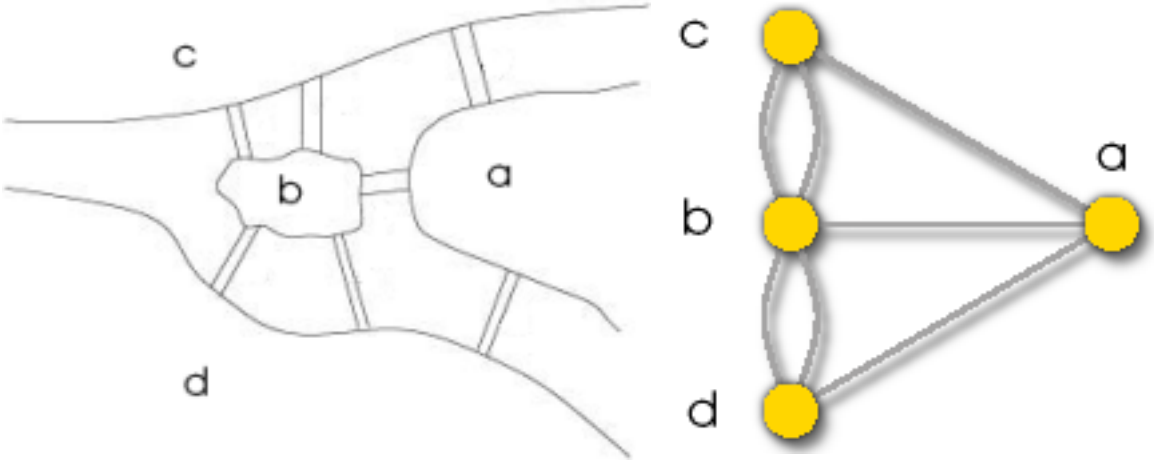


Euler, l'un des plus grands mathématiciens, aimait à faire une promenade dans sa bonne ville de Koenigsberg. Il affectionnait tout particulièrement de parcourir les 7 ponts qui enjambent la rivière.

L'âge venant, il se demanda si sans sacrifier à sa promenade, il pouvait en raccourcir la longueur en ne parcourant chaque pont qu'une seule fois. Ce problème est sans doute l'un des plus anciens en théorie des graphes : celui de l'existence d'un cycle passant une et une seule fois par chaque arête.

Définition **Graphe Eulérien**

Un **cycle eulérien** est un cycle passant une et une seule fois par chaque arête du graphe.
Un graphe est dit Eulérien si il admet un cycle eulérien.



La rivière sépare la ville de Koenigsberg en 4 partie, ***a, b, c*** et ***d***. Un pont relie 2 de ces parties

On peut alors représenter notre problème par un graphe avec 4 sommets, où chaque arête représente l'un des 7 ponts de Koenigsberg. Sur cet exemple le graphe n'est pas un graphe simple.

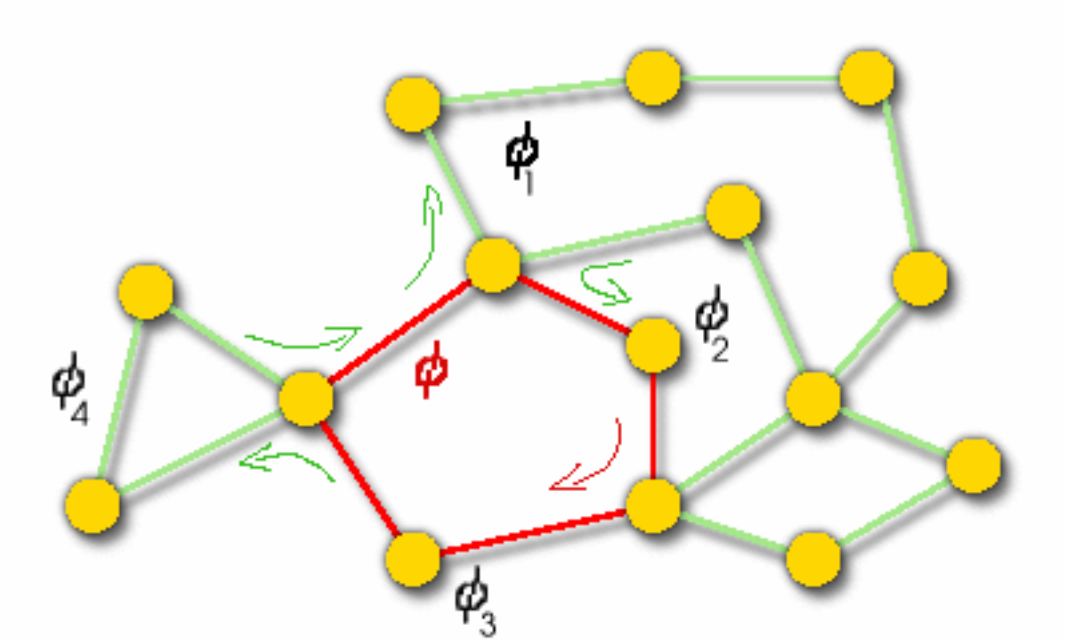
Comment savoir si un graphe est eulérien ou non ? Si pour notre problème le graphe obtenu est eulérien, il faut exhiber un cycle eulérien, ce qui ne semble pas facile. Mais si il ne l'est pas ? Euler a donné une caractérisation très forte des graphes eulériens : un graphe est eulérien ssi il est connexe et tous ses sommets sont de degré pair. Avec cette caractérisation, les sommets ***a, b, c*** et ***d*** étant de degré impair, on sait immédiatement qu'il est imposible de parcourir tous les ponts de Koenigsberg seulement une fois au cours d'une promenade.

Un graphe est eulérien ssi il est connexe et tous ses sommets sont de degré pair.

Preuve. Supposons qu'un graphe ***G*** soit eulérien. Il existe alors un cycle ***c*** parcourant une et une seule fois chaque arête. Le graphe ***G*** est donc connexe, puisque ***c*** relie tous les sommets entre eux. Considérons un sommet ***x***. Lors du parcours du cycle, à chaque fois que nous passons par lui, nous y arrivons et nous en repartons par 2 arêtes non encore parcourues. Le sommet ***x*** est donc de degré pair.

Réciproquement considérons un graphe ***G*** connexe dont tous les sommets de degré pair. Nous allons montrer par induction sur le nombre d'arêtes que ***G*** est alors eulérien.

- Si ***G*** se réduit à un unique sommet isolé, il est évidemment eulérien.
- Sinon tous les sommets de ***G*** sont de degré supérieur ou égal à 2. Ceci implique qu'il existe un cycle ϕ sur ***G***. Considérons le graphe partiel ***H*** constitué des arêtes en dehors du cycle ϕ. Les sommets de ***H*** sont également de degré pair, le cycle contenant un nombre pair d'arêtes incidentes pour chaque sommet. Par induction chaque composante connexe ***Hi*** de ***H*** est un graphe eulérien, et admet donc un cycle eulérien ϕ_i. Pour reconstruire un cycle eulérien sur ***G***, il nous suffit de fusionner le cycle ϕ avec les différents cycles ϕ_i. Pour cela on parcours le cycle ϕ depuis un sommet arbitraire; lorsque l'on rencontre pour la première fois un sommet ***x*** appartenant à ***Hi***, on lui substitue le cycle ϕ_i. Le cycle obtenu est un cycle eulérien pour ***G***, le cycle ϕ et les cycles ϕ_i formant une partition des arêtes.

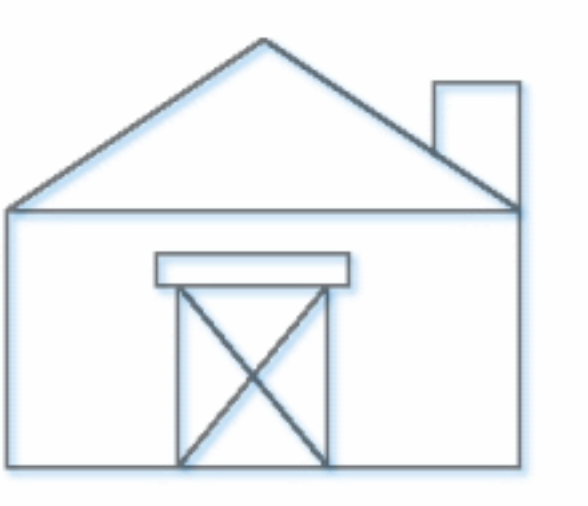


Le cycle ϕ représenté en rouge définit 4 composantes connexes pour le graphe ***H***, dont 2 sommets isolés pour lesquels leur cycle eulérien est sans arête.

Les flèches vertes symbolisent l'opétation de fusion des 2 cycles non vide avec ϕ .

Les graphes eulériens ont de nombreuses applications, certaines ludiques.

Par exemple est-il possible de dessiner cette maison sans lever le crayon, et bien sûr sans repasser par le même trait ?



Ce problème correspond à déterminer si un graphe admet un *chemin eulérien*

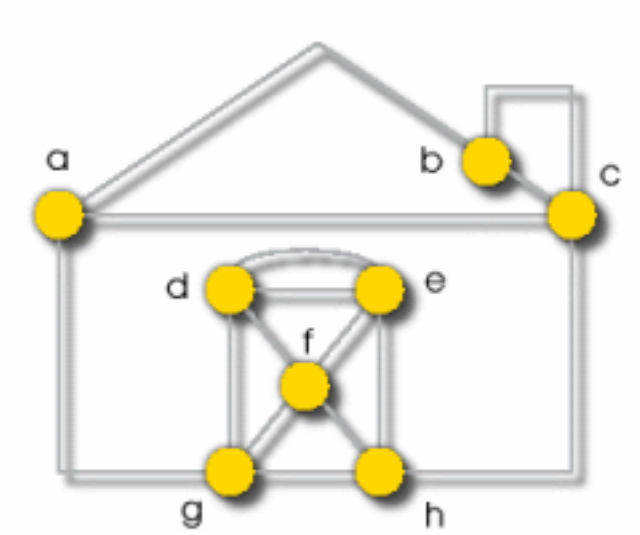
Définition **Chemin Eulérien**

Un **chemin eulérien** est un chemin passant une et une seule fois par chaque arête du graphe.
Un graphe admet un chemin eulérien ssi il est connexe et au plus 2 de ses sommets sont de degré impair.

Les conditions d'existence d'un chemin eulérien sont une conséquence du théorème pour les graphes eulériens. En effet si un graphe ***G*** n'admet pas de sommet de degré impair, il est alors eulérien, et il suffit de prendre comme chemin un cycle eulérien. Sinon le graphe ***G*** admet, de part la propriété des degrés, exactement 2 sommets de degré impair, disons ***x*** et ***y***. En ajoutant l'arête ***(x,y)***, on obtient un graphe eulérien : la suppression de l'arête ***(x,y)*** de son cycle eulérien montre l'existence dans ***G*** d'un chemin eulérien entre ***x*** et ***y***.

Le graphe représentant notre maison comporte 2 sommets de degré impair : ***a*** et ***b***. En ajoutant cette arête le graphe devient eulérien.

Il admet par conséquent un chemin eulérien entre ces 2 sommets. Autrement dit, il est possible dessiner la maison sans lever le trait.



La recherche d'un chemin eulérien dans un graphe revient à la recherche d'un cycle eulérien. Si nous connaissons les conditions d'existence pour un cycle eulérien, nous n'avons pas donner de méthode (un algorithme) permettant de le construire. La preuve du théorème des graphes eulériens nous fournit une idée pour construire un cycle eulérien sur un graphe, en recherchant récursivement des cycles sur des composantes connexes et en les fusionnant au fur et à mesure. L'algorithme récursif prend en entrée un graphe, non nécessairement connexe, dont tous les sommets sont de degré pair, et un sommet ***x***. Il retourne comme résultat un cycle eulérien sur la composante connexe de ***x***. Pour ce faire il commence par construire un cycle ϕ contenant ***x*** en utilisant un algorithme proche de l'algorithme de marquage utilisé dans la preuve de l'existence d'un cycle sur les graphes de degré supérieure à 2. Ce cycle ϕ est ensuite parcouru, et pour chacun de ses sommets x_i l'algorithme est rappelé récursivement pour construire un cycle ϕ_i. Les différents cycles sont finalement concaténés entre eux pour construire un cycle eulérien sur la composante connexe. L'opération de concaténation entre 2 chemins, ***(u,...,v)*** ○ ***(u',...,v')***, retourne le chemin ***(u,...,v,u',...,v')***.

`ALGORITHME` Euler
`ENTREES` ***G=(V,E)*** un graphe dont tous les sommets sont de degré pair,***x*** un sommet de ***V***
`SORTIE` ϕ un cycle eulérien sur la composante connexe de ***x***

ϕ : LISTE des sommets du cycle dans l'ordre de parcours
Initialiser ϕ := (***x***)
*// Base de la récursivité : **x** est isolé*
`Si` ***x*** est un sommet isolé
`Alors`
    `Retourner` ϕ
`Sinon` *// On construit un cycle contenant **x***
    Initialiser ***y*** := ***x***
    `Tant Que` ***y*** n'est pas un sommet isolé

Choisir ***z*** l'un de ses voisins
Supprimer l'arête ***(y,z)*** ; ***y*** := ***z***
$\phi \leftarrow y$ *// on ajoute le sommet au cycle*
`Fin TantQue`
*// Appel récursif sur chacun des **k** sommets du cycle ϕ en concaténant les réponses*
`Retourner` ***Euler***(***G***, ϕ (1)) ○ ... ○ ***Euler***(***G***, ϕ (k))
`Fin Si`

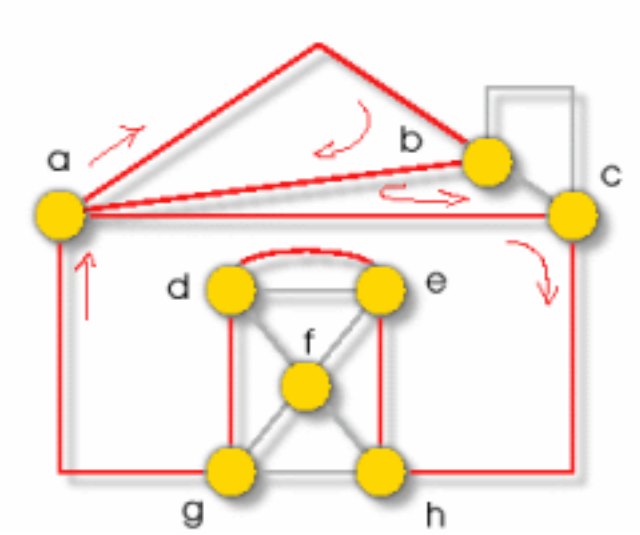
Théoreme

L'algorithme d'Euler construit un cycle eulérien en temps ***O(|E|)*** sur tout graphe eulérien ***G=(V,E)***

Preuve. Tout d'abord remarquons que la première phase de l'algorithme construit bien un cycle contenant ***x***. En effet chaque fois que nous arrivons et repartons d'un sommet dans notre marche, nous supprimons 2 de ses arêtes incidentes. Tous les sommets étant de degré pair, seul le sommet de départ, ***x***, peut être déconnecté lors de l'arrivée à ce sommet. Le fait que l'algorithme construit un cycle eulérien peut alors se montrer par induction sur le nombre d'arêtes du graphe, de manière similaire à la preuve du théorème d'Euler. Les arêtes du graphe étant supprimées au fur et à mesure de la construction, elles apparaissent bien exactement une fois dans le cycle final.

Complexité. A chaque fois qu'une arête du graphe est parcourue dans la première phase de l'algorithme pour construire le cycle ϕ, elle est supprimée. Le parcours du cycle pour effectuer les appels récursifs se fait en temps proportionnel au nombre de ses arêtes. La complexité totale de l'algorithme est donc bien en ***O(|E|)***. Il est optimal en temps, puisqu'un cycle eulérien comprend ***|E|*** arêtes.

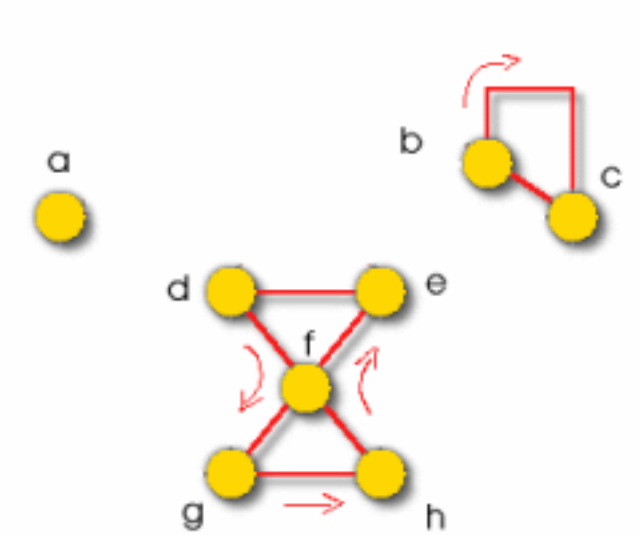
Si nous appliquons l'algorithme d'Euler à notre maison, après ajout de l'arête ***(a,b)*** pour le rendre eulérien.



La première phase construit par exemple le cycle ***a,b,a,c,h,e,d,g,a*** en partant du sommet ***a***.

Evidemment d'autres choix dans le parcours du graphe peuvent conduire à des cycles différents, voire directement à un cycle eulérien.

Récursivement l'algorithme est appelé sur chacun des somemts du cycle.



- Le sommet ***a*** étant isolé, l'algorithme retourne immédiatement ***(a)***.
- Pour le sommet ***b***, l'algorithme construit récursivement le cycle ***(b,c,b)***.
- Le sommet ***c*** étant maintenant isolé, l'algorithme retourne ***(c)***.
- Pour le sommet ***h***, l'algorithme construit le cycle ***(h,f,e,d,f,g,h)***.
- Les sommets restant à visiter sur le cycle, ***(e,d,g,a)***, sont désormais tous isolés.

L'algorithme retourne le cycle ***(a)*** ○ ***(b,c,b)*** ○ ***(a)*** ○ ***(c)*** ○ ***(h,f,e,d,f,g,h)*** ○ ***(e)*** ○ ... ○ ***(a)***
c'est à dire ***(a,b,c,b,a,c,h,f,e,d,f,g,h,e,d,g,a)***

Lorsqu'un chemin existe entre deux sommets dans un graphe, l'être humain se pose rapidement la question non seulement de trouver un tel chemin, mais bien souvent il est intéressé par le *plus court chemin* possible entre ces deux sommets. Notre oeil est d'ailleurs particulièrement efficace dans cette tâche, tant que le graphe est de taille raisonnable... Mais dès que le graphe comporte plusieurs dizaines de sommets et d'arêtes, trouver le plus court chemin entre deux points devient vite un casse-tête : quel est l'itinéraire entre Ménilmontant et la Rue du Bac passant par le minimum de stations de métro?

Nous aimerions disposer d'une méthode systématique capable, pour tout graphe et pour toute paire de sommets ***s*** et ***t***, de déterminer le plus court chemin entre eux. Résoudre ce problème va donc consister à proposer un algorithme, aussi rapide que possible. Le problème de trouver un plus court chemin est le premier *problème d'optimisation* auquel nous sommes confrontés. Nous allons moins nous intéresser ici à l'algorithme permettant de le résoudre, BFS, au final très simple, qu'à la genèse d'un tel algorithme, à la démarche permettant d'aborder la résolution d'un tel problème.

La première étape est de comprendre et de caractériser au mieux la structure d'une solution : quelles sont les propriétés d'un plus court chemin? Le lemme de Koenig nous donne une première caractéristique intéressante : un plus court chemin est élémentaire.

Propriété

Un plus court chemin entre 2 sommets est élémentaire

HINT

Lemme de Koenig

Cette propriété nous donne un premier algorithme pour trouver un plus court chemin. Le nombre de chemins élémentaires entre ***s*** et ***t*** étant finis, nous pouvons tous les énumérer en calculant leur longueur et retenir le plus court. Cet algorithme est correct mais révèle un grave défaut : son temps d'exécution. Si le nombre de chemins élémentaires est bien fini, il n'en demeure pas moins très grand, de l'ordre de ***n!*** sur un graphe d'ordre ***n***. Si nous retenons cette estimation, sur un graphe de seulement 20 sommets, en supposant que trouver un chemin et calculer sa longueur puissent se faire en une seule opération, sur un ordinateur pouvant effectuer 1 milliard d'opérations par seconde, il nous faudra patienter 77 ans pour obtenir le meilleur trajet. Et si le graphe a 25 sommets, armons nous de patience pour les ... 490 millions d'années à venir!

Pour éviter la combinatoire d'explorer tous les chemins élémentaires du graphe, nous avons besoin d'une seconde propriété, fondamentale, la *sous-optimalité*. Il s'agit simplement de remarquer qu'être un plus court chemin, ca s'hérite : le "bout" d'un plus court chemin est encore un plus court chemin entre ses extrémités. Ou encore si le trajet le plus rapide entre ***s*** et ***t*** passe par 2 sommets ***x*** et ***y***, ce trajet emprunte nécessairement le plus court chemin entre ***x*** et ***y***.

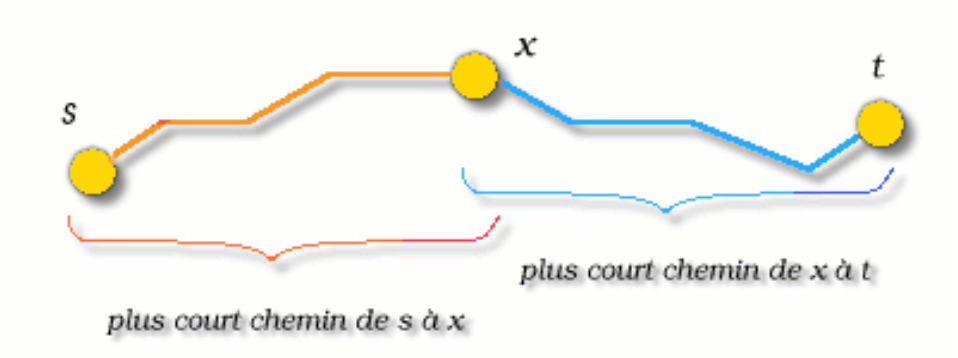
Propriété **Principe de sous-optimalité**

Si ***p=(s,...,t)*** est un plus court chemin entre ***s*** et ***t***, alors pour tout sommet ***x*** sur le chemin ***p***,

- le sous-chemin de ***p*** jusqu'à ***x***, ***(s,...,x)***, est un plus court chemin de ***s*** à ***x***
- le sous-chemin de ***p*** depuis ***x***, ***(x,...,t)***, est un plus court chemin de ***x*** à ***t***

HINT

Argument d'échange



Preuve. La preuve utilise un argument d'échange. Par l'absurde supposons qu'il existe par exemple entre ***s*** et ***x*** un chemin ***q*** de longueur strictement inférieur à celle de ***(s,...,x)***. Alors nous pouvons construire un nouveau chemin ***p'*** entre ***s*** et ***t*** en subsituant ***q*** à ***(s,...,x)*** dans le chemin ***p***. La longueur de ***p'*** est alors strictment inférieure à celle de ***p***, ce qui contredit que ***p*** est un plus court chemin de ***s*** à ***t***.

Le principe de sous-optimalité montre que trouver le plus court chemin entre deux sommets ***s*** et ***t*** conduit à trouver tous les plus court chemins entre ***s*** et les sommets ***x*** sur le trajet. Bien entendu nous ne savons pas *a priori* quels seront les sommets sur le chemin (nous l'aurions déjà trouvé!). Pour chercher le plus court chemin entre ***s*** et ***t***, nous allons en fait résoudre le problème de *trouver tous les plus court chemins* entre ***s*** et les autres sommets du graphe.

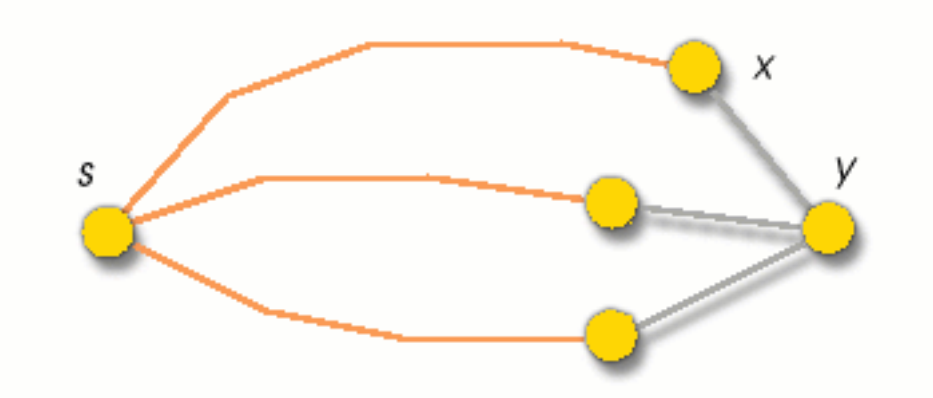
> Nous allons plus particulièrement nous intéresser à calculer la *longueur* de tous les plus courts chemins, c'est à dire pour tout sommet ***y***, sa distance ***D(y)*** au sommet ***s***.

Revenons sur le principe de sous-optimalité avec un point de vue plus "local", algorithmique. Le plus court chemin, ***p***, entre ***s*** et un sommet ***y*** visite nécessairement, juste avant d'arriver à ***y***, l'un de ses voisins, ***z*** : ***p=(s, ..., z, y)***. Le principe de sous-optimalité implique alors que ***D(y) = D(z)*** **+1** Evidemment nous ne savons pas plus *a priori* lequel des voisins de ***y*** joue le rôle de ***z***. Cependant pour tout voisin ***x***, par définition du plus court chemin, ***D(x)*****+1** est supérieur ou égal à ***D(y)***.

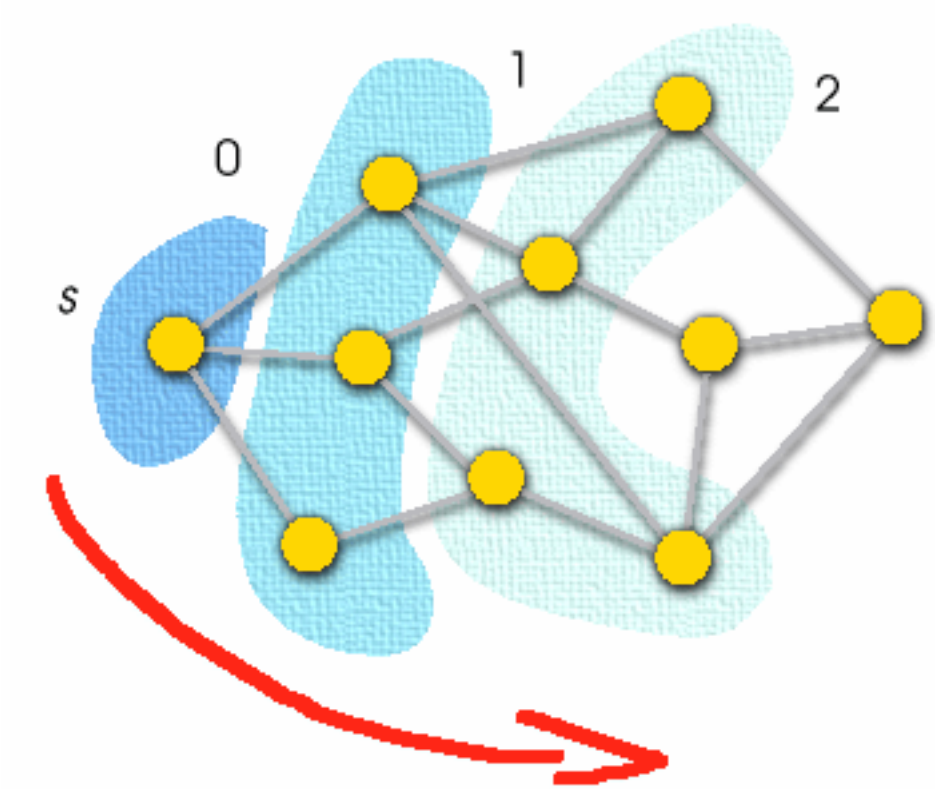
Propriété

La longueur ***D(y)*** du plus court chemin de ***s*** à ***y*** vérifie

D(y) = **0** si ***y=s***
D(y) = **1 +** **min {** ***D(x)*** **|** ***x voisin de y*** **}** sinon



Pas mal... Cette propriété semble nous donner une formule récursive pour calculer ***D()***. Malheureusement il n'en est rien! Il apparaît vite que le calcul de ***D(x)*** demande le calcul de ***D(y)*** qui dépend lui-même de ***D(x)***... Pour ne pas tourner ainsi éternellement en rond, une idée intuitive est de visiter le graphe en calculant de proche en proche les distances ***D(y)***et en ne tenant compte que des voisins ***x*** déjà visités, pour lesquels leur distance ***D(x)***est donc déjà connue. Evidemment une telle visite nous permet de déterminer les bonnes distances seulement si nous visitons les sommets dans le bon ordre, des plus proches aux plus éloignés de ***s***



Il s'agit donc de partir de ***s*** (distance 0), et de visiter successivement ses voisins (distance 1), puis les voisins de ses voisins non encore visités (distance 2), les voisins des voisins des voisins non encore visités, et ainsi de suite...

Lorsqu'un chemin existe entre deux sommets dans un graphe, l'être humain se pose rapidement la question non seulement de trouver un tel chemin, mais bien souvent il est intéressé par le *plus court chemin* possible entre ces deux sommets. Notre oeil est d'ailleurs particulièrement efficace dans cette tâche, tant que le graphe est de taille raisonnable... Mais dès que le graphe comporte plusieurs dizaines de sommets et d'arêtes, trouver le plus court chemin entre deux points devient vite un casse-tête : quel est l'itinéraire entre Ménilmontant et la Rue du Bac passant par le minimum de stations de métro?

Nous aimerions disposer d'une méthode systématique capable, pour tout graphe et pour toute paire de sommets ***s*** et ***t***, de déterminer le plus court chemin entre eux. Résoudre ce problème va donc consister à proposer un algorithme, aussi rapide que possible. Le problème de trouver un plus court chemin est le premier *problème d'optimisation* auquel nous sommes confrontés. Nous allons moins nous intéresser ici à l'algorithme permettant de le résoudre, BFS, au final très simple, qu'à la genèse d'un tel algorithme, à la démarche permettant d'aborder la résolution d'un tel problème.

La première étape est de comprendre et de caractériser au mieux la structure d'une solution : quelles sont les propriétés d'un plus court chemin? Le lemme de Koenig nous donne une première caractéristique intéressante : un plus court chemin est élémentaire.

Propriété

Un plus court chemin entre 2 sommets est élémentaire

HINT
Lemme de Koenig

Cette propriété nous donne un premier algorithme pour trouver un plus court chemin. Le nombre de chemins élémentaires entre ***s*** et ***t*** étant finis, nous pouvons tous les énumérer en calculant leur longueur et retenir le plus court. Cet algorithme est correct mais révèle un grave défaut : son temps d'exécution. Si le nombre de chemins élémentaires est bien fini, il n'en demeure pas moins très grand, de l'ordre de ***n!*** sur un graphe d'ordre ***n***. Si nous retenons cette estimation, sur un graphe de seulement 20 sommets, en supposant que trouver un chemin et calculer sa longueur puissent se faire en une seule opération, sur un ordinateur pouvant effectuer 1 milliard d'opérations par seconde, il nous faudra patienter 77 ans pour obtenir le meilleur trajet. Et si le graphe a 25 sommets, armons nous de patience pour les ... 490 millions d'années à venir!

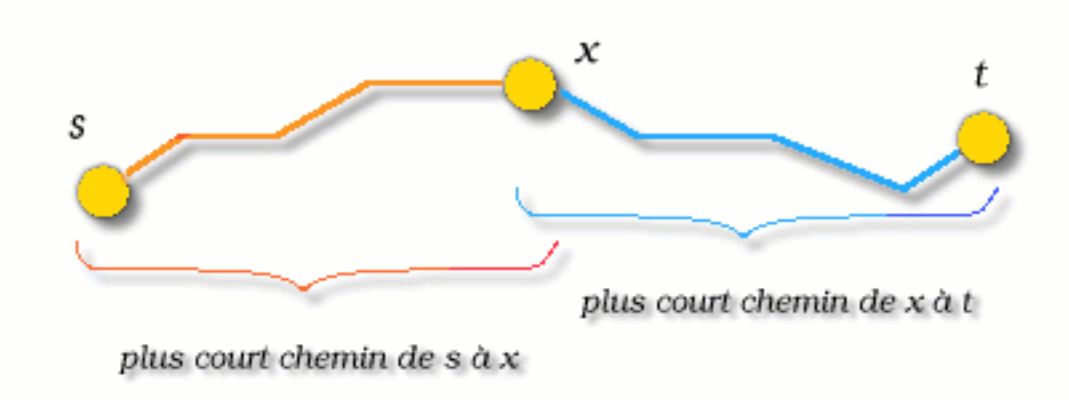
Pour éviter la combinatoire d'explorer tous les chemins élémentaires du graphe, nous avons besoin d'une seconde propriété, fondamentale, la *sous-optimalité*. Il s'agit simplement de remarquer qu'être un plus court chemin, ca s'hérite : le "bout" d'un plus court chemin est encore un plus court chemin entre ses extrémités. Ou encore si le trajet le plus rapide entre ***s*** et ***t*** passe par 2 sommets ***x*** et ***y***, ce trajet emprunte nécessairement le plus court chemin entre ***x*** et ***y***.

Propriété **Principe de sous-optimalité**

Si ***p=(s,...,t)*** est un plus court chemin entre ***s*** et ***t***, alors pour tout sommet ***x*** sur le chemin ***p***,

- le sous-chemin de ***p*** jusqu'à ***x***, ***(s,...,x)***, est un plus court chemin de ***s*** à ***x***
- le sous-chemin de ***p*** depuis ***x***, ***(x,...,t)***, est un plus court chemin de ***x*** à ***t***

HINT
Argument d'échange



Preuve. La preuve utilise un argument d'échange. Par l'absurde supposons qu'il existe par exemple entre ***s*** et ***x*** un chemin ***q*** de longueur strictement inférieur à celle de ***(s,...,x)***. Alors nous pouvons construire un nouveau chemin ***p'*** entre ***s*** et ***t*** en subsituant ***q*** à ***(s,...,x)*** dans le chemin ***p***. La longueur de ***p'*** est alors strictment inférieure à celle de ***p***, ce qui contredit que ***p*** est un plus court chemin de ***s*** à ***t***.

Pour visiter le graphe depuis le sommet ***s***, nous allons utiliser un *algorithme de recherche*. Cet algorithme est un algorithme de marquage très simple qui correspond bien à l'exploration que nous ferions naturellement d'un graphe en visitant les sommets de proche en proche.
Au départ tous les sommets sont non marqués, à l'exception de notre sommet de départ, ***s***. A chaque étape nous sélectionnons simplement un sommet non marqué, adjacent à un sommet déjà marqué, et nous le marquons à son tour. Le choix d'un sommet à marquer voisin d'un sommet déjà marqué assure que l'ensemble des sommets marqués est un sous-graphe connexe à chaque étape de l'algorithme.

`ALGORITHME` Recherche
`ENTREES` Graphe ***G=(V,E)***, Sommet ***s***

Initialiser tous les sommets à non marqué ; Marquer ***s***

`TantQue` il existe un sommet non marqué adjacent à un sommet marqué

> Sélectionner un sommet ***y*** non marqué adjacent à un sommet marqué
> Marquer ***y***

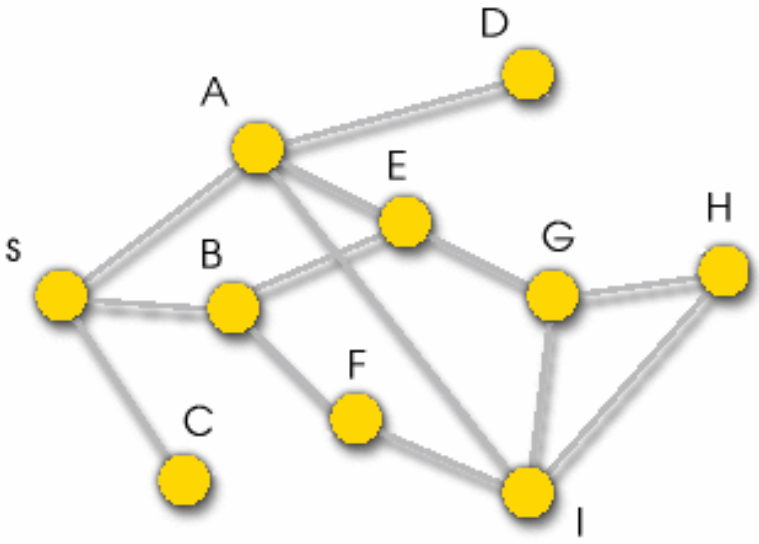
`FinTantQue`

L'algorithme de recherche est simple au point d'en devenir sommaire. Et pourtant, le monde est beau, ses différentes variantes permettent de résoudre tous les problèmes que nous pouvons nous poser : recherche des composantes connexes, recherche d'un arbre couvrant, détermination des plus courts chemins, ...

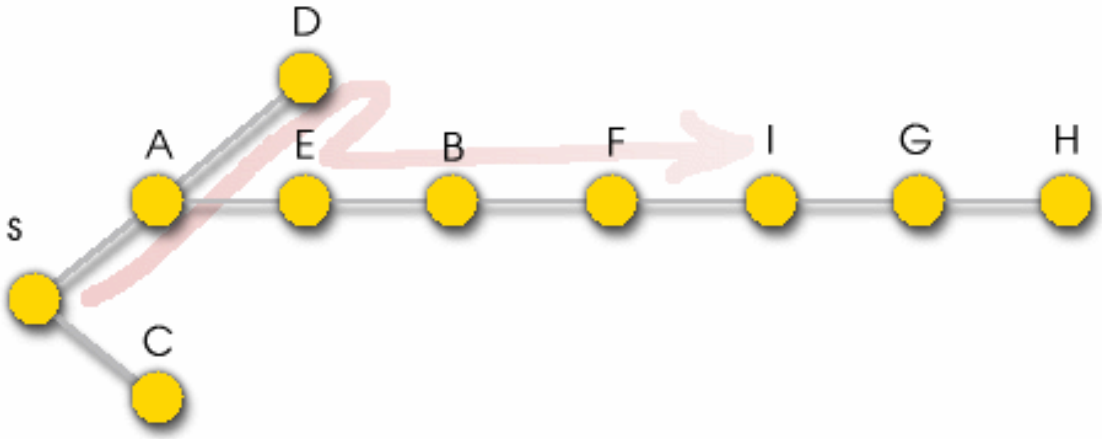
Il existe deux grandes stratégies "opposées" pour sélectionner à chaque étape le sommet à marquer :

- La **recherche en profondeur** DFS (*Depth First Search*)
  Dans l'exploration, l'algorithme cherche à aller très vite "profondément" dans le graphe, en s'éloignant du sommet ***s*** de départ. La recherche sélectionne à chaque étape un sommet voisin du sommet marqué à l'étape précédente.

- La **recherche en largeur** BFS (*Breath First Search*).
  Dans l'exploration l'algorithme cherche au contraire à épuiser la liste des sommets proches de ***s*** avant de poursuivre l'exploration du graphe.

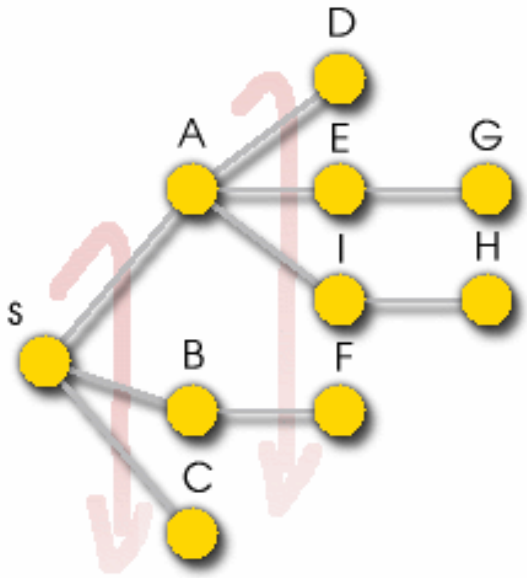


Un exemple de recherches possibles en profondeur et en largeur. Le résultat de chacune des recherche est représenté par son arbre de recherche : une arête existe entre deux sommets ***x*** et ***y*** si la visite (marquage) de ***y*** se fait à partir de ***x***.



RECHERCHE EN PROFONDEUR

Les sommets sont visités depuis ***s*** dans l'ordre ***A, D, E, B, F, I, G, H, C***



RECHERCHE EN LARGEUR

Les sommets sont visités depuis ***s*** dans l'ordre ***A, B, C, D, E, I, F, G, H*** Remarquez que le chemin de ***s*** à un sommet dans l'arbre de recherche est un plus court chemin dans ***G***

La recherche en largeur semble bien correspondre à l'ordre de visite du graphe que nous cherchons pour résoudre le problème des plus courts chemins. Pour expliciter plus précisément (algorithmiquement) ce qu'est une recherche en largeur, revenons sur notre algorithme de recherche. Nous l'avons présenté du point de vue des sommets non marqués. Nous pouvons à l'inverse nous placer du côté des sommets déjà marqués et sélectionner l'un de leurs voisins non encore marqués.

Pour cela, nous maintenons dans une structure de données ***M*** l'ensemble des sommets marqués dont nous n'avons pas encore traité les voisins. Une étape de la recherche consiste à sélectionner l'un de ces sommets (en le retirant de ***M***), à marquer tous ses voisins (non encore marqués) et à les ajouter à ***M***. La structure de données ***M*** doit donc posséder 2 actions : l'ajout d'un sommet ***x*** (noté $M \leftarrow x$), et le retrait du sommet en tête de la structure (noté $x \leftarrow M$).

`ALGORITHME` Recherche
`ENTREES` Graphe ***G=(V,E)***, Sommet ***s***

M : structure de données ordonnée
Initialiser tous les sommets à non marqué ; Marquer ***s***
M $\leftarrow$ ***s***

`Tant Que` ***M*** n'est pas vide
    x $\leftarrow$ ***M*** *// Retirer le "premier" sommet de l'ensemble **M***
    `Pour` chaque voisin ***y*** non marqué de ***x***
        Marquer ***y***
        M $\leftarrow$ ***y*** *// Ajouter **y** à l'ensemble **M***
    `Fin Pour`
`Fin TantQue`

La **Recherche en Profondeur** correspond à prendre pour ***M*** une structure de PILE, c'est à dire une liste LIFO *LastIn/FirstOut* (dernier entré/premier sorti)

La **Recherche en Largeur** correspond à prendre pour ***M*** une structure de FILE, c'est à dire une liste FIFO *FirstIn/FirstOut* (premier entré/premier sorti)

Les arbres sont des graphes particuliers, très populaires en algorithmiques et en informatique.

Définition **Arbre et Forêt**

Un arbre est un graphe connexe sans cycle.
Une forêt est un graphe sans cycle.

Une forêt étant un graphe sans cycle, chacune de ses composantes connexes est acyclique, et par définition connexe. La définition d'une forêt correspond donc bien au sens usuel d'un ensemble d'arbres, chacune de ses composantes connexes étant un arbre.

Les 6 graphes suivants sont des arbres. Pris ensemble ils constituent une forêt, n'en déplaise à Magritte.

Il existe de nombreuses caractérisations pour les arbres que nous détaillons dans la propriété ci-dessous. Si nous regardons la propriété de connexité, les arbres apparaîssent comme des graphes connexes minimaux, au sens où retirer la moindre arête les déconnecte. Si nous regardons les arbres sous l'angle des graphes acycliques, ils sont maximaux, au sens où ajouter la moindre arête crée un cycle.
Il existe également une relation très forte entre le nombre d'arêtes et le nombre de sommets d'un arbre : un arbre $T=(V,E)$ comporte exactement $|V|-1$ arêtes. En effet nous avons vu que :

- Un graphe connexe comporte au moins $|V|-1$ arêtes (cf propriété des graphes connexes).
- Un graphe sans cycle comporte au plus $|V|-1$ arêtes (cf propriété des graphes acycliques).

Propriété — Caractérisations d'un Arbre

Pour un graphe T d'ordre n, il y a équivalence entre les propriétés :

1. T est un arbre
2. T est un graphe connexe à $n-1$ arêtes
3. T est connexe, et la suppression de toute arête le déconnecte
4. T est acyclique à $n-1$ arêtes
5. T est acyclique et l'ajout de toute arête le rend cyclique.

Preuve. Pour montrer l'équivalence entre ces 5 propriétés, le plus simple est d'établir une série d'implications.

(1) $\Rightarrow$ (2) Le graphe T étant à la fois connexe et acyclique, il possède exactement $n-1$ arêtes. (cf propriété des graphes connexes et propriété des graphes acycliques).

(2) $\Rightarrow$ (3) La suppression d'une arête de T donne un graphe à $n-2$ arêtes : il ne peut être connexe (cf propriété des graphes connexes).

(3) $\Rightarrow$ (4) Par l'absurde si T possèdait un cycle, la suppression d'une arête de ce cycle ne saurait le déconnecter. Par suite T est acyclique. Puisqu'il est également connexe, il possède donc exactement $n-1$ arêtes.

(4) ⇒ (5) L'ajout d'une arête à ***T*** donne un graphe à ***n*** arêtes : il ne peut être acyclique (cf propriété des graphes acycliques).

(5) ⇒ (1) Considérons 2 sommets ***x*** et ***y*** de ***T***. Si il existe l'arête ***(x,y)***, alors c'est un chemin de ***x*** à ***y***. Sinon ajoutons cette arête à ***T*** : nous créons alors un cycle, de la forme ***(x, a,..., z, y, x)***. Ceci montre l'existance du chemin ***(x, a,..., z, y)*** entre ***x*** et ***y*** dans ***T***. Par définition ***T*** est donc un graphe connexe.

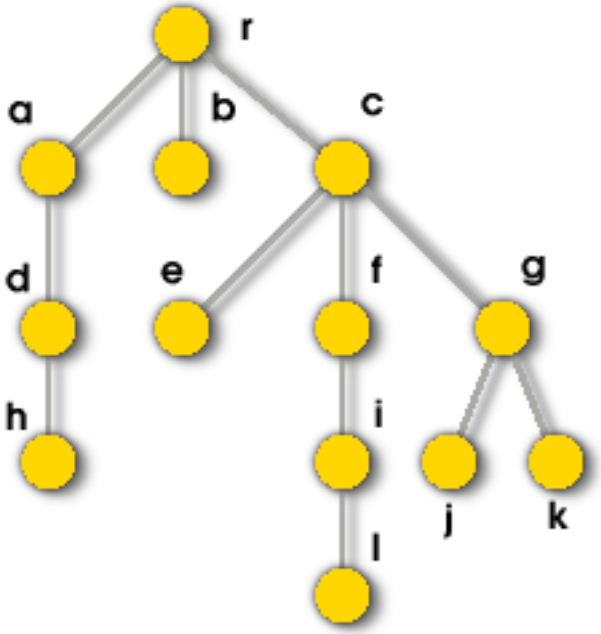
Souvent, pour manipuler un arbre, nous particularisons un sommet du graphe que nous appelons *racine*. Dans le cas des graphes non orientés, le choix d'une racine ***r*** dans l'arbre est arbitraire. Dans le cas des graphes orientés, la racine est définie de manière unique comme le sommet sans prédécesseur de l'arbre.
Le choix d'une racine revient dans un certain sens à orienter l'arbre, la racine apparaissant comme l'ancêtre commun à la manière d'un arbre généalogique. Le vocabulaire de la théorie des graphes s'en inspire directement : on parle de fils, de père, de frère,...

Définition **Feuille et hauteur**

Pour un arbre ***T*** de racine ***r***

- Le **père** d'un sommet ***x*** est l'unique voisin de ***x*** sur le chemin de la racine à ***x***. La racine ***r*** est le seul sommet sans père.
- Les **fils** d'un sommet ***x*** sont les voisins de ***x*** autres que son père.
- Une **feuille** est un sommet sans fils. Les feuilles correspondent aux sommets de degré 1.
- La **hauteur *h(T)*** de l'arbre ***T*** est la longueur du plus long chemin de la racine à une feuille.

La définition du père repose sur le fait que pour deux sommets ***u*** et ***v*** d'un arbre, il existe un et un seul chemin élémentaire reliant ***u*** et ***v***.
L'existence de 2 chemins distincts impliquerait en effet l'existence d'un cycle dans le graphe.



On représente habituellement un graphe de racine ***r*** par *niveaux*, à la façon d' un arbre généalogique. Au niveau 0 apparaît la racine, au niveau 1 ses fils, au niveau 2 les fils de ses fils, etc... Le niveau d'un sommet correspond à sa *profondeur* dans l'arbre, c'est à dire la longueur de son chemin à la racine.
Dans l'exemple l'arbre a une hauteur de 4. Les feuilles de l'arbres sont ***h, b, e, l, j, k***. Les fils de la racine sont ***a, b, c***. Le père de ***g*** est ***c***

Avec la recherche en largeur, nous avons toutes les cartes en main pour écrire un algorithme calculant la longueur de tous les plus court chemins depuis un sommet ***s***. Il suffit au cours de la visite de mettre à jour un label ***L*** pour chaque sommet. Le label ***L(y)*** du sommet ***y*** est calculé comme le label ***L(x)*** de son voisin ***x*** depuis lequel il est visité, plus 1. A la fin de l'algorithme, les labels sont égaux aux distances ***D(y)***.

`ALGORITHME` BFS
`ENTREES` Graphe ***G=(V,E)***, Sommet ***s***

F : FILE (liste FIFO)
Initialiser tous les sommets à non marqué ; Marquer ***s***
L(s) := 0
F $\leftarrow$ ***s***

`Tant Que` ***F*** n'est pas vide
    x $\leftarrow$ ***F*** *// Retirer le premier sommet de la file*
    `Pour` chaque voisin ***y*** non marqué de ***x***
        Marquer ***y***
        L(y) := ***L(x)*** + 1
        F $\leftarrow$ ***y*** *// Ajouter **y** à la fin de la file*
    `Fin Pour`
`Fin TantQue`

Théoreme

L'algorithme BFS calcule en temps ***O(|E|)*** la longueur des plus courts chemins du sommet ***s*** à tous les autres sommets du graphe

Pour tout ***x***, ***L(x)=D(x)***

Preuve. Clairement les labels ***L(x)*** correspondent à la longueur d'un chemin de ***s*** à ***x*** dans le graphe : celui du chemin dans l'arbre de recherche construit par BFS. Nous avons donc ***L(x)*** $\geq$ ***D(x)***
Par induction il est également facile de montrer que si un sommet ***u*** est ajouté à la file ***F*** avant un sommet ***v***, alors le label de ***u*** est inférieur au label de ***v***, du fait de la politique FIFO d'une file.

Par l'absurde, supposons qu'il existe des sommets dont le label calculé par BFS n'est pas leur distance à ***s***. Choisissons, parmi ces sommets, un sommet ***y*** dont la distance ***D(y)*** est la plus petite. Le sommet ***y*** à été visité au cours de la recherche depuis l'un de ses voisins ***x***, ce qui implique ***L(y) = L(x)+1 = D(x)+1***. Cependant puisque ***L(y) > D(y)***, il existe un second voisin ***z*** tel que ***D(y) = D(z)+1*** (principe de sous-optimalité) et ***D(x) > D(z)***. Par notre choix de ***y*** les labels de ***x*** et ***z*** sont égaux à leur distance à ***s***. Le sommet ***z*** a donc été ajouté à la file ***F*** avant le sommet ***x***. Ceci contredit que ***y*** ait été marqué depuis ***x*** et non depuis ***z***.

Complexité. Chaque étape de la visite consiste à retirer un sommet ***x*** de la file et à explorer ses voisins. Un sommet ***x*** entre exactement une fois dans la file : au moment de son marquage. Le nombre d'opérations de BFS est donc proportionnel à la somme du nombre de voisins de chaque sommet, c'est à dire la somme des degrés. La propriété sur les degrés montre que le nombre d'opérations de BFS est en ***O(|E|)***

BFS est optimal en temps : un algorithme doit considérer au moins une fois chaque arête du graphe si il ne veut pas manquer un plus court chemin.

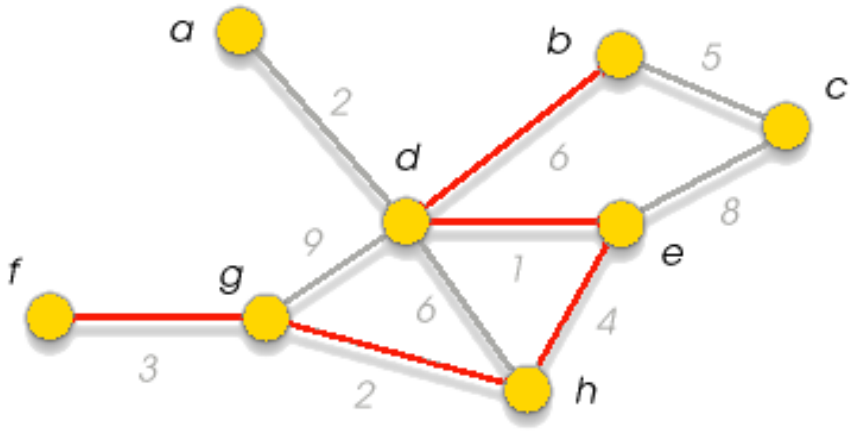
Nous allons généraliser notre notion de plus court chemin dans le cas de graphes valués, où chaque arête ***e*** est associée à une valeur, appelée souvent son poids, ***c(e)***. La valuation des arêtes peut représenter des coûts de transit, des distances kilométriques, le temps nécessaire pour parcourir les arêtes,... Ainsi le graphe ci-contre peut être valué avec les temps de voyage en train entre 2 villes.

Nous sommes alors intéressés par trouver non plus le plus court chemin en nombre d'arêtes (qui correspond sur notre exemple à minimiser le nombre de changement de train) mais le chemin de poids minimum, celui dont la somme des poids des arêtes est le plus faible possible (qui correspond sur notre exemple à minimiser le temps passé dans le train au cours du voyage).

Définition **Plus court chemin**

Dans un graphe valué, le **poids *c(p)*** d'un chemin ***p*** est la somme des poids des arêtes le long du chemin.
Dans ce qui suit nous appellerons le poids d'un chemin sa **longueur**. Le **plus court chemin** entre 2 sommets ***s*** et ***t*** est alors défini comme le chemin de plus faible poids reliant ***s*** et ***t***.

Pour différencier la longueur d'un chemin telle que que nous l'avons définie pour un graphe non valué, nous préciserons qu'il s'agit de la longueur en nombre d'arêtes. Cette confusion des 2 définitions de la longueur se justifie par le fait que la longueur en nombre d'arêtes correspond au poids du chemin avec une valuation implice de 1 pour toutes les arêtes.



Dans le

graphe valué ci-contre, ***p=(f, g, h, e, d, b)*** a une longueur de 16. Il est le plus court chemin reliant les sommets ***f*** et ***b***

Nous allons nous intéresser dans un premier temps à des graphes sur lesquels toutes les valuations des arêtes sont positives (nous verrons un peu plus loin ce qui peut se passer lorsque certaines valuations sont négatives). Sous cette condition, les propriétés que nous avons données pour le plus court chemin en nombre d'arêtes restent vérifiées pour le chemin de plus faible poids :

Propriété **Principe de sous-optimalité**

Dans un graphe avec une valuation positive ***c()*** de ses arêtes :

- Un plus court chemin entre 2 sommets est élémentaire
- Les plus courts chemins vérifient le principe de **sous-optimalité** : si ***p=(s,...,t)*** est un plus court chemin entre ***s*** et ***t***, alors pour tout sommet ***x*** sur le chemin, le sous-chemin de ***p*** jusqu'à ***x***, ***(s,...,x)***, est un plus court chemin de ***s*** à ***x***
- Si ***D(y)*** est la longueur du plus court chemin d'un sommet ***y*** à ***s***, le principe de sous-optimalité se traduit localement par les égalités :

  D(y)* = 0** si ***y=s
  ***D(y)* = min { *D(x)* + *c(x,y)* | *x voisin de y* }** sinon

HINT
Démonstration identique au plus court chemin en nombre d'arêtes (élémentaire,sous-optimalité)

Pour trouver les plus court chemins dans un graphe valué, nous allons voir 2 algorithmes très différents :

- *Algorithme de Dijkstra*. Cet algorithme est une adaptation de l'algorithme de recherche pour calculer les plus court chemin d'un sommet ***s*** à tous les autres sommets du graphe.
- *Algorithme de Bellman*. Cet algorithme est plus complexe, plus long en temps, mais plus puissant : il détermine la distance de tous les plus courts chemins entre toutes les paires de sommets d'un graphe, pouvant avoir des valuations négatives ou positives de ses arêtes. Son principe repose sur un paradigme algorithmique très général : la programmation dynamique.

L'algorithme de Dijkstra détermine les plus court chemins dans un graphe avec des valuations positives de ses arêtes, entre un sommet ***s*** et tous les autres sommets. Nous allons nous intéresser plus particulièrement au calcul de la longueur ***D(x)*** d'un plus court chemin entre ***s*** et ***x***.

L'algorithme de Dijkstra est un algorithme de marquage, une adaptation pour le problème du plus court chemin de l'algorithme général de recherche. Comme pour la longueur en nombre d'arêtes, l'idée est d'explorer le graphe dans le "bon" ordre, des sommets les plus proches de ***s*** aux sommets les plus éloignés pour pouvoir au fur et à mesure calculer leur distance à ***s***. Ce principe avait conduit à l'algorithme bfs de recherche en largeur.

Imaginons que nous ayons déjà marqué avec notre algorithme de recherche un ensemble ***S*** de sommets, pour lesquels nous avons déterminé leur plus court chemin à ***s***. Un "bon" ordre d'exploration revient à sélectionner le prochain sommet ***y*** à marquer de telle sorte que nous puissions calculer ***D(y)*** uniquement à partir des distances ***D(x)*** des sommets ***x*** marqués. Une telle exploration est possible grace à la propriété suivante :

Propriété **Distance partielle**

Considérons un ensemble ***S*** de sommets incluant la source ***s***. Pour tout sommet ***z*** en dehors de ***S***, nous définissons sa distance partielle à ***s*** par

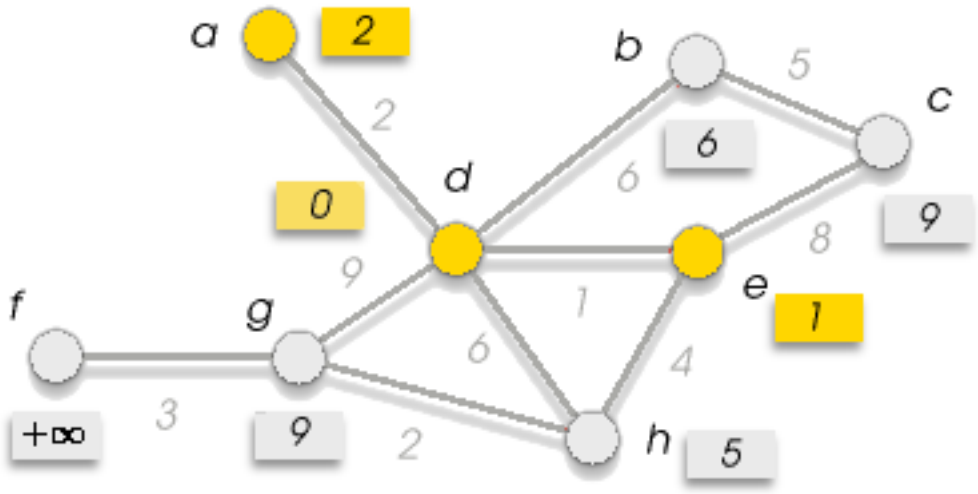
HINT
Principe de sous-optimalité

$$DP(S,z) = \min\{ D(x) + c(x,z) \mid x \in S \text{ et } x \text{ voisin de } z \}$$

Alors si ***y*** est un sommet avec la plus petite distance partielle, sa distance partielle est égale à sa distance à ***s*** :

$$DP(S,y) = D(y)$$

La distance partielle correspond à la longueur du plus court chemin de ***s*** à ***z*** qui n'emprunte que des sommets de ***S***, à l'exception bien sûr du dernier sommet ***z***. C'est donc le plus court chemin dans le sous-graphe induit par $S \cup \{z\}$. Par convention la distance partielle d'un sommet est infinie si il n'est adjacent à aucun sommet de ***S***.



Dans l'exemple ci-contre le sommet ***d*** joue le rôle de la source ***s***. L'ensemble ***S*** des sommets marqués est **{*a, d, e*}**. Les distances sont indiquées à coté de chaque sommet, en grisé pour les distances partielles des sommets non marqués, en jaune pour les plus courtes distances des sommets marqués.

La distance partielle de ***f*** est infinie, sa distance à ***d*** est ***D(f)* =10**
La distance

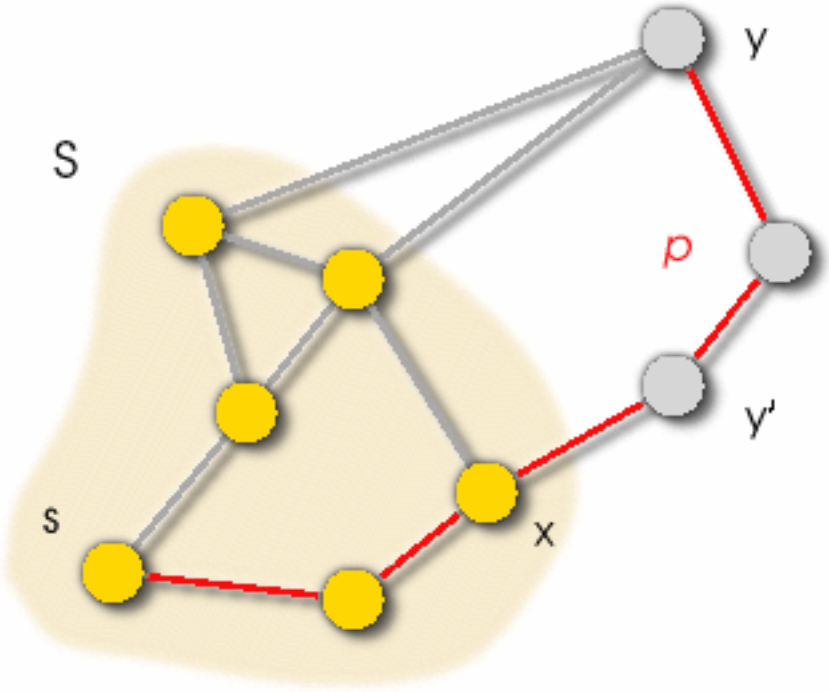
partielle de ***g*** est ***DP(S,g)*** **=9**, sa distance est ***D(g)*** **=7**
Par contre pour ***h***, le sommet de plus petite distance partielle, on a bien l'égalité ***DP(h)*** **=** ***D(h)*** **= 5**

Preuve.

La preuve se fait par l'absurde en utilisant le principe de sous-optimalité. Supposons que pour un sommet ***y*** avec la plus petite distance partielle, on ait $D(y) \neq DP(S,y)$. La distance partielle correspondant à la longueur d'un chemin dans le graphe, on a donc de fait l'inégalité $D(y) < DP(S,y)$.

Pour obtenir une contradiction considérons un plus court chemin ***p*** de ***s*** à ***y***, et sur ce chemin intéressons nous au premier sommet ***y'*** n'appartenant pas à ***S*** que nous rencontrons en partant de ***s***. La longueur du chemin ***p*** est par définition ***D(y)***; elle est aussi d'après le principe de sous-optimalité $D(y') + c(y',...,y)$. Nous avons donc $D(y') \leq D(y)$

Cependant le sommet ***y'*** étant non marqué, de par notre choix de ***y*** nous avons $DP(S,y) \leq DP(S,y')$. En mettant bout à bout ces inégalités, on obtient $D(y') < DP(S,y')$. Pour conclure intéressons nous au sommet ***x*** précédant ***y'*** sur le chemin ***p*** ; le principe de sous-optimalité implique que $D(y') = D(x) + c(x,y')$. Mais de par la définition de la distance partielle, le sommet ***x*** appartenant à ***S***, on a $DP(S,y') \leq D(x) + c(x,y')$. Contradiction.



Nous sommes maintenant prêts pour donner un algorithme de recherche des plus court chemins. L'idée est de marquer les noeuds du graphe en sélectionnant à chaque étape le sommet non marqué de plus petite distance partielle. Un label est maintenu pour chaque sommet : à la fin de l'algorithme ce label est égal à sa distance à ***s***. On peut remarquer que l'algorithme ci-dessous suit le schéma de notre algorithme de recherche : à chaque étape le sommet ***y*** sélectionné est en effet adjacent à un sommet déjà marqué, sinon son label serait infini.

`ALGORITHME` Recherche des plus court chemins
`ENTREES` ***G=(V,E)*** un graphe avec une valuation positive ***c*** des arêtes, ***s*** un sommet de ***V***

Initialiser tous les sommets à non marqué ; Marquer ***s***
L(s) := 0 *// Initialise le label de **s** à 0*

`Tant Que` il existe un sommet non marqué
  `Pour` chaque sommet ***y*** non marqué
    Calculer ***L(y)*** := **min{** ***L(x)*** **+** ***c(x,y)*** **|** ***x*** voisin marqué de ***y*** **}**
  `Fin Pour`
  Choisir le sommet ***y*** non marqué de plus petit label ***L***
  Marquer ***y***
`Fin TantQue`

Théoreme

L'algorithme de recherche calcule en temps ***O(\|V\|.\|E\|)*** la longueur des plus courts chemins du sommet ***s*** à tous les autres sommets du graphe

Pour tout ***x***, ***L(x)=D(x)***

Preuve. Nous pouvons remarquer que le label d'un sommet n'est plus modifié par l'algorithme de recherche un fois le sommet marqué. Nous pouvons alors facilement établir par induction sur les étapes de notre algorithme, en utilisant la propriété des distances partielles :

- qu'à chaque étape le label ***L(x)*** d'un sommet marqué est égale à sa distance ***D(x)***,
- qu'à chaque étape le label ***L(y)*** d'un sommet non marqué est égale à sa distance partielle ***DP(S,y)***.

Complexité. L'algorithme comporte ***\|V\|*** étapes correspondant au marquage de chacun des sommets. Pour une étape, il nous faut calculer les labels de tous les sommets non marqués, en scannant chacun de leurs voisins. La propriété sur les degrés montre que cette opération se fait en temps ***O(\|E\|)***, d'où la complexité totale de l'algorithme en ***O(\|V\|.\|E\|)***

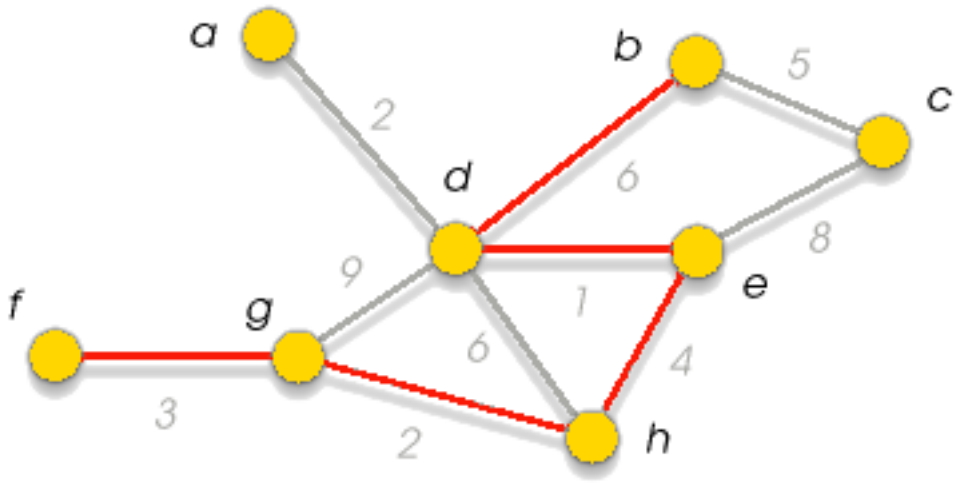
Nous allons généraliser notre notion de plus court chemin dans le cas de graphes valués, où chaque arête ***e*** est associée à une valeur, appelée souvent son poids, ***c(e)***. La valuation des arêtes peut représenter des coûts de transit, des distances kilométriques, le temps nécessaire pour parcourir les arêtes,... Ainsi le graphe ci-contre peut être valué avec les temps de voyage en train entre 2 villes.

Nous sommes alors intéressés par trouver non plus le plus court chemin en nombre d'arêtes (qui correspond sur notre exemple à minimiser le nombre de changement de train) mais le chemin de poids minimum, celui dont la somme des poids des arêtes est le plus faible possible (qui correspond sur notre exemple à minimiser le temps passé dans le train au cours du voyage).

Définition **Plus court chemin**

Dans un graphe valué, le **poids** ***c(p)*** d'un chemin ***p*** est la somme des poids des arêtes le long du chemin.
Dans ce qui suit nous appellerons le poids d'un chemin sa **longueur**. Le **plus court chemin** entre 2 sommets ***s*** et ***t*** est alors défini comme le chemin de plus faible poids reliant ***s*** et ***t***.

Pour différencier la longueur d'un chemin telle que que nous l'avons définie pour un graphe non valué, nous préciserons qu'il s'agit de la longueur en nombre d'arêtes. Cette confusion des 2 définitions de la longueur se justifie par le fait que la longueur en nombre d'arêtes correspond au poids du chemin avec une valuation implice de 1 pour toutes les arêtes.



Dans le graphe valué ci-contre, ***p=(f, g, h, e, d, b)*** a une longueur

de 16. Il est le plus court chemin reliant les sommets ***f*** et ***b***

Nous allons nous intéresser dans un premier temps à des graphes sur lesquels toutes les valuations des arêtes sont positives (nous verrons un peu plus loin ce qui peut se passer lorsque certaines valuations sont négatives). Sous cette condition, les propriétés que nous avons données pour le plus court chemin en nombre d'arêtes restent vérifiées pour le chemin de plus faible poids :

Propriété — Principe de sous-optimalité

Dans un graphe avec une valuation positive ***c()*** de ses arêtes :

- Un plus court chemin entre 2 sommets est élémentaire
- Les plus courts chemins vérifient le principe de **sous-optimalité** : si ***p=(s,...,t)*** est un plus court chemin entre ***s*** et ***t***, alors pour tout sommet ***x*** sur le chemin, le sous-chemin de ***p*** jusqu'à ***x***, ***(s,...,x)***, est un plus court chemin de ***s*** à ***x***
- Si ***D(y)*** est la longueur du plus court chemin d'un sommet ***y*** à ***s***, le principe de sous-optimalité se traduit localement par les égalités :

  D(y)* = 0** si ***y=s
  ***D(y)* = min { *D(x)* + *c(x,y)* | *x voisin de y* }** sinon

HINT
Démonstration identique au plus court chemin en nombre d'arêtes (élémentaire,sous-optimalité)

Pour trouver les plus court chemins dans un graphe valué, nous allons voir 2 algorithmes très différents :

- *Algorithme de Dijkstra*. Cet algorithme est une adaptation de l'algorithme de recherche pour calculer les plus court chemin d'un sommet ***s*** à tous les autres sommets du graphe.
- *Algorithme de Bellman*. Cet algorithme est plus complexe, plus long en temps, mais plus puissant : il détermine la distance de tous les plus courts chemins entre toutes les paires de sommets d'un graphe, pouvant avoir des valuations négatives ou positives de ses arêtes. Son principe repose sur un paradigme algorithmique très général : la programmation dynamique.

Le reste est histoire d'algorithmique : l'algorithme de Dijkstra est une implémentation efficace de l'algorithme de recherche des plus courts chemins. Dans cet algorithme les opérations les plus coûteuses sont le calcul à chaque étape des distances partielles. Cependant une bonne partie de ces opérations sont inutiles : lors du marquage d'un sommet ***x***, *seuls* les voisins de ***x*** peuvent éventuellement voire leur distance partielle modifiée.

Propriété — Mise à jour des distances partielles

Considérons un ensemble ***S*** de sommets incluant la source ***s***, et un sommet ***y***. Pour tout sommet ***z*** en dehors de $S \cup \{y\}$:

HINT
Définition de la distance partielle

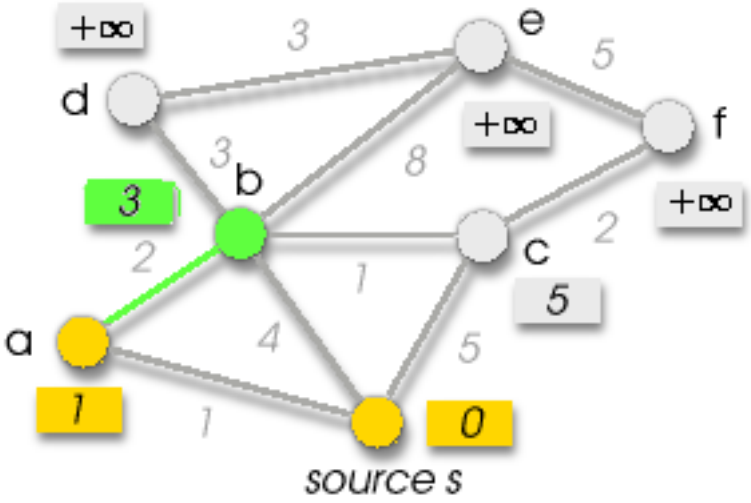
$$DP(S \cup \{y\}, z) = \min\{ DP(S, z), D(y) + c(y,z) \} \quad \text{Si } (y,z) \in E$$

$$DP(S \cup \{y\}, z) = DP(S, z) \quad \text{Sinon.}$$

L'algorithme de Dijkstra utilise cette propriété pour ne mettre à jour à chaque étape que les distances partielles des sommets adjacents au sommet qui vient d'être marqué. Initialement toutes les labels, représentant les distances, sont initialisés à $+\infty$, sauf celui de la source ***s*** mis à **0**. Chaque étape de l'algorithme comporte ensuite 2 phases :

Une phase de sélection
Comme pour l'algorithme de recherche des plus courts chemins, le sommet ***y*** non marqué de plus petit label est sélectionné.

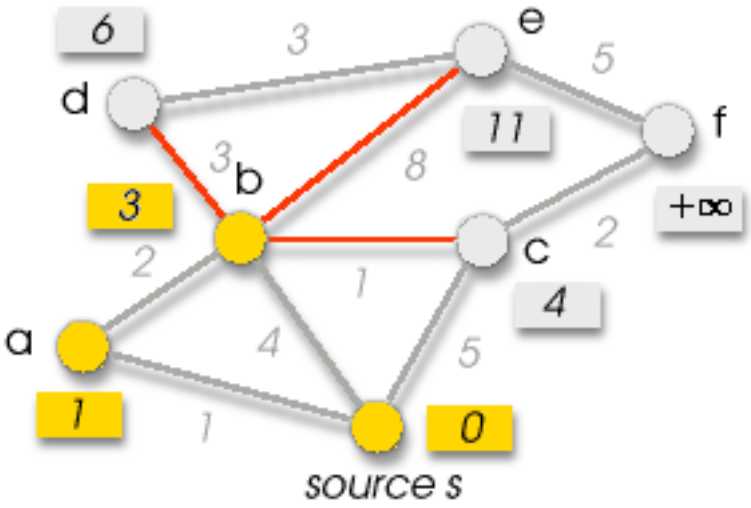
Sur l'exemple le sommet ***b*** de label **3** est le sommet sélectionné à la deuxième étape.



Une phase de mise à jour

Les labels des sommets ***z*** non marqués adjacents au sommet ***y*** qui vient d'être sélectionné sont mis à jour :

L(z) = **min{** ***L(z)***, ***L(y)*** + ***c(y,z)*** **}**



Sur l'exemple les labels des sommets ***c***, ***d*** et ***e*** sont mis à jour.

`ALGORITHME` Dijkstra
`ENTREES` ***G=(V,E)*** un graphe avec une valuation positive ***c*** des arêtes, ***s*** un sommet de ***V***

Initialiser tous les sommets à non marqué ; Initialiser tous les labels ***L*** à $+\infty$
L(s) := 0

`Tant Que` il existe un sommet non marqué
    // Sélection du plus 'proche' sommet non marqué
    Choisir le sommet ***y*** non marqué de plus petit label ***L***
    Marquer ***y***
    // Mise à jour des labels de ses voisins non marqués
    `Pour` chaque sommet ***z*** non marqué voisin de ***y***
        L(z) := **min{** ***L(z)*** **,** ***L(y)*** **+** ***c(y,z)*** **}**
    `Fin Pour`
`Fin TantQue`

Théoreme

L'algorithme de Dijkstra calcule en temps $O(|V|^2)$ la longueur des plus courts chemins du sommet ***s*** à tous les autres sommets du graphe

Pour tout ***x***, ***L(x)=D(x)***

Preuve. En utilisant la propriété de mise à jour des distances partielles, par induction sur les étapes de l'algorithme on établit de manière similaire à l'algorithme de recherche des plus courts chemins on montre facilement :

- qu'à chaque étape le label ***L(x)*** d'un sommet marqué est égale à sa distance ***D(x)***,
- qu'à chaque étape le label ***L(y)*** d'un sommet non marqué est égale à sa distance partielle ***DP(S,y)***.

Complexité. Lorsqu'un sommet ***y*** est marqué, la mise à jours des labels de ses voisins demande un temps ***O(*****d*****(y))***. Un sommet étant marqué une seule fois, le temps total des opérations de mise à jour des labels est en ***O(|E|)*** d'après la propriété des degrés.

La recherche du sommet de plus petit label peut se faire à chaque étape en temps au plus ***O(|V|)*** en parcourant tous les sommets. La complexité de l'algorithme est donc en ***O(|V|.|V| + |E|)***.

L'applet ci-dessous vous permet de faire tourner l'algorithme de Dijkstra sur un exemple.

- *Pour commencer*, sélectionnez une source ***s*** en cliquant sur l'un des sommets du graphe.
- ReLoad. Remet le graphe dans l'état initial.
- Step. Visualise un pas de l'algorithme (sélection du prochain sommet sommet ou mise à jour des sommets adjacents).
- Back. Retourne au pas précédent.
- Play. Déroule l'algorithme en continu (avec une pause de 1s entre chaque pas).

Votre navigateur n'est actuellement pas configuré pour Java

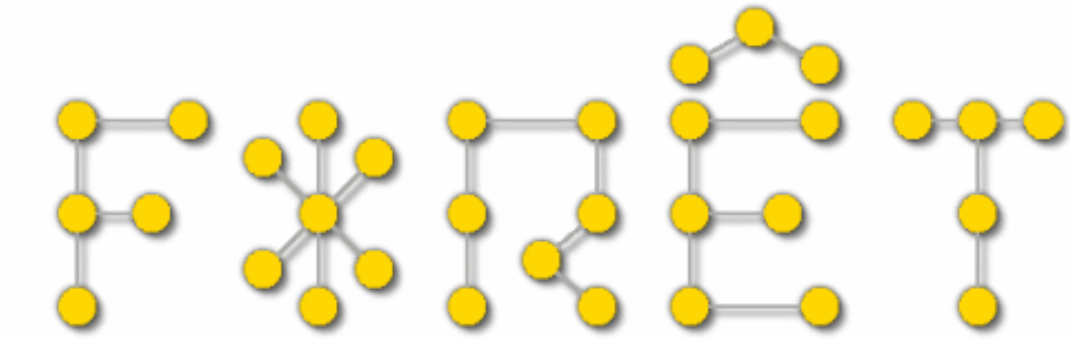
Les arbres sont des graphes particuliers, très populaires en algorithmiques et en informatique.

Définition **Arbre et Forêt**

Un arbre est un graphe connexe sans cycle.
Une forêt est un graphe sans cycle.

Une forêt étant un graphe sans cycle, chacune de ses composantes connexes est acyclique, et par définition connexe. La définition d'une forêt correspond donc bien au sens usuel d'un ensemble d'arbres, chacune de ses composantes connexes étant un arbre.

Les 6 graphes suivants sont des arbres. Pris ensemble ils constituent une forêt, n'en déplaise à Magritte.



Il existe de nombreuses caractérisations pour les arbres que nous détaillons dans la propriété ci-dessous. Si nous regardons la propriété de connexité, les arbres apparaîssent comme des graphes connexes minimaux, au sens où retirer la moindre arête les déconnecte. Si nous regardons les arbres sous l'angle des graphes acycliques, ils sont maximaux, au sens où ajouter la moindre arête crée un cycle.
Il existe également une relation très forte entre le nombre d'arêtes et le nombre de sommets d'un arbre : un arbre $T=(V,E)$ comporte exactement $|V|-1$ arêtes. En effet nous avons vu que :

- Un graphe connexe comporte au moins $|V|-1$ arêtes (cf propriété des graphes connexes).
- Un graphe sans cycle comporte au plus $|V|-1$ arêtes (cf

propriété des graphes acycliques).

Propriété Caractérisations d'un Arbre

Pour un graphe ***T*** d'ordre ***n***, il y a équivalence entre les propriétés :

1. ***T*** est un arbre
2. ***T*** est un graphe connexe à ***n*-1** arêtes
3. ***T*** est connexe, et la suppression de toute arête le déconnecte
4. ***T*** est acyclique à ***n*-1** arêtes
5. ***T*** est acyclique et l'ajout de toute arête le rend cyclique.

Preuve. Pour montrer l'équivalence entre ces 5 propriétés, le plus simple est d'établir une série d'implications.

(1) ⇒ (2) Le graphe ***T*** étant à la fois connexe et acyclique, il possède exactement ***n*-1** arêtes. (cf propriété des graphes connexes et propriété des graphes acycliques).

(2) ⇒ (3) La suppression d'une arête de ***T*** donne un graphe à ***n*-2** arêtes : il ne peut être connexe (cf propriété des graphes connexes).

(3) ⇒ (4) Par l'absurde si ***T*** possèdait un cycle, la suppression d'une arête de ce cycle ne saurait le déconnecter. Par suite ***T*** est acyclique. Puisqu'il est également connexe, il possède donc exactement ***n*-1** arêtes.

(4) ⇒ (5) L'ajout d'une arête à ***T*** donne un graphe à ***n*** arêtes : il ne peut être acyclique (cf propriété des graphes acycliques).

(5) ⇒ (1) Considérons 2 sommets ***x*** et ***y*** de ***T***. Si il existe l'arête ***(x,y)***, alors c'est un chemin de ***x*** à ***y***. Sinon ajoutons cette arête à ***T*** : nous créons alors un cycle, de la forme ***(x, a,..., z, y, x)***. Ceci montre l'existance du chemin ***(x, a,..., z, y)*** entre ***x*** et ***y*** dans ***T***. Par définition ***T*** est donc un graphe connexe.

Souvent, pour manipuler un arbre, nous particularisons un sommet du graphe que nous appelons *racine*. Dans le cas des graphes non orientés, le choix d'une racine ***r*** dans l'arbre est arbitraire. Dans le cas des graphes orientés, la racine est définie de manière unique comme le sommet sans prédécesseur de l'arbre.
Le choix d'une racine revient dans un certain sens à orienter l'arbre, la racine apparaissant comme l'ancêtre commun à la manière d'un arbre généalogique. Le vocabulaire de la théorie des graphes s'en inspire directement : on parle de fils, de père, de frère,...

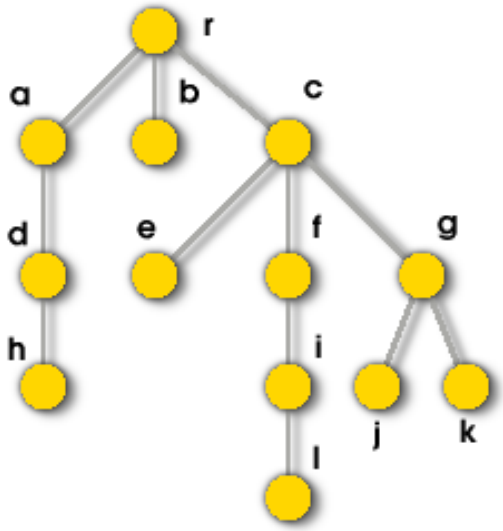
Définition **Feuille et hauteur**

Pour un arbre ***T*** de racine ***r***

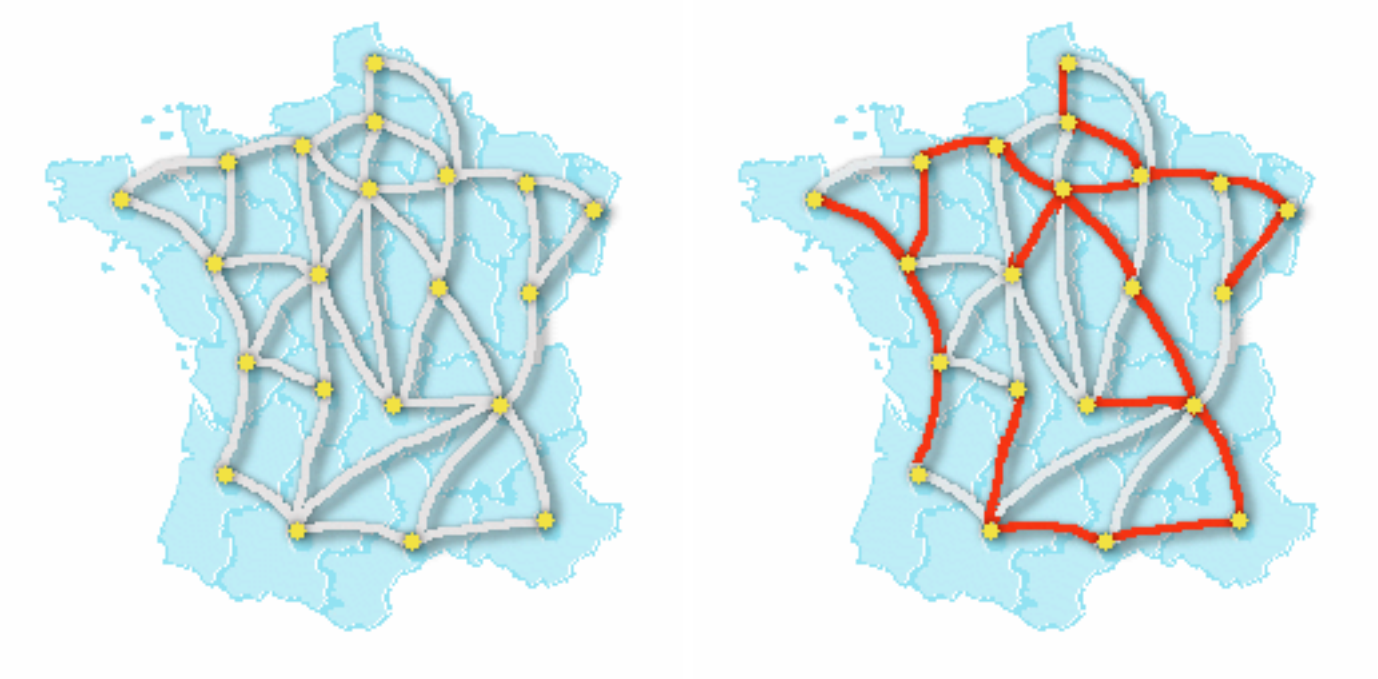
- Le **père** d'un sommet ***x*** est l'unique voisin de ***x*** sur le chemin de la racine à ***x***. La racine ***r*** est le seul sommet sans père.
- Les **fils** d'un sommet ***x*** sont les voisins de ***x*** autres que son père.
- Une **feuille** est un sommet sans fils. Les feuilles correspondent aux sommets de degré 1.
- La **hauteur *h(T)*** de l'arbre ***T*** est la longueur du plus long chemin de la racine à une feuille.

La définition du père repose sur le fait que pour deux sommets ***u*** et ***v*** d'un arbre, il existe un et un seul chemin élémentaire reliant ***u*** et ***v***. L'existence de 2 chemins distincts impliquerait en effet l'existence d'un cycle dans le graphe.

On représente habituellement un graphe de racine ***r*** par *niveaux*, à la façon d' un arbre généalogique. Au niveau 0 apparaît la racine, au niveau 1 ses fils, au niveau 2 les fils de ses fils, etc... Le niveau d'un sommet correspond à sa *profondeur* dans l'arbre, c'est à dire la longueur de son chemin à la racine.
Dans l'exemple l'arbre a une hauteur de 4. Les feuilles de l'arbres sont ***h, b, e, l, j, k***. Les fils de la racine sont ***a, b, c***. Le père de ***g*** est ***c***



Imaginons que nous ayons à connecter des villes entre elles, par exemple avec un nouveau réseau très haut débit. Un certains nombre de connexions directes point à point entre les villes sont techniquement possibles. Il nous faut choisir lesquelles parmi ces connexions nous allons effectivement mettre en place. La distance entre 2 villes dans le réseau final a peu d'importance au vu des débits; cependant les coûts d'installation du réseau ne sont évidemment pas les mêmes pour les différentes liaisons point à point. Nous aimerions donc déterminer comment connecter toutes les villes en minimisant le coût total du réseau.



Ce problème en théorie des graphe correspond à la recherche d'un arbre couvrant de poids minimum. L'ensemble des connexions potentielles peut être représenté par un graphe ***G = (V,E)*** dans lequel chaque arête ***e*** est associée à un coût ***c(e)*** positif. Connecter toutes les villes correspond à sélectionner un ensemble d'arêtes ***F*** de ***G*** tel que le graphe partiel ***H = (V,F)*** induit est connexe. Le poids de cette solution ***c(H)*** est définie comme la somme des poids de ses arêtes. Notre problème s'énonce donc :

*Etant donné un graphe **G = (V,E)**, déterminer un graphe partiel connexe **H = (V,F)** de poids minimum*

Il est facile de voir qu'un tel graphe partiel ***H*** doit être un arbre : si ***H*** n'est pas acyclique, on peut supprimer une arête d'un de ses cycles sans le déconnecter et en faisant diminuer le poids total.

Définition

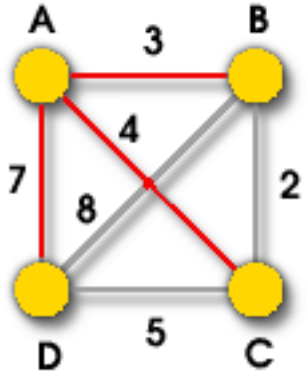
Arbre couvrant

Un arbre couvrant pour un graphe ***G=(V,E)*** est un arbre construit uniquement à partir des arêtes de ***E*** et qui connecte ("couvre") tous les sommets de ***V***. Un arbre couvrant d'un graphe ***G*** est donc un graphe ***T*** tel que

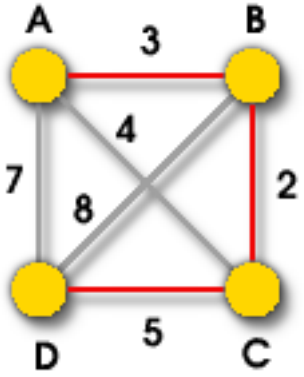
- Le graphe ***T*** est un arbre.
- Le graphe ***T*** est un graphe partiel de ***G***.

Le problème de l'**arbre couvrant de poids minimum** consiste à trouver un arbre couvrant dont la somme des poids ***c(e)*** des arêtes est minimum.

La seule condition, nécessaire et suffisante, pour qu'un graphe admette un arbre couvrant est qu'il soit connexe.
Un arbre couvrant de poids minimum (MST, *Minimum Spanning Tree*) est en général différent de l'arbre des plus courts chemins (SPT, *Shortest Paths Tree*) construit par exemple par BFS ou Dijsktra. Un arbre des plus courts chemins est bien un arbre couvrant, mais il minimise la distance de la racine à chaque sommet, et non la somme des poids des arêtes.



Un arbre des plus courts chemins (SPT) de racine **A**
Sa hauteur est **7**
Son poids est **14**



Un arbre de poids minimum (MST)
Sa hauteur est **10**
Son poids est **10**

Il existe 2 algorithmes célèbres pour résoudre le problème de l'arbre couvrant de poids minimum. Chacun de ces 2 algorithmes utilise plus particulièrement l'une des caractérisations des arbres pour trouver un MST : soit en considérant les arbres comme des graphes connexes avec le mimimum d'arêtes, soit en considérant les arbres comme des graphes acycliques avec le maximum d'arêtes. Ces 2 algorithmes utilisent également 2 techniques de résolution très différentes :

- *Algorithme de Prim* Il maintient au fur et à mesure de la construction un sous-graphe connexe qui grossit petit à petit. Nous allons retrouver une adaptation de notre bon vieil algorithme de recherche dans un graphe.

- *Algorithme de Kruskal* Il maintient au fur et à mesure de la construction un graphe partiel acyclique. Si les algorithmes de recherche sont spécifiques aux graphes, l'algorithme de Kruskal utilise lui un paradigme de résolution plus général : les algorithmes gloutons.

L'algorithme de Prim se base sur la caractérisation des arbres comme des graphes connexes minimaux au sens de l'inclusion : on ne peut enlever une arête à un arbre sans le déconnecter (cf caractérisation des arbres).

L'idée de l'algorithme est de maintenir un sous-graphe partiel connexe, en connectant à nouveau sommet à chaque étape. L'algorithme de Prim va ainsi faire grossir un arbre jusqu'à ce qu'il couvre tous les sommets du graphe. Si à une étape un ensemble ***U*** de sommets sont connectés entre eux, pour choisir le prochain sommet à connecter, l'algorithme part d'une constatation simple : dans un arbre couvrant, il existe nécessairement une arête qui relie l'un des sommets de ***U*** avec un sommet en dehors de ***U***. Pour construire un arbre couvrant de poids minimum (MST), il suffit de choisir parmi ces arêtes *sortantes* celle de poids le plus faible.

Pour détecter les arêtes sortantes, nous pouvons marquer au fur et à mesure de l'algorithme les sommets déjà connectés. Une arête sortante relie alors nécessairement un sommet marqué et un sommet non marqué. L'algorithme de Prim apparaît ainsi comme une adaptation de l'algorithme de recherche pour le problème du MST. Reste une question : de quel sommet partir ? Eh bien le choix du sommet initial n'a pas d'importance... tout sommet doit de toute manière être relié aux autres dans l'arbre final.

`ALGORITHME` Prim
`ENTREES` ***G=(V,E)*** un graphe connexe avec une valuation positive des arêtes
`SORTIE` ***T*** un arbre couvrant de poids minimum

F : ENSEMBLE des arêtes de l'arbre
Initialiser ***F*** à vide
Marquer arbitrairement un sommet

`Tant Que` il existe un sommet non marqué adjacent à un sommet marqué
    Sélectionner un sommet ***y*** non marqué adjacent à un sommet marqué ***x*** tel que ***(x,y)*** est l'arête sortante de plus faible poids
    F := ***F*** ⊔ **{*(x,y)*}**
    Marquer ***y***
`Fin TantQue`
`Retourner` ***T=(V,F)***

Si le graphe ***G*** n'est pas connexe, l'algorithme de Prim sous cette forme construit un arbre couvrant uniquement sur la composante connexe du sommet initialement marqué.

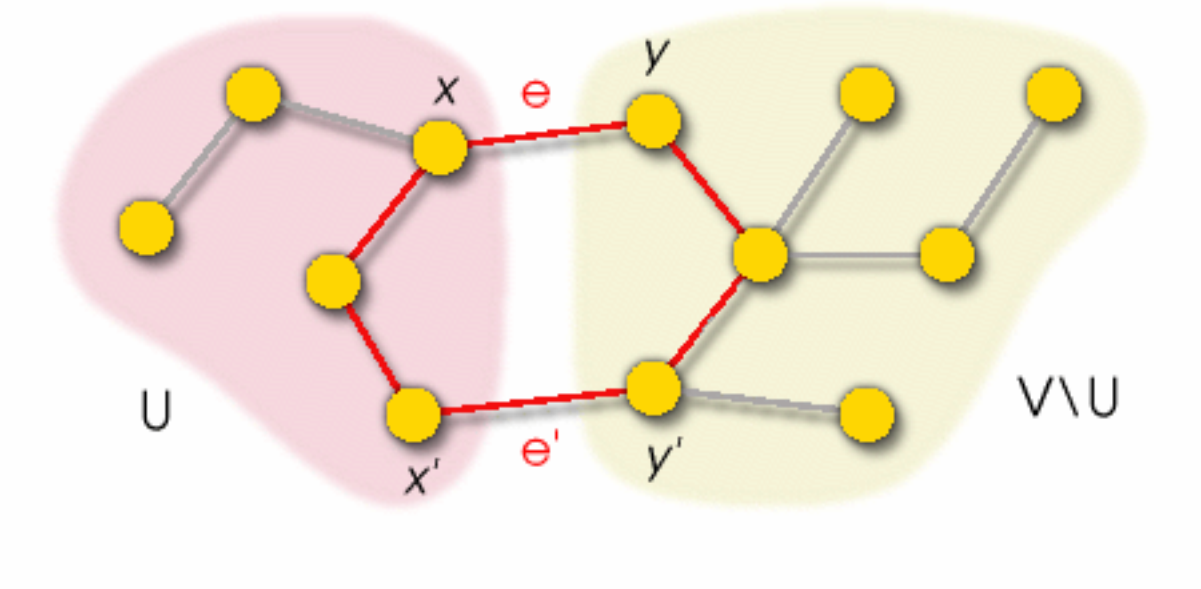
Théoreme

L'algorithme de Prim construit en temps $O(|V|^2)$ un arbre couvrant de poids minimum sur tout graphe connexe $G=(V,E)$

Preuve. Remarquons tout d'abord que l'algorithme de Prim retourne un arbre couvrant. En effet, lorsque l'algorithme s'arrête, il ne reste plus de sommet non marqué adjacent à un sommet marqué. Le graphe ***G*** étant connexe, tous les sommets sont donc marqués. Le marquage d'un sommet allant de paire avec l'ajout de l'une de ses arêtes incidentes le connectant à un sommet déjà marqué, le graphe $T=(V,F)$ est connexe et comporte exactement $|V|-1$ arêtes : c'est un arbre couvrant.

Il nous reste à établir que ***T*** est de poids minimum. Pour cela nous allons montrer par induction qu'à chaque étape, si ***F*** est l'ensemble des arêtes sélectionnées, alors il existe un arbre couvrant de poids minimum comportant les arêtes de ***F***.

- Initialement ***F*** est vide : la propriété est évidemment vérifiée
- Considérons une étape où un ensemble ***U*** de sommets sont marqués et un ensemble ***F*** d'arêtes ont été sélectionnées, et suppososns qu'il existe un arbre couvrant de poids minimum ***T**** comportant les arêtes de ***F***. L'étape suivante consiste au marquage d'un sommet ***y*** et à l'ajout à ***F*** de l'arête $e = (x,y)$. Il nous faut montrer qu'il existe un MST comportant les arêtes de $F' = F \cup \{e\}$.
  Si l'arête ***e*** appartient à ***T****, nous en avons terminé. Sinon ajoutons ***e*** aux arêtes de ***T****. Les caractérisations des arbres montrent que nous créons alors un cycle. Ce cycle parcours à la fois des sommets de ***U*** (parmi eux ***x***) et des sommets en dehors de ***U*** (parmi eux ***y***). Il existe donc nécessairement au moins une arête $e'=(x',y')$ du cycle, différente de ***e***, telle que x' soit marqué et y' soit non marqué. Le graphe T' obtenu en retirant cette arête redevient un arbre couvrant pour ***G***. De plus il est optimal. En effet l'arête e' est une arête sortante pour ***U*** : par construction ***e*** est donc de poids inférieur ou égal.



Complexité. L'algorithme comporte **|*V*|** étapes, correspondant au marquage des sommets. A chaque étape il faut déterminer l'arête sortante de poids minimum. Une implémentation possible consiste à scanner à chaque étape tous les sommets, et pour chaque sommet non marqué, à scanner ses voisins pour déterminer si il est adjacent à un sommet marqué et quelle est son arête incidente sortante de plus faible poids. Cette implémentation conduit à une complexité de l'ordre de la somme des degrés pour chaque étape, soit au total ***O(|V|.|E|)*** d'après la propriété des degrés. Pour améliorer cette complexité, une solution algorithmique analogue à l'algorithme de Dijkstra consiste à mémoriser pour chaque sommet son arête incidente sortante de plus faible poids. Lors du marquage d'un sommet, seuls ses voisins doivent être mis à jour. La complexité pour maintenir cette information pour chaque sommet est donc au total en ***O(|E|)***, qui est dominée par la complexité en ***O(|V|.|V|)*** pour déterminer à chaque étape le sommet à marquer.

La notion d'arête sortante correspond à la définition du cocycle d'un ensemble de sommets. Nous la donnons ici, avec une formulation concise de l'algorithme de Prim sur les graphes connexes, masquant l'algorithme de recherche.

Définition **Cocycle**

Dans un graphe ***G=(V,E)***, le **cocycle** d'un ensemble de sommets ***U*** est l'ensemble des arêtes ***(u,v)*** telles que ***u*** ∈ ***U*** et ***v*** ∈ ***V*** \ ***U***.

ALGORITHME Prim

ENTREES ***G=(V,E)*** un graphe connexe avec une valuation positive des arêtes

SORTIE ***T*** un arbre couvrant de poids minimum

F : ENSEMBLE des arêtes de l'arbre

U : ENSEMBLE des sommets connectés par F

Initialiser ***F*** à vide

Choisir arbitrairement un sommet ***s*** ; Initialiser ***U***:= {***s***}

Tant Que $U \neq V$

    Sélectionner l'arête ***(x,y)*** du cocycle de ***U*** de poids minimum

    $F := F \cup \{(x,y)\}$

    $U := U \cup \{y\}$

Fin TantQue

Retourner ***T=(V,F)***

L'applet ci-dessous vous permet de faire tourner l'algorithme de Prim sur un exemple.
Chaque étape a été décomposée en 2 phases : mise à jour du cocyle et sélection de l'arête à ajouter à l'arbre couvrant.
Les sommets encore non marqués à une étape de l'algorithme apparaisssent en grisé; les arêtes du cocycle des sommets marqués sont affichées en vert, et les arêtes de l'arbre apparaissent en rouge.

Pour commencer	Sélectionnez une source ***s*** en cliquant sur l'un des sommets du graphe.
ReLoad	Remet le graphe dans l'état initial, en tirant aléatoirement des poids pour les arêtes.
Step	Visualise un pas de l'algorithme (mise à jour du cocyle, sélection d'une arête). Les arêtes du cocycle courant sont affichées en vert. Les arêtes sélectionnées pour appartenir à l'arbre couvrant sont représentées en rouge.
Play	Déroule l'algorithme en continu (avec une pause de 1s entre chaque pas).
Stop.	Suspend le déroulement de l'algorithme en mode continu.

Votre navigateur n'est actuellement pas configuré pour Java

L'algorithme de Kruskal se base sur la caractérisation des arbres comme des graphes acycliques maximaux au sens de l'inclusion : on ne peut ajouter une arête à un arbre sans créer de cycle (cf caractérisation des arbres).
L'algorithme de Kruskal va ainsi ajouter au fur et à mesure des arêtes, en s'assurant que le graphe partiel reste à chaque étape une forêt, jusqu'à ce que la forêt devienne un arbre ! L'algorithme de Kruskal construit une forêt couvrante maximale.

Définition **Forêt couvrante maximale**

Une forêt couvrante ***F*** pour un graphe ***G=(V,E)*** est un graphe partiel acyclique.

Une forêt couvrante est **maximale** si l'ajout de toute arête de ***E*** crée un cycle. ***F*** est une forêt couvrante maximale de ***G*** ssi elle vérifie **(1)** ou **(2)** :

1. ***F*** est un arbre couvrant pour chaque composante connexe de ***G***
2. ***F*** comporte exactement $|V|-k$ arêtes, où ***k*** est le nombre de composantes connexes de ***G***

La caractérisation **(2)** d'une forêt couvrante maxmale montre une propriété importante : *toutes les forêts couvrantes maximales ont le même nombre d'arêtes* sur un graphe ***G***.
La caractérisation **(1)** implique bien sûr qu'une forêt couvrante maximale est simplement un arbre couvrant si le graphe ***G*** est connexe.

Initialement la forêt ne comporte aucune arête. A chaque étape l'algorihme de Kruskal ajoute une nouvelle arête à la forêt, en s'assurant que cet ajout laisse le graphe acyclique. L'algorithme s'arrête lorsque plus aucun ajout n'est possible. Le choix de l'arête à ajouter se fait de manière **gloutonne** : on séléctionne l'arête de plus faible poids qui ne crée pas de cycle. Un tel algorithme est **glouton** parce que son choix à une étape correspond au meilleur choix immédiat, sans prendre en compte les conséquences de ce choix sur la suite des décisions. Nous donnons ci dessous une version de l'algorithme de Kruskal sur un graphe général, non nécessairement connexe.

`ALGORITHME` Kruskal
`ENTREES` Graphe ***G=(V,E)***, ***c*** une valuation positive des arêtes
`SORTIE` ***T*** une forêt couvrante maximale de poids minimum

---

e : TABLEAU des arêtes du graphe ***G***
F : ENSEMBLE des arêtes de la forêt
Initialiser ***F*** à vide

Trier les arêtes de ***e*** par poids **c(e[i])** croissant.
`Pour` ***i*** = 1 à |***E*** |
    `Si` ***F*** ∪ **{e[i]}** est acyclique `Alors`
        F := ***F*** ∪ **{e[i]}**
    `Fin Si`
`Fin Pour`
`Retourner` ***T=(V,F)***

Si le graphe ***G*** est connexe, l'algorithme de Kruskal délivre bien sûr un arbre couvrant de poids minimum (MST). Si l'on sait que le graphe ***G*** est connexe en entrée de l'algorithme, on peut modifier la condition d'arrêt dans la construction de ***F***, en s'arrêtant dès que ***F*** comporte |***V*** |**-1** arêtes.

Théoreme

L'algorithme de Kruskal calcule en temps ***O(|E|log |E|)*** une forêt couvrante maximale de poids minimum.

Preuve. Remarquons tout d'abord que l'algorithme de Kruskal délivre une forêt couvrante maximale :

- Le graphe ***T*** retourné par l'algorithme est par construction une forêt couvrante, puisque l'on ajoute des arêtes de ***E*** sans jamais créer de cycle.
- Il est maximal. En effet s'il existait une arête $e \in E$ telle que l'ajout à ***F*** ne crée pas de cycle, nécessairement, pour un indice ***i***, ***e* = e[i]** , et l'algorithme aurait ajouté ***e*** lors de l'étape ***i***.

Il nous reste à établir que ***T*** est de poids minimum.
Notons ***F = { e(1), ..., e(p)}*** les arêtes choisies par l'algorithme de Kruskal, par ordre croissant de leur poids. Nous allons comparer le poids de cette solution au poids de toute autre solution réalisable, pour montrer qu'il est inférieur ou égale. Considérons donc une autre forêt couvrante maximale, définie par les arêtes ***F* = { e*(1), ..., e*(p)}***, également indéxées par poids croissant. Les 2 ensembles ont la même cardinalité, d'après les caractérisations des forêts couvrantes maximales. Nous allons en fait établir une propriété plus forte : le poids de la ième arête ***e(i)*** de la solution de Kruskal est au plus le poids de la ième arête ***e*(i)*** de toute solution.
Supposons par l'absurde que ***e*(i)*** soit de poids strictement inférieur à ***e(i)***. Pour obtenir une contradiction, changeons de graphe ! A la place de ***G*** prenons le graphe partiel ***G'*** contenant uniquement les arêtes de poids inférieur ou égal au coût ***c(e*(i))***. Que se passe-t-il ?

- Sur ***G'*** l'algorithme de Kruskal s'exécute exactement comme sur ***G***, jusqu'à épuisement des arêtes, et retourne comme solution un ensemble d'arêtes inclus dans ***{ e(1), ..., e(i-1)}***.
- L'ensemble d'arêtes ***{ e*(1), ..., e*(i)}*** est une forêt couvrante pour ***G'***.

L'algorithme de Kruskal ne délivre donc pas une forêt couvrante maximale sur ***G'***, puisque nous avons exhibé une autre forêt couvrante comportant strictement plus d'arêtes. Contradiction.

Complexité. La première phase de l'algorithme, le tri des arêtes, se fait en temps ***O(|E|log |E|)***. C'est la phase la plus coûteuse de l'algorithme !
La seconde phase nécessite de pouvoir détecter si l'ajout d'une arête crée un cycle dans le graphe. Pour cela on maintient à jour les composantes connexes de la forêt : l'ajout d'une arête ***e=(x,y)*** crée un cycle **ssi** ***x*** et ***y*** sont dans la même composante. Si l'arête ***e=(x,y)*** est ajoutée, il nous faut fusionner les composantes de ses deux extrémités en une seule. Pour effectuer cette opération de façon efficace, il suffit de modifier les étiquettes des sommets de la plus *petite* des 2 composantes. Avec une structure de donnée appropriée pour les composantes (par exemple une LISTE de ses sommets), les opérations de mises à jour se font au total en temps ***O(|V|log |V|)*** pour tout l'algorithme.

ALGORITHME Kruskal
ENTREES Graphe ***G=(V,E)***, ***c*** une valuation positive des arêtes
SORTIE ***T*** une forêt couvrante maximale de poids minimum

---

e : TABLEAU des arêtes du graphe ***G***
F : ENSEMBLE des arêtes de la forêt
CC : ENSEMBLE des sommets de chaque composante connexe de ***T***

Initialiser ***F*** à vide
Pour chaque sommet ***x*** initialiser ***CC*****[x]** := **{*x*}** FinPour

Trier les arêtes de ***e*** par poids **c(e[i])** croissant.
Pour ***i*** = 1 à |***E*** |
    Si les extrémités ***x*** et ***y*** de **e[i]** sont dans des composantes connexes différentes Alors
        F := ***F*** ∪ **{e[i]}**
        Si |***CC*****[x]**| < |***CC*****[y]**| Alors
            Pour chaque sommet ***u*** de ***CC*****[x]** Faire ***CC*****[y]** := ***CC*****[y]** ∪ **{*u*}**
FinPour
            CC**[x]** := ***CC*****[y]**
        Sinon
            Pour chaque sommet ***u*** de ***CC*****[y]** Faire ***CC*****[x]** := ***CC*****[x]** ∪ **{*u*}**
FinPour
            CC**[y]** := ***CC*****[x]**
        Fin Si
    Fin Si
Fin Pour
Retourner ***T=(V,F)***

L'applet ci-dessous vous permet de faire tourner l'algorithme de Kruskal sur un exemple.

ReLoad	Remet le graphe dans l'état initial, en tirant aléatoirement des poids pour les arêtes.
Step	Visualise un pas de l'algorithme (sélection, ajout ou rejet d'une arête). L'arête courante sélectionnée par l'algorithme est affichée en vert. Si elle est rejetée, elle est affichée en gris clair. Si elle est ajoutée à l'arbre, elle est représentée en rouge.
Play	Déroule l'algorithme en continu (avec une pause de 1s entre chaque pas).
Stop.	Suspend le déroulement de l'algorithme en mode continu.

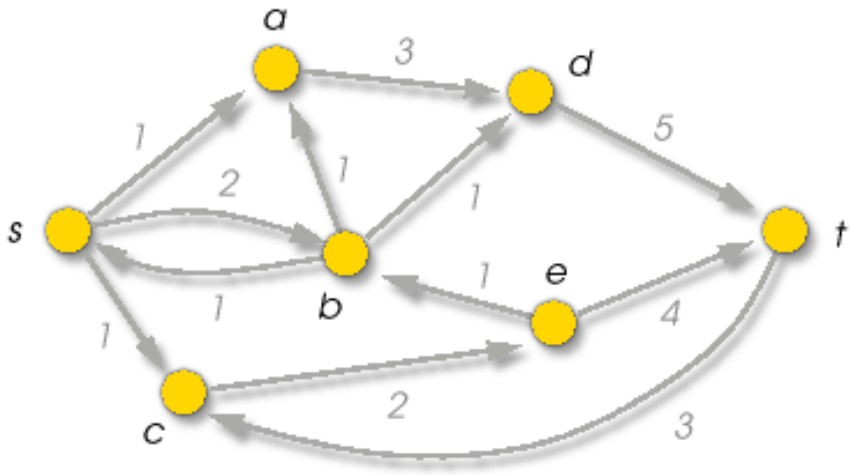
Votre navigateur n'est actuellement pas configuré pour Java

Les flots permettent de modéliser une très large classe de problèmes. Leur interprétation correspond à la circulation de flux physiques sur un réseau : distribution électrique, réseau d'adduction, acheminement de paquets sur Internet, ... Il s'agit d'acheminer la plus grande quantité possible de matière entre une source ***s*** et une destination ***t***. Les liens permettant d'acheminer les flux ont une capacité limitée, et il n'y a ni perte ni création de matière lors de l'acheminement : pour chaque noeud intermédiaire du réseau, le flux entrant (ce qui arrive) doit être égal au flux sortant (ce qui repart).

Définition **Réseau**

Un **réseau** est un graphe orienté ***N=(V,A)*** avec une valuation positive de ses arcs. La valuation ***c(x,y)*** d'un arc ***(x,y)*** est appelée la **capacité** de l'arc.
On distingue sur ***N*** deux sommets particuliers : une source ***s*** et une destination ***t***. Les autres sommets sont les noeuds intermédiaires du réseau.

Le choix d'une source et d'une destination est arbitraire et dépend simplement du problème que nous avons à traiter sur le réseau.



Un exemple de réseau.

Le réseau comporte 5 noeuds intermédiaires. La capacité de l'arc ***(c,e)*** est de **2**, celle de l'arc ***(e,t)*** est de **4**

Remarque

Nous pouvons supposer que tous les arcs ***(x,y)*** entre 2 sommets sont présents dans le réseau. Si un arc est absent, il est en effet possible de le rajouter en lui attribuant une capacité nulle sans changer le problème du flot maximum. Seuls les arcs de capacité non nulle seront représentés sur les exemples.

Un flot représente l'acheminement d'un flux de matière depuis une source ***s*** vers une destination ***t***. Le flot est ainsi décrit par la quantité de matière transitant sur chacun des arcs du réseau. Cette quantité doit être inférieure à la capacité de l'arc, qui limite ainsi le flux pouvant transiter par lui. De plus il n'est pas possible de stocker ou de produire de la matière aux noeud intermédiaires : un flot vérifie localement une loi de conservation analogue aux lois de Kirshoff en électricité.

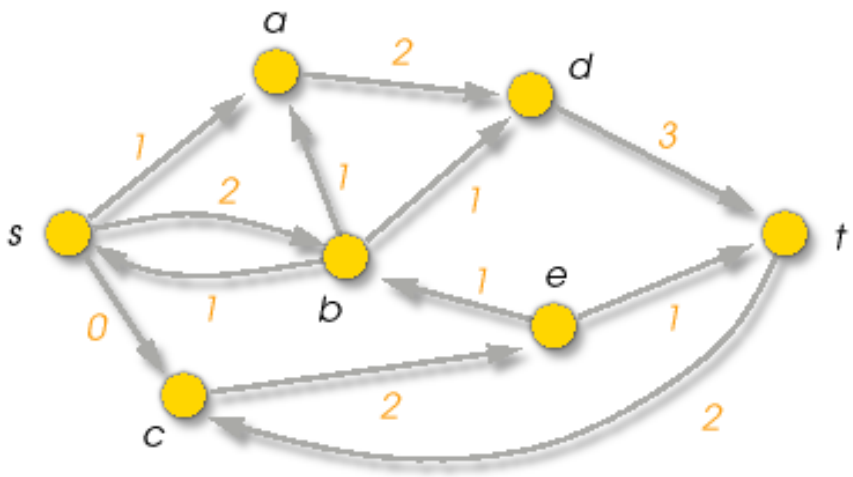
Définition **Flot**

Un **flot** ***F*** sur un réseau ***N=(V,A)*** est une valuation positive des arcs, c'est à dire une application de ***A*** dans ***R+***, qui vérifie les deux propriétés suivantes :

1. Pour tout arc $a \in A$, $0 \leq F(a) \leq c(a)$
2. Pour tout sommet intermédiaire $x \in V \setminus \{ s,t \}$, $\sum_y F(y,x) = \sum_y F(x,y)$

Un exemple de flot sur notre réseau.

Le flot entrant en ***b*** a une valeur de **3**.
La *valeur* du flot est définie comme le



flux net sortant de ***s*** ou entrant dans ***t***.

Sur cet exemple le flot a une valeur de **2**.

Définition **Flot Maximum (MaxFlow)**

La somme $F^-(x) = \sum_y F(y,x)$ est le *flot entrant* au sommet ***x***
La somme $F^+(x) = \sum_y F(x,y)$ est le *flot sortant* du sommet ***x***
La **valeur** $|F|$ d'un flot ***F*** est définie comme le flot sortant moins le flot entrant en ***s*** : $|F| = F^+(s) - F^-(s)$

Le problème du **Flot Maximum** consiste à trouver un flot ***Fmax*** de valeur maximale sur le réseau ***N***.

La valeur du flot correspond au flux net partant de ***s***. La valeur du flot peut être définie de manière équivalente comme le flux net arrivant à ***t***, c'est à dire le flot entrant moins le flot sortant de ***t***. En effet pour tout noeud intermédiaire ***x***, le flot entrant étant égal au flot sortant, on a :

$$\sum_{x \neq s,t} F^-(x) = \sum_{x \neq s,t} F^+(x)$$

ce qui se réécrit : $\sum_{x \neq s,t} (F(s,x) + F(t,x) + \sum_{y \neq s,t} F(y,x)) = \sum_{x \neq s,t} (F(x,s) + F(x,t) + \sum_{y \neq s,t} F(x,y))$

Le dernier terme de part et d'autre de l'égalité est le même, donc on a $F^+(s) + F^+(t) = F^-(s) + F^-(t)$, le flux net sortant de ***s*** est effectivement égale au flux net entrant en ***t***, ce qui correspond bien à notre problème d'acheminement de flux de ***s*** à ***t***.

Comment aborder un problème comme la recherche d'un flot maximum sur un réseau ? Nous allons utiliser le paradigme très général de l'*optimisation locale*. Son principe consiste à se donner une solution réalisable, c'est à dire un flot sur le réseau, et à essayer de l'améliorer à chaque itération. L'algorithme s'arrête lorsqu'il ne parvient plus à trouver un flot meilleur (de plus grande valeur) que le flot courant.

Tout cela semble frappé au coin du bon sens, mais l'amélioration que nous recherchons à chaque étape doit être rapide à calculer : il ne s'agit pas de rechercher parmi toutes les solutions possibles si l'une d'elles est meilleure que notre solution courante, mais de faire cette recherche sur un petit nombre de solutions proches de la solution courante, sur un *voisinage* de celle-ci. L'algorithme conduit à une solution localement optimale, c'est à dire meilleure que toutes les autres solutions de son voisinage,... mais qui n'est pas nécessairement un optimum global. L'exemple le plus fameux d'algorithme d'optimisation locale est le Simplex, que vous verrez en recherche opérationnelle, et qui permet de résoudre la programmation linéaire.

`ALGORITHME` Optimisation Locale
`ENTREES` Réseau ***N=(V,A)***, ***s, t*** des sommets de ***V***
`SORTIE` ***F*** un flot entre ***s*** et ***t***

Construire un flot ***F*** sur le réseau entre ***s*** et ***t***

`Tant Que` il existe une amélioration locale de ***F***
    Améliorer ***F***
`Fin TantQue`

`Retourner` ***F***

Trouver un flot initiale n'est pas très compliqué : le flot nulle ***F* = 0**, qui ne fait rien transiter sur les arcs est toujours un flot possible (il respecte la loie de conservation et les capacités). Notre problème est de trouver un moyen simple pour améliorer la valeur d'un flot ***F*** qui ne serait pas maximum. Imaginons que nous puissions faire passer un peu plus de matière de ***s*** à ***t***. Si nous suivons le parcours dans le réseau d'une quantité élémentaire de matière supplémentaire, elle emprunte un chemin orienté allant de ***s*** à ***t***, dont aucun des arcs n'est *saturé*

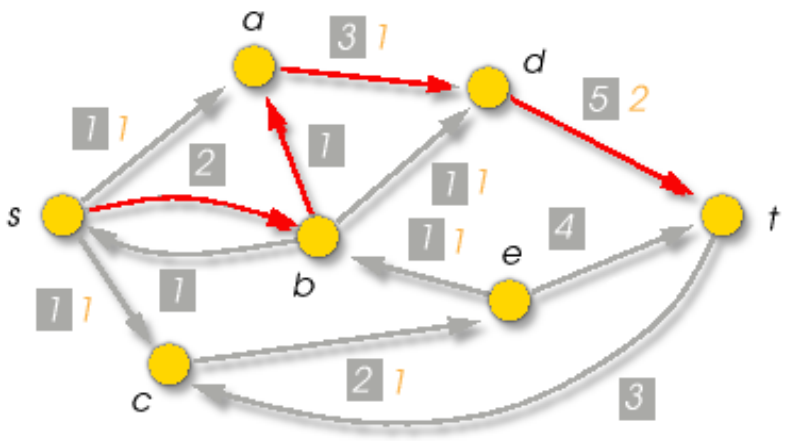
Définition **Saturation**

Un arc ***(x,y)*** est **saturé** par un flot ***F*** si la valeur du flot sur l'arc égale sa capacité : ***F(x,y) = c(x,y)***. Un chemin est saturé si l'un de ses arcs est saturé.

La **capacité résiduelle** d'un arc ***(x,y)*** est la quantité ***c(x,y) - F(x,y)*** de flot pouvant encore transiter par lui. La capacité résiduelle d'un chemin est la plus petite capacité résiduelle de ses arcs.

Saturer un chemin ***p*** de ***s*** à ***t*** consiste à augmenter le flot de ses arcs de la capacité résiduelle du chemin.

Lorsque l'on sature un chemin entre ***s*** et ***t*** on conserve un flot sur le réseau : les capacités sont évidemment respectées, et la loie de conservation reste vraie, le chemin repartant autant de fois qu'il arrive de tout noeud intermédiaire. La valeur du flot augmente alors de la capacité résiduelle du chemin.



Les capacités du réseau sont indiqués dans les carrés grisés sur les arcs. Un flot de valeur 2 est décrit par les valuations en orange (les flots nuls sont implicites).
La capacité résiduelle de l'arc ***(d,t)*** est 3, celle de l'arc ***(c,e)*** est de 1. L'arc ***(b,d)*** a une capacité résiduelle nulle : il est saturé.

Le chemin ***(s, c, e, t)*** est saturé. Le chemin ***(s, b, a, d, t)*** a une capacité résiduelle de 1. Saturer ce chemin fait augmenter la valeur du flot de 1.

Bien sûr, si il existe des chemins non saturés entre ***s*** et ***t***, le flot ***F*** n'est pas maximum : il suffit de saturer l'un de ces chemins pour obtenir un flot de valeur supérieure ! L'idée la plus naturelle pour écrire un algorithme d'optimisation locale consiste alors à saturer itérativement tous les chemins entre ***s*** et ***t*** :

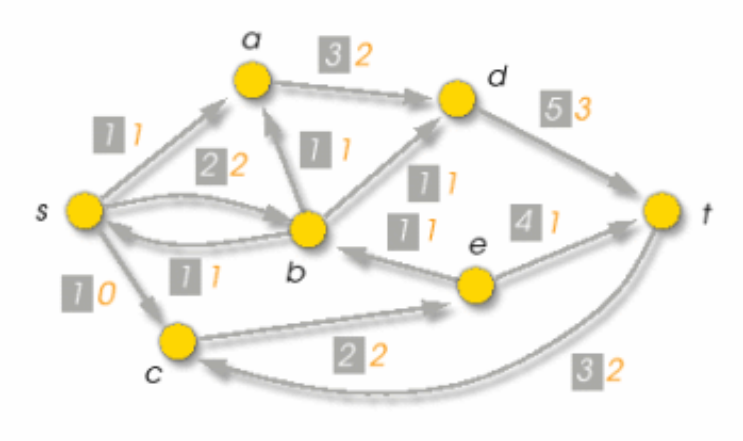
`ALGORITHME` Saturation
`ENTREES` Réseau ***N=(V,A)***, ***s, t*** des sommets de ***V***
`SORTIE` ***F*** un flot entre ***s*** et ***t***

Initialiser ***F*** **:= 0** *// On part d'un flot possible entre **s** et **t***

`Tant Que` il existe un chemin ***p*** de ***s*** à ***t*** non saturé par le flot ***F***
    Augmenter le flot ***F*** en saturant ***p***
`Fin TantQue`

`Retourner` ***F***

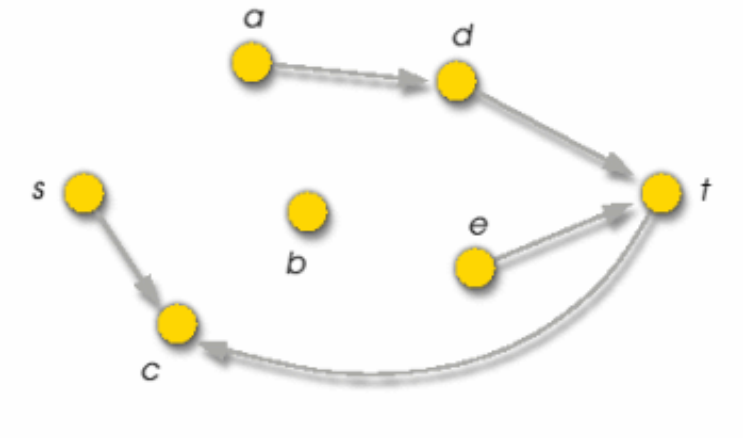
Malheureusement... L'algorithme Saturation ne marche pas très bien. Il délivre bien un flot, mais pas toujours le flot maximum. Pourquoi ? Si effectivement un flot dans lequel il existe un chemin non saturé entre ***s*** et ***t*** n'est pas maximum, la récirpoque n'est pas vraie : ce n'est pas parce que tous les chemins de ***s*** à ***t*** sont saturés que le flot est maximum ! Cela peut sembler étrange de prime abord : si les chemins sont saturés, comment pourrait-on acheminer un flot supplémentaire de ***s*** à ***t*** ? Apparemment toutes les routes possibles sont bloquées, et pourtant...

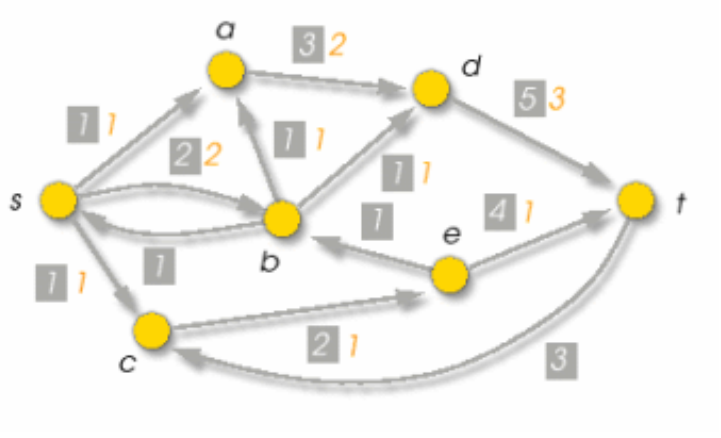


Sur le réseau ***N*** qui nous tient lieu d'exemple depuis le début, considérons le flot ***F***ci-contre défini par les valuations oranges des arcs, à côté des capacitées indiquées dans les carrés grisés.

Dans ce flot tous les chemins de ***s*** à ***t*** sont saturés. Pour s'en convaincre, on peut dessiner le réseau partiel (représenté sous notre réseau ***N***) comprenant uniquement les arcs de ***N*** non saturés par le flot ***F*** (c'est à dire de capacité résiduelle non nulle). Il existe alors un chemin de ***s*** à ***t*** non saturé par ***F*** dans le réseau ***N*** simplement si il existe un chemin dans le réseau partiel : ce qui visiblement n'est pas le cas.

Pourtant le flot ***F*** n'est pas maximum. Sa valeur est 2, alors qu'il existe un flot de valeur 4 représenté ci-dessous.





Les chemins non saturés ne constituent pas le bon voisinage pour trouver un flot maximum par une optimisation locale. Nous avons besoin de la notion de *chemin augmentant* définie à partir du *réseau résiduel*

Les flots permettent de modéliser une très large classe de problèmes. Leur interprétation correspond à la circulation de flux physiques sur un réseau : distribution électrique, réseau d'adduction, acheminement de paquets sur Internet, ... Il s'agit d'acheminer la plus grande quantité possible de matière entre une source ***s*** et une destination ***t***. Les liens permettant d'acheminer les flux ont une capacité limitée, et il n'y a ni perte ni création de matière lors de l'acheminement : pour chaque noeud intermédiaire du réseau, le flux entrant (ce qui arrive) doit être égal au flux sortant (ce qui repart).

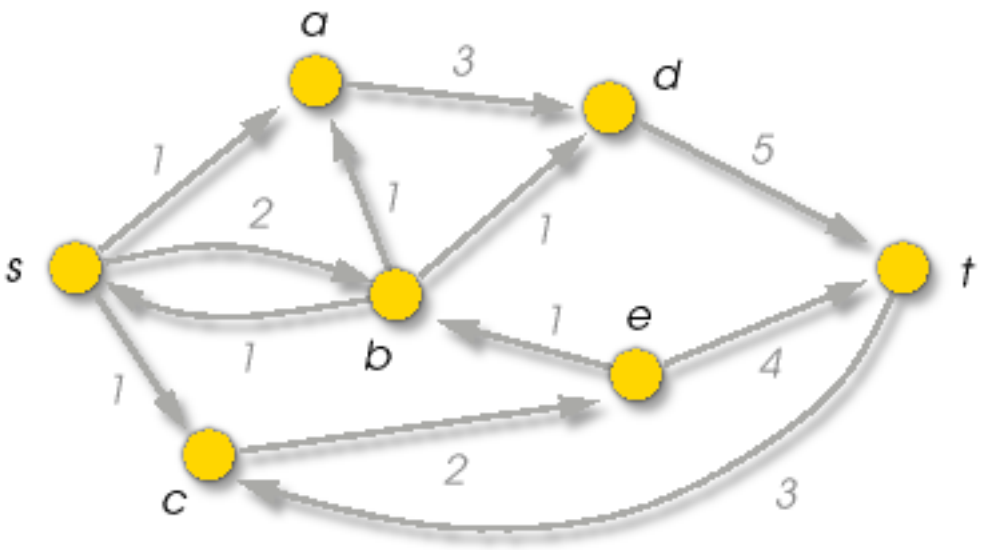
Définition **Réseau**

Un **réseau** est un graphe orienté ***N=(V,A)*** avec une valuation positive de ses arcs. La valuation ***c(x,y)*** d'un arc ***(x,y)*** est appelée la **capacité** de l'arc.
On distingue sur ***N*** deux sommets particuliers : une source ***s*** et une destination ***t***. Les autres sommets sont les noeuds intermédiaires du réseau.

Le choix d'une source et d'une destination est arbitraire et dépend simplement du problème que nous avons à traiter sur le réseau.

Un exemple de réseau.

Le réseau comporte 5



noeuds intermédiaires. La capacité de l'arc ***(c,e)*** est de **2**, celle de l'arc ***(e,t)*** est de **4**

Remarque

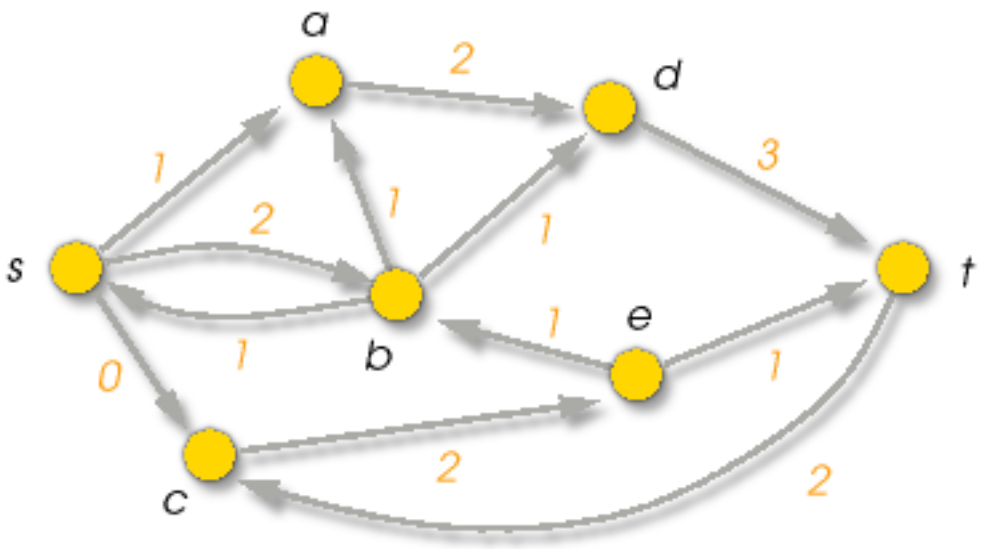
Nous pouvons supposer que tous les arcs ***(x,y)*** entre 2 sommets sont présents dans le réseau. Si un arc est absent, il est en effet possible de le rajouter en lui attribuant une capacité nulle sans changer le problème du flot maximum. Seuls les arcs de capacité non nulle seront représentés sur les exemples.

Un flot représente l'acheminement d'un flux de matière depuis une source ***s*** vers une destination ***t***. Le flot est ainsi décrit par la quantité de matière transitant sur chacun des arcs du réseau. Cette quantité doit être inférieure à la capacité de l'arc, qui limite ainsi le flux pouvant transiter par lui. De plus il n'est pas possible de stocker ou de produire de la matière aux noeud intermédiaires : un flot vérifie localement une loi de conservation analogue aux lois de Kirshoff en électricité.

Définition **Flot**

Un **flot** ***F*** sur un réseau ***N=(V,A)*** est une valuation positive des arcs, c'est à dire une application de ***A*** dans ***R+***, qui vérifie les deux propriétés suivantes :

1. Pour tout arc $a \in A$, $0 \leq F(a) \leq c(a)$
2. Pour tout sommet intermédiaire $x \in V \setminus \{ s,t \}$, $\sum_y F(y,x) = \sum_y F(x,y)$



Un exemple de flot sur notre réseau.

Le flot entrant en ***b*** a une valeur de **3**.
La *valeur* du flot est définie comme le flux net sortant de ***s*** ou entrant dans ***t***.

Sur cet exemple le flot a une valeur de **2**.

Définition — Flot Maximum (MaxFlow)

La somme $F^-(x) = \sum_y F(y,x)$ est le *flot entrant* au sommet ***x***
La somme $F^+(x) = \sum_y F(x,y)$ est le *flot sortant* du sommet ***x***
La **valeur** $|F|$ d'un flot ***F*** est définie comme le flot sortant moins le flot entrant en ***s*** : $|F| = F^+(s) - F^-(s)$

Le problème du **Flot Maximum** consiste à trouver un flot ***Fmax*** de valeur maximale sur le réseau ***N***.

La valeur du flot correspond au flux net partant de ***s***. La valeur du flot peut être définie de manière équivalente comme le flux net arrivant à ***t***, c'est à dire le flot entrant moins le flot sortant de ***t***. En effet pour tout noeud intermédiaire ***x***, le flot entrant étant égal au flot sortant, on a :

$$\sum_{x \neq s,t} F^-(x) = \sum_{x \neq s,t} F^+(x)$$

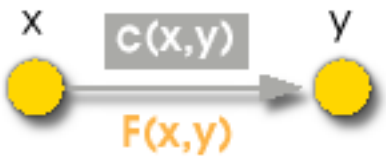
ce qui se réécrit : $\sum_{x \neq s,t} (F(s,x) + F(t,x) + \sum_{y \neq s,t} F(y,x)) = \sum_{x \neq s,t} (F(x,s) + F(x,t) + \sum_{y \neq s,t} F(x,y))$

Le dernier terme de part et d'autre de l'égalité est le même, donc on a $F^+(s) + F^+(t) = F^-(s) + F^-(t)$, le flux net sortant de ***s*** est effectivement égale au flux net entrant en ***t***, ce qui correspond bien à notre problème d'acheminement de flux de ***s*** à ***t***.

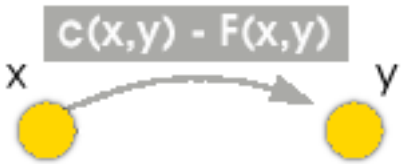
Notre premier essai d'optimisation locale, l'algorithme Saturation, avait en fait tout d'un algorithme glouton : la saturation d'un chemin n'est jamais remise en cause, le flot sur un arc ne peut qu'augmenter au cours de l'algorithme.

Pour éviter l'approche gloutonne, il nous faut être capable non seulement d'augmenter, mais aussi de diminuer la valeur du flot sur un arc ***(x,y)***. Une solution serait de faire transiter un flot négatif sur ***(x,y)***. Cependant nous avons défini un flot comme une valuation positive des arcs, correspondant à un transit de matière. Alors nous allons sortir un lapin du chapeau : pour faire diminuer par exemple de **1** le flot sur l'arc ***(x,y)***, nous faisons passer un flot de **1** dans l'autre sens, sur l'arc ***(y,x)***. En terme de bilan de matière, si un flot de **3** transite sur ***(x,y)*** et un flot de **1** sur ***(y,x)***, cela revient en effet à n'acheminer qu'un flot de **2** de ***x*** à ***y***.

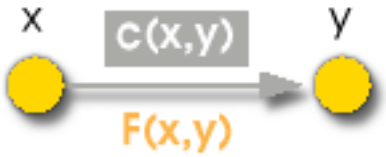
Cependant l'arc ***(y,x)*** peut ne pas exister,... ou être déjà saturé par le flot. Aussi allons nous changer de réseau : nous associons à un flot ***F*** sur un réseau ***N*** le *réseau résiduel* N_F composé d'arcs *forward* (avant) et *backward* (arrière) .



A un arc ***(x,y)*** du réseau ***N*** est associé dans le réseau résiduel l'arc *forward* noté $(x,y)_F$ de capacité ***c(x,y) - F(x,y)***



La capacité de l'arc forward traduit que l'on peut encore augmenter le flot ***F*** sur ***(x,y)***, d'au plus ***c(x,y) - F(x,y)*** qui correspond à la saturation de l'arc. Un arc forward "existe" (il est de capacité non nulle) donc dans N_F ssi il n'est pas saturé dans ***N***



A un arc ***(x,y)*** du réseau ***N*** est associé dans le réseau résiduel l'arc *backward* noté $(y,x)_B$ de capacité ***F(x,y)***

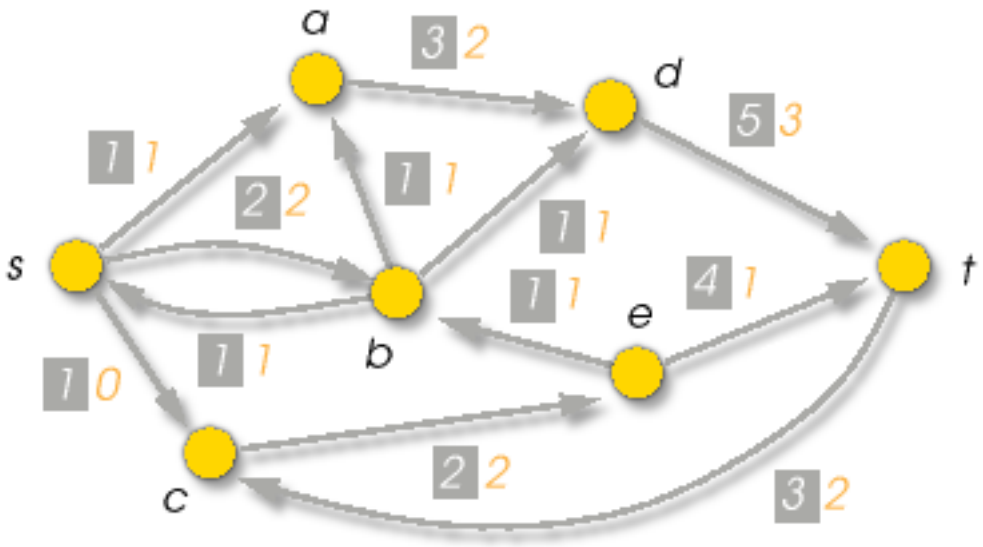
La capacité de l'arc backward traduit que l'on peut diminuer la valeur du flot ***F*** allant de ***x*** à ***y***, d'au plus ***F(x,y)*** qui correspond à annuler le flot. Un arc backward "existe" (il est de capacité non nulle) donc dans N_F ssi le flot ***F*** n'est pas nulle sur l'arc ***(x,y)***.

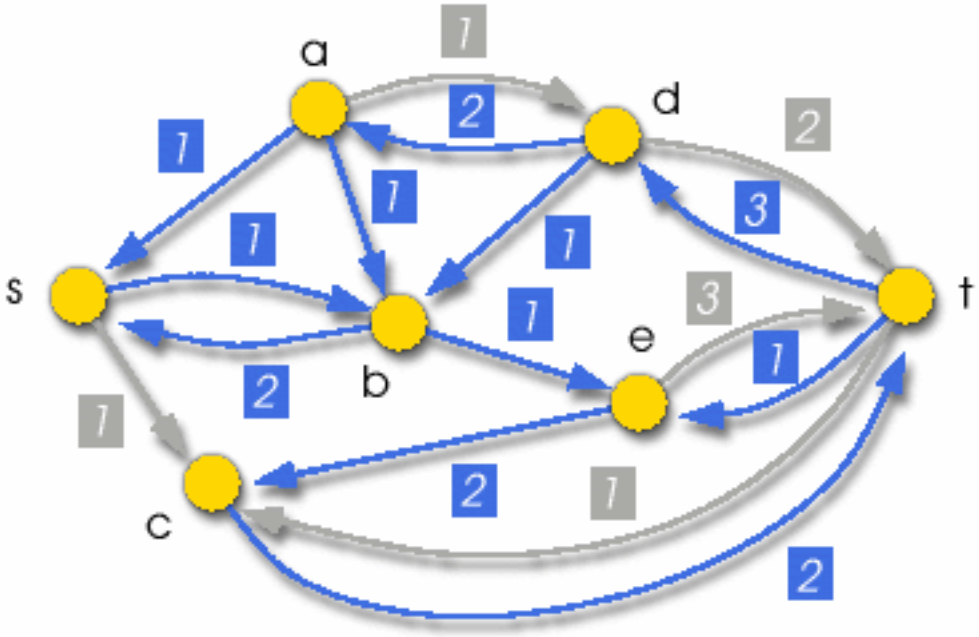
Définition **Réseau résiduel**

Le réseau résiduel N_F associé à un flot ***F*** sur un réseau ***N*** est un réseau comprenant les mêmes sommets que ***N***. A chaque arc ***(x,y)*** de ***N*** est associé dans N_F :

- l'arc forward $(x,y)_F$ de capacité ***c(x,y) - F(x,y)***
- l'arc backward $(y,x)_B$ de capacité ***F(x,y)***

Reprenons l'exemple du flot ***F*** sur le réseau ***N***. Les capacités sont représentées dans les carrés à coté des arcs; les valeurs du flot apparaissent en orange.





Le réseau résiduel N_F associé à ***F*** est représenté ci contre. Les arcs forward apparaissent en gris, les arcs backward sont en bleu. Les arcs de capacité nulle n'ont pas été représentés.

Dans le réseau ***N***, l'arc ***(s,a)*** est saturé : seul apparaît dans le réseau résiduel son arc backward ***(a,s)***, de capacité **1** (capacité de l'arc d'origine).
Dans le réseau ***N***, aucun flot ne passe par l'arc ***(s,c)*** : seul apparaît dans le réseau résiduel l'arc forward ***(s,c)***, de capacité **1** (capacité de l'arc d'origine)
Enfin prenons l'exemple de l'arc ***(d,t)***. Sa capacité est de **5** dans le réseau ***N*** et un flot de **3** y circule. Dans le réseau résiduel lui sont alors associés :

- Un arc forward ***(d,t)*** de capacité **2** qui correspond à la quantité de flot pouvant encore transiter par cet arc
- Un arc backward ***(t,d)*** de capacité **3** qui correspond à la quantité de flot pouvant être diminuée sur cet arc.

Maintenant que nous avons un nouveau réseau, N_F, il est bien sûr tentant d'essayer d'y faire transiter un flot. Quel est alors le rapport entre un flot ***f*** sur le réseau résiduel N_F et le flot ***F*** sur le réseau ***N*** ? A partir de ces 2 flots nous allons construire un nouveau flot ***F'*** sur ***N*** :

- En ajoutant à ***F*** la valeur du flot ***f*** sur les arcs forward
- En retranchant à ***F*** la valeur du flot ***f*** sur les arcs backward

La valeur de ce nouveau flot ***F'*** est alors égale à la valeur de ***F*** plus celle de ***f*** : nous avons fait augmenter la valeur du flot transitant sur le réseau ***N***

Définition **Ajout d'un flot résiduel**

Si ***f*** est un flot sur le réseau résiduel N_F, nous définissons la valuation des arcs ***F'*** = ***F*** $\oplus$ ***f*** sur le réseau ***N*** par :

$$F'(x,y) = F(x,y) + f_F(x,y) - f_B(y,x)$$

Alors :

- ***F'*** est un flot sur le réseau ***N***
- La valeur du flot ***F'*** est la somme des 2 flots : | ***F'*** | = | ***F*** | + | ***f*** |

Preuve. La valuation ***F'*** est un flot sur le réseau ***N*** si elle est positive et respecte les capacités, et si la loi de conservation est vérifiée à tous les noeuds internes. Il est facile de se convaincre de cette dernière condition, les flots ***F*** et ***f*** la vérifiant tous les deux. Pour le respect des capacités, nous avons :

$\mathbf{0} \leq f(x,y)_F \leq c(x,y) - F(x,y)$ et $\mathbf{0} \leq f(y,x)_B \leq F(x,y)$, ce qui implique $\mathbf{0} \leq F'(x,y) \leq c(x,y)$

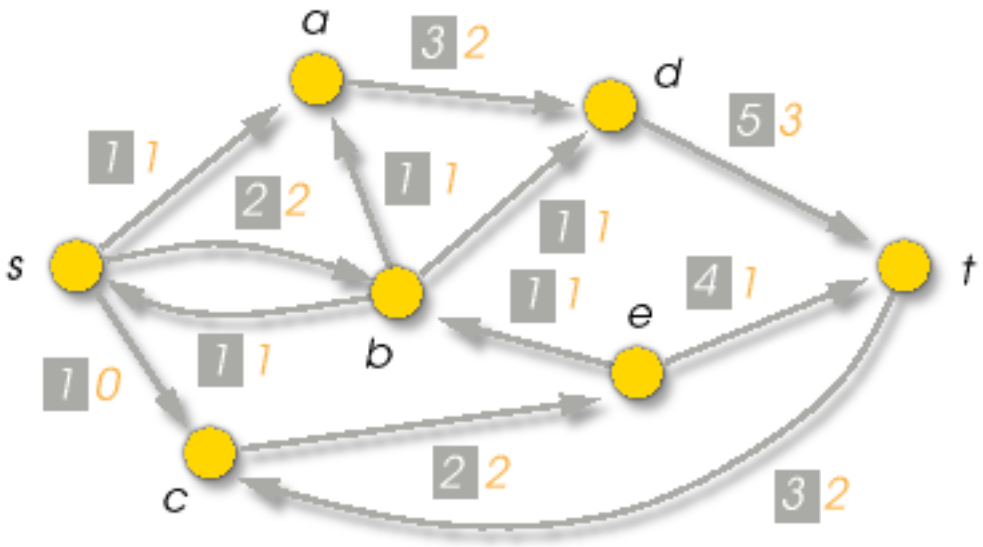
Si nous calculons la valeur du flot ***F'***, nous avons :

$$|F'| = \sum_x (F(s,x) + f(s,x)_F - f(x,s)_B) - \sum_x (F(x,s) + f(x,s)_F - f(s,x)_B)$$

$$|F'| = \sum_x F(s,x) - F(x,s) + \sum_x f(s,x)_F + f(s,x)_B - \sum_x f(x,s)_F + f(x,s)_B$$

On a donc bien $|F'| = |F| + |f|$

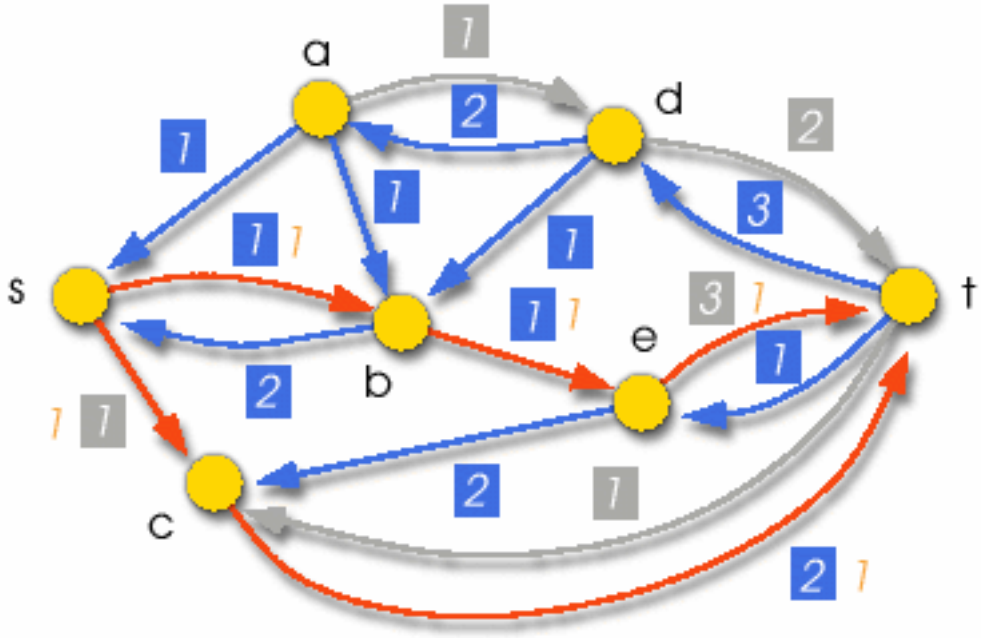
Exemple



Reprenons l'exemple du flot ***F*** sur le réseau ***N***. La valeur du flot ***F*** est **2**.

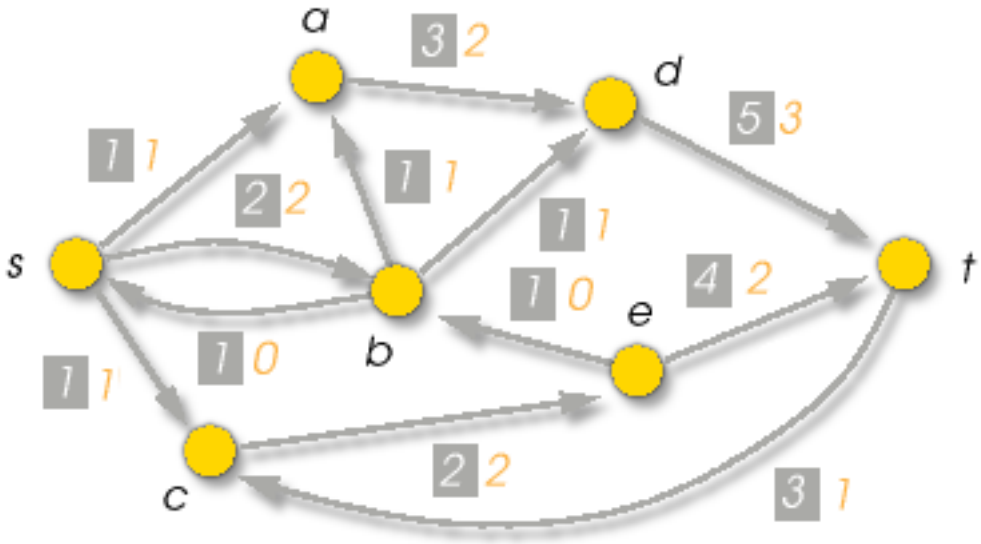
Nous pouvons définir sur le réseau résiduel N_F le flot ***f*** ci-contre. Sa valeur est **2**.

Il emprunte les arcs forward



(s,c) et ***(e,t)***, et les arcs backward ***(s,b)***, ***(b,e)*** et ***(c,t)***. Sur cet exemple la valuation du flot ***f*** sur tous ces arcs est de **1**.

Le flot résultant ***F*** ⊕ ***f*** sur le réseau ***N*** est représenté ci-contre.



Les arcs ***(s,c)*** et ***(e,t)*** ont vu leur flot augmenté de **1**, le flot ***f*** empruntant leur arc forward. A l'inverse les arcs ***(b,s)***, ***(e,b)*** et ***(t,c)*** ont vu leur flot diminuer de **1**, le flot ***f*** empruntant leur arc backward.

La valeur du flot $F \oplus f$ est bien de **4**.

Trouver un flot sur le réseau résiduel permet donc de faire augmenter, en l'ajoutant, le flot ***F*** sur le réseau ***N***. Comme nous l'avions vu dans l'algorithme Saturation, la manière la plus simple de rechercher un flot est de regarder si il existe un chemin (orienté) de capacité non nulle reliant ***s*** à ***t***. Il est alors possible de faire passer un flot non nulle en saturant le chemin. La notion de *chemin augmentant* reprend cette idée dans le réseau résiduel.

Définition **Chemin augmentant**

Un chemin augmentant pour un flot ***F*** sur un réseau ***N*** est un chemin (orienté) de capacité non nulle reliant ***s*** à ***t*** dans le réseau résiduel N_F.

Avec la convention de ne considérer que les arcs de capacité non nulle, un chemin augmentant ***p*** est simplement un chemin orienté de ***s*** à ***t*** dans le réseau résiduel N_F. Il est "augmentant" puisque le flot résultant $F \oplus p$ sur le réseau ***N*** augmente de la capacité du chemin ***p***. Lorsque nous écrivons $F \oplus p$, nous entendons implicitement pour ***p*** le flot correspondant à la saturation du chemin.

Dans l'exemple précédent le réseau résiduel admettait comme chemin augmentant ***(s,b,e,t)*** et ***(s,c,t)***, mais aussi ***(s,b,e,c,t)***, ***(s,c,t,d,b,e,t)***, etc, tous de capacité **1**.

Les chemins augmentants sont le "bon voisinage" pour appliquer avec succès le paradigme de l'optimisation locale. L'approche reprend la structure de l'algorithme Saturation, mais pour faire augmenter le flot ***F***, nous ne saturons plus des chemins (orienté) de ***s*** à ***t*** directement dans le réseau ***N*** mais dans le réseau résiduel N_F.

L'algorithme consiste alors à partir d'un flot ***F*** réalisable (le flot nullle fait parfaitement l'affaire) et à l'améliorer itérativement. Pour cela à chaque étape l'algorihtme vérifie si il existe un chemin augmentant pour le flot, c'est à dire un chemin (orienté) de ***s*** à ***t*** dans le réseau résiduel. Si un tel chemin existe, il est saturé dans N_F et le flot correspondant est "ajouté" à ***F***. Sinon, il n'existe plus de chemin augmentant et l'algorithme retourne le flot courant.
Le principe de cet algorithme est du à *Ford & Fulkerson* dans les années 60.

`ALGORITHME` Ford & Fulkerson
`ENTREES` Réseau ***N=(V,A)***, ***s, t*** des sommets de ***V***
`SORTIE` ***F*** un flot maximum entre ***s*** et ***t***

Initialiser ***F*** **:= 0** *// On part d'un flot possible entre **s** et **t***

`Tant Que` il existe un chemin augmentant ***p*** de ***s*** à ***t***
    Saturer le chemin ***p*** dans le réseau résiduel N_F
    F **:=** ***F*** $\oplus$ ***p***
`Fin TantQue`

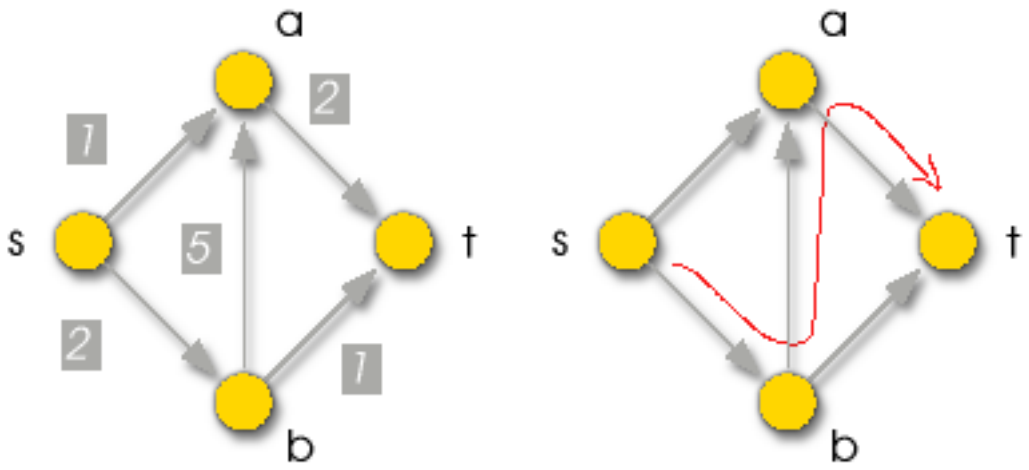
`Retourner` ***F***

Vérifier si il existe un chemin augmentant n'est pas très difficile. Il suffit de vérifier si il existe un chemin (orienté) de ***s*** à ***t*** dans le réseau résiduel. Ceci peut se faire en temps $O(|A|)$ par un algorithme de recherche de plus court chemin comme BFS.

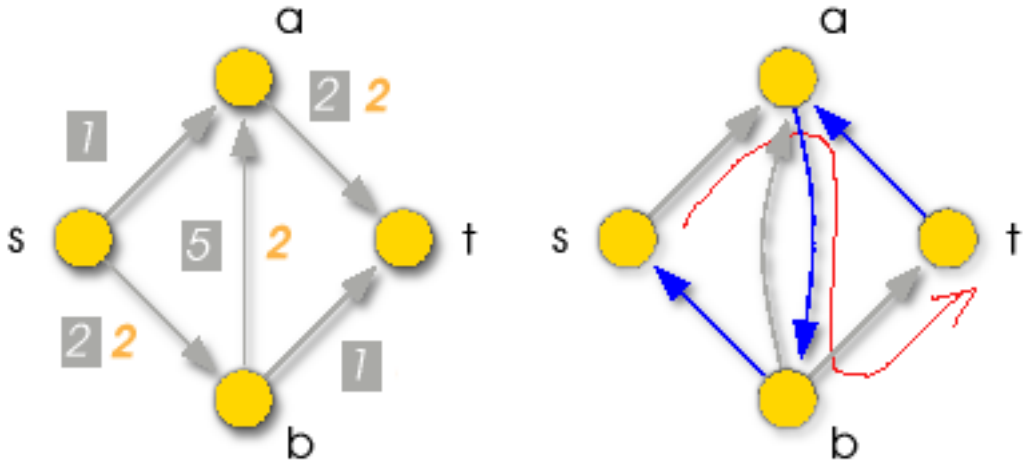
Théoreme

L'algorithme de Ford & Fulkerson délivre un flot maximum

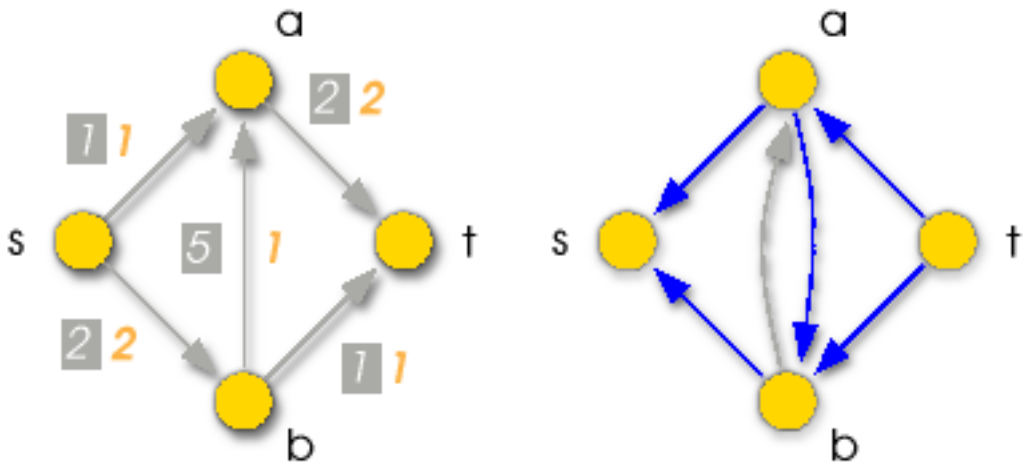
Déroulons l'algorithme sur un petit exemple où il est facile de voir que le flot maximum a une valeur de **3**



Le réseau apparaît à gauche. Initialement un flot nulle transite sur les arcs. Son réseau résiduel représenté à droite est alors bien sûr identique au réseau de départ. Un chemin augmentant possible est ***(s,b,a,t)***, de capacité **2**.



La saturation du chemin passe le flot ***F*** à **2**. On peut remarquer que l'algorithme Saturation s'arrêterait à cette étape : il n'y a plus de chemin non saturé de ***s*** à ***t*** dans le réseau. Cependant il existe un chemin augmentant ***(s,a,b,t)*** dans le réseau résiduel.



La saturation du chemin augmentant fait passer le flot ***F*** à **3**. Il n'existe plus de chemin de ***s*** à ***t*** dans le réseau résiduel. L'algorithme de Ford & Fulkerson s'arrête et délivre le flot ***F***, qui est optimal.

Nous avons ici choisi arbitrairement les chemins augmentants. En théorie l'algorithme converge en moins d'étapes vers le flot maximum en choisissant le plus court (en terme de nombre d'arcs) chemin augmentant.

Le théorème que nous avons énoncé affirme que l'algorithme de Ford & Fulkerson délivre un flot maximum. Mais... comment montrer un tel résultat? La démonstration de l'optimalité de l'algorithme semble bien plus difficile que pour Dijkstra ou Prim par exemple. Ces algorithmes étaient constructifs et prenaient au fur et à mesure les bonnes décisions sur des solutions partielles pour aboutir au final à l'optimum. Rien de tel pour l'algorithme de Ford et Fulkerson qui par principe part d'une solution réalisable et l'améliore au fur et à mesure des itérations...

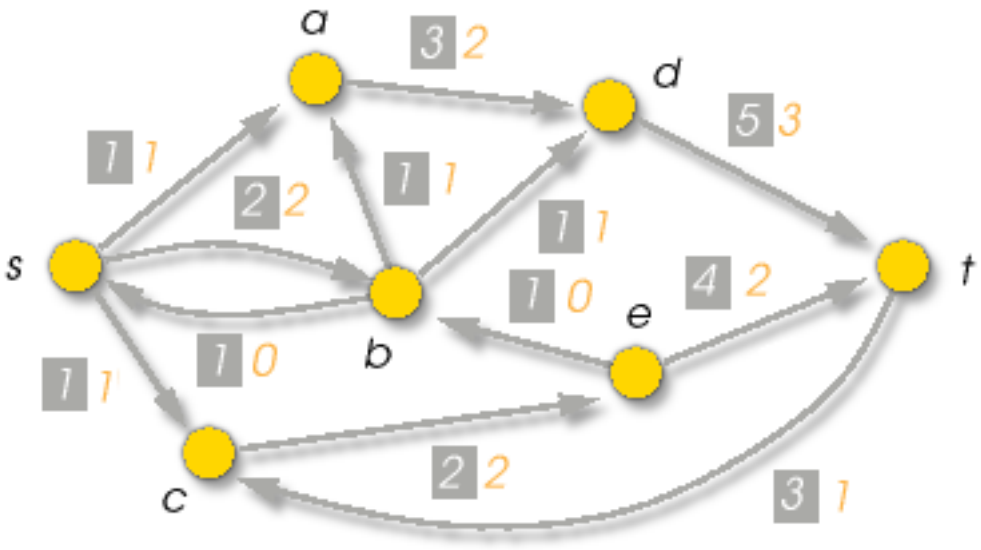
Pour prouver l'optimalité de l'algorithme, nous allons avoir recours à une notion **fondamentale** en optimisation : celle de *borne supérieure*.

Définition **Borne Supérieure**

Un problème d'optimisation décrit un ensemble d'instances, les contraintes à satisfaire pour qu'une solution soit réalisable et une *fonction objectif* à maximiser ou à minimiser.
L'optimum **OPT*(I)*** est la meilleure valeur possible de la fonction objectif d'une solution réalisable pour une instance ***I*** du problème.

Une *borne supérieure* est une application à valeur réelle qui à chaque instance ***I*** associe une valeur **BS*(I)*** supérieure à **OPT*(I)***.

Quel est l'intérêt d'une borne supérieure pour un problème de maximisation comme le flot maximum ? Elle sert justement à apprécier si nous sommes proches ou loin de l'optimum. Dans le cas idéal si nous trouvons une solution dont la valeur est égale à une borne supérieure... c'est que notre solution est optimale !



Prenons l'exemple du flot que nous avions trouvé précédemment, de valeur **4**. La question se pose de savoir si ce flot est maximum ou non.

Cependant la valeur d'un flot est égale au flot sortant de ***s*** moins le flot entrant. Il est clair que cette valeur ne peut pas être supérieure à la somme des capacités des arcs sortants de ***s***, puisqu'un flot doit respecter la capacité de chaque arc.

Sur l'exemple aucun flot ne peut donc avoir une valeur supérieure à la somme des capacités des

arcs ***(s,a)***, ***(s,b)*** et ***(s,c)***, soit **4**. Cette valeur est une borne supérieure pour notre instance.

Le flot ***F*** atteignant la borne supérieure, il est optimal.

Nous avons une première borne supérieure pour le flot maximum : la somme des capacités des arcs sortants de ***s***. Mais cette borne supérieure n'est pas assez précise. Si par exemple la capacité de l'arc ***(s,b)*** n'était pas **2** mais **4**, le flot ***F*** serait-il toujours optimal ? Oui... mais nous ne pouvons plus le prouver avec notre borne supérieure!

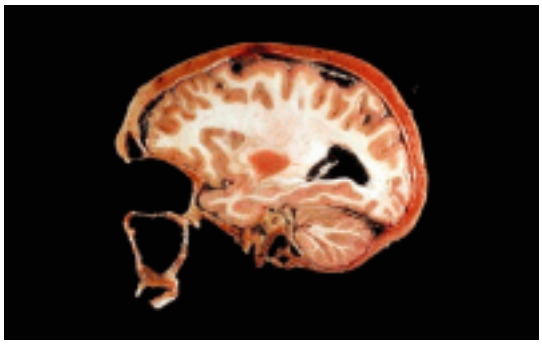
L'ensemble des arcs sortants de ***s*** est en fait un cas particulier d'une *coupe* dans le réseau. La notion de coupe va nous fournir une borne supérieure toujours atteinte par le flot maximum : c'est ce que nous dira le théorème *MaxFlow-MinCut*.

Une coupe dans un réseau est un ensemble d'arcs déconnectant la source du puits, c'est à dire qu'il n'existe plus de chemin orienté de s à t.
Une coupe peut également être vue comme une partition $S \sqcup T$ des sommets où s appartient à S et t appartient à T.

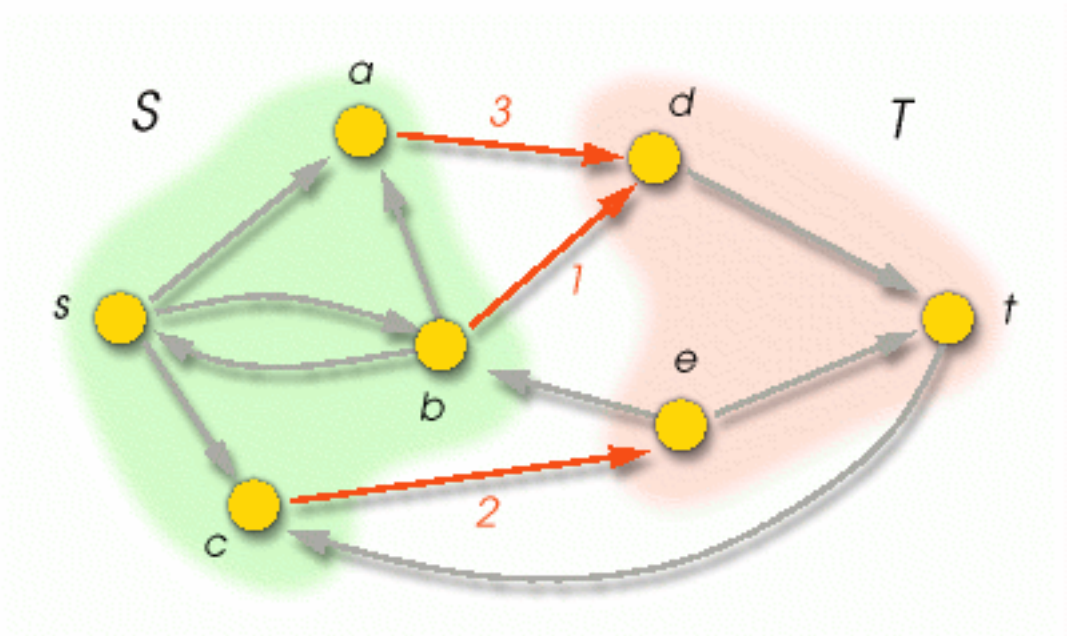
Définition **Coupe**

Si s et t sont 2 sommets d'un réseau $N = (V,A)$,

- Une **(s,t)-coupe** est un ensemble C d'arcs déconnectant s de t : dans le graphe partiel $(V, A \setminus C)$ il n'existe pas de chemin orienté de s à t.
- Une **(s,t)-coupe** se définit également par une partition $C = S \sqcup T$ des sommets telle que $s \in S$ et $t \in T$. Les arcs de la coupe sont alors les arcs (x,y) ayant leur origine x dans S et leur destination y dans T
- La *capacité* d'une coupe est la somme des capacités des arcs de la coupe.



Nous manipulerons dans la suite une (s,t)-coupe (nous dirons simplement une coupe) comme une partition $S \sqcup T$. Il est clair que si nous enlevons tous les arcs (x,y) ayant leur origine x dans S et leur destination y dans T, il ne peut plus exister de chemin orienté allant de s à t.



Reprenons l'exemple de notre réseau.

La coupe ci-contre est définie par la partition ***S*={ *s, a, b, c* }** et ***T*={ *d, e, t* }**. Elle comporte les arcs ***(a,d) (b,d)*** et ***(c,e)***. Sa capacité est de **6**,

Le problème qui va nous intéresser est de déterminer parmi toutes les coupes d'un réseau celle de capacité minimale.

Définition **Coupe Minimum (MinCut)**

Le problème de la **Coupe Minimum** consiste à trouver une coupe ***Cmin*** entre ***s*** et ***t*** de capacité minimale.

Une coupe est un passage obligé pour un flot, et potentiellement un goulot d'étranglement. En effet un flot transitant de ***s*** à ***t*** doit nécessairement emprunter les arêtes de la coupe (tous chemins de ***s*** à ***t*** en comporte au moins une par définition). La valeur du flot peut ainsi être redéfinie comme le flot sortant de la coupe moins le flot entrant.

Propriété Valeur d'un flot

Si $C = S \sqcup T$ est une coupe et F un flot sur le réseau N, alors la valeur du flot est égale au flot sortant de S moins le flot entrant de S.

$$|F| = F^+(S) - F^-(S)$$

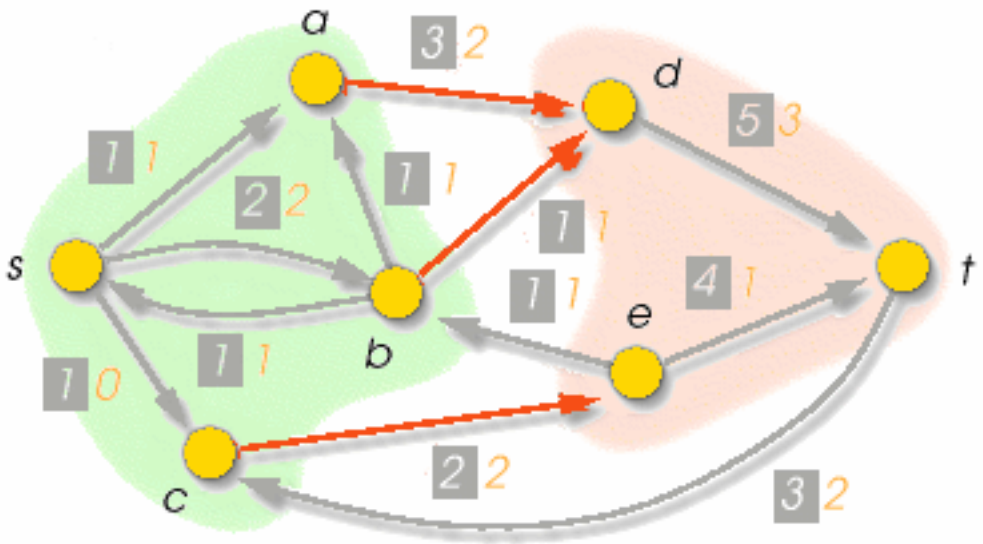
avec $F^+(S) = \sum_{x,y} F(x,y)$, $x \in S$ et $y \in T$ et $F^-(S) = \sum_{x,y} F(y,x)$, $x \in S$ et $y \in T$

Preuve. La valeur d'un flot F est définie par $|F| = F^+(s) - F^-(s)$
Or pour tout sommet intermédiaire x, le flot entrant est égale au flot sortant. Nous pouvons donc réécrire l'égalité précédente: $|F| = \sum_{x \in S} F^+(x) - \sum_{x \in S} F^-(x)$
Il suffit alors de décomposer chaque flot entrant et sortant en distinguant les somets de S et de T:

$$|F| = \sum_{x \in S} \left(\sum_{y \in S} F(x,y) + \sum_{y \in T} F(x,y) \right) - \sum_{x \in S} \left(\sum_{y \in S} F(y,x) + \sum_{y \in T} F(y,x) \right)$$

$$|F| = \sum_{x \in S} \sum_{y \in T} F(x,y) - \sum_{x \in S} \sum_{y \in T} F(y,x)$$

$$|F| = F^+(S) - F^-(S)$$



La valeur du flot F est de **2**

Le flot sortant de la coupe $F^+(S)$ est égale à **5**: **2** sur l'arc ***(a,d)***, **1** sur l'arc ***(b,d)*** et **2** sur l'arc ***(c,e)***.

Le flot entrant de la coupe $F^-(S)$ est égale à **3**: **2** sur l'arc ***(t,c)***, **1** sur l'arc ***(e,b)***

On a bien $|F| = F^+(S) - F^-(S)$

La définition de la valeur comme le flot sortant moins le flot entrant en ***s*** est un cas particulier de la propriété, avec la coupe ***{s}*** $\sqcup$ ***V \ {s}***. De même dire que la valeur du flot est égale au flot entrant moins le flot sortant en ***t*** correspond au cas particulier de la coupe ***V \ {t}*** $\sqcup$ ***{t}***
Une conséquence de cette propriété est que la valeur de tout flot entre ***s*** et ***t*** est toujours inférieure à la capacité de toute ***(s,t)***-coupe. La capacité minimale d'une coupe est donc une borne supérieure de la valeur maximum d'un flot.

Propriété **MaxFlow MinCut (forme faible)**

Si ***F*** est un flot sur un réseau entre ***s*** et ***t*** et ***C*** est une ***(s,t)***-coupe, alors: $|F| \leq |C|$
Ce qui peut également s'énoncer par rapport au flot maximum et à la coupe minimum:

$$|Fmax| \leq |Cmin|$$

La capacité d'une coupe constitue une borne supérieure pour la valeur maximale d'un flot. Le théorème *MaxFlow-MinCut* énonce que cette borne supérieure est toujours atteinte: la valeur maximale d'un flot est toujours égale à la capacité minimale d'une coupe. Ce résultat sera réinterprété dans le cadre générale de la *dualité* en programmation linéaire.

MaxFlow-MinCut

La valeur maximale d'un flot entre 2 sommets ***s*** et ***t*** dans un réseau est égale à la capacité minimale d'une ***(s,t)***-coupe.

$$| \boldsymbol{Fmax} | = | \boldsymbol{Cmin} |$$

Et un théorème de plus à prouver! Mais nous allons faire d'une pierre deux coups: pour établir le théorème MaxFlow-MinCut et l'optimalité de l'algorithme de Ford & Fulkerson, nous allons énoncer un résultat plus général qui nous montrera que chercher le flot maximum ou la coupe minimum sur un réseau sont 2 problèmes équivalents.

Si ***F*** est un flot entre 2 sommets ***s*** et ***t*** dans un réseau ***N***, il t a équivalence entre les propriétés:

1. Le flot ***F*** est maximum
2. Il n'existe pas de chemin augmentant pour ***F***
3. Il existe une ***(s,t)***-coupe ***C*** de capacité égale à la valeur du flot ***F***

Preuve. Pour montrer l'équivalence entre ces 3 propriétés, nous allons établir la suite des implications.

(1) $\Rightarrow$ (2) Si il existait un chemin augmentant ***p***, alors $F \oplus p$ serait un flot de valeur $|F| + |p|$, ce qui contradit que ***F*** est un flot maximum.

(2) $\Rightarrow$ (3) Considérons l'ensemble ***S*** des sommets ***x*** tels qu' il existe un chemin orienté de***s*** à ***x*** dans le réseau résiduel $\mathcal{N}_F$, et ***T*** son complémentaire.
La partition $C = S \cup T$ définit alors une ***(s,t)***-coupe sur le réseau, puisque par définition $s \in S$ et que $t \in T$ du fait qu'il n'existe pas de chemin augmentant. Nous allons montrer que la capacité de cette coupe est égale à la valeur du flot ***F***. Rappelons que la valeur du flot est égale à

$$|F| = F^+(S) - F^-(S)$$

Il nous faut donc établir que:

- Tous les arcs de la coupe, c'est à dire les arcs sortants de ***S*** sont saturés par le flot ***F***
- Tous les arcs entrants de ***S*** ne voient aucun flot transiter sur eux.

Considérons un arc ***(x,y)***, $x \in S$ et $y \in T$. Supposons par l'absurde que cet arc ne soit pas saturé. Alors dans le graphe résiduel l'arc forward $(x,y)_F$ a une capacité non nul. Or il existe un chemin de ***s*** à ***x***. En prolongeant ce chemin par l'arc $(x,y)_F$, on obtient alors un chemin de ***s*** au sommet ***y***. Ce qui contredit que $y \in T$.

Considérons un arc ***(y,x)*** avec $y \in T$ et $x \in S$. Supposons également par l'absurde que le flot ***F*** soit non nul sur cet arc. Alors il existe dans le réseau résiduel l'arc backward $(y,x)_R$ de capacité non nulle. De même ceci contredit que le sommet ***y*** n'appartienne pas à ***S***
Nous avons donc:

$$|F| = F^+(S) - F^-(S)$$
$$|F| = \sum_{x \in S} \sum_{y \in T} c(x,y) - 0 = |C|$$

(3) $\Rightarrow$ (1) Conséquence de la propriété MaxFlow MinCut (forme faible) que nous avons démontrée.

GLOSSAIRE

A

Adjacence
deux sommets sont adjacents si ils sont reliés par une arête

Arbre
graphe connexe sans cycle

Arbre couvrant
graphe partiel qui est un arbre

Arc Backward
Arc retour associé dans le réseau résiduel à un arc non nulle

Arc Forward
Arc associé dans le réseau résiduel à un arc non saturé

B

BFS (*Breath First Search*)
algorithme de recherche en largeur (calcul des plus court chemin)

C

Chemin
suite de sommets telle deux sommets successifs sont adjacents

Chemin Eulérien
chemin passant une et une seule fois par chaque arête du graphe

Chemin augmentant
chemin (orienté) reliant la source au puits dans le réseau résiduel

Chemin simple
une arête du graphe est empruntée au plus une fois par le chemin

Chemin élémentaire
un sommet du graphe est visité au plus une fois par le chemin

Clique
sous-graphe complet

Cocycle (d'un ensemble ***U***)
*arêtes ayant une extrémité dans **U** et une extrémité en dehors de **U***

Composante connexe
sous-graphe connexe maximal

Connexe (graphe)
il existe un chemin entre chaque paire de sommets

Coupe (Graphe orienté)
Une (s,t)-coupe est un ensemble d'arc déconnectant le sommet s du sommet t.

Cycle
chemin simple finissant au sommet de départ

Cycle Eulérien
cycle passant une et une seule fois par chaque arête du graphe

D

Degré
nombre d'arêtes incidentes à un sommet

Dijkstra
algorithme des plus courts chemins dans un graphe valué positif

E

Euler (algorithme)
algorithme construisant un cycle eulérien

F

Feuille
sommet de degré 1 dans un arbre

Fils
voisins d'un sommet autres que son père dans un arbre

Flot
valuation positive des arcs d'un réseau, respectant les capacités et la loi de Kirshoff en chacun des noeuds intermédiaires

Flot Maximum
flot de valeur maximale

Ford & Fulkerson
algorithme de chemins augmentants pour la recherche d'un flot maximum sur un réseau

Forêt
graphe dont les composantes connexes sont des arbres

Forêt couvrante maximale
graphe partiel qui est un arbre couvrant pour chaque composante connexe

G

Graphe
ensemble fini de sommets et d'arêtes

Graphe Complet
graphe où chaque sommet est relié à tous les autres.

Graphe Eulérien
graphe admettant un cycle Eulérien

Graphe partiel
graphe contenant tous les sommets mais seulement une partie des arêtes d'un autre graphe

H

Hauteur
longueur du plus long chemin de la racine d'un arbre à une feuille

I

Incidence (arête)
une arête est incidente aux sommets qui sont ses extrémités

K

Kruskal
algorithme pour l'arbre couvrant de poids minimum

L

Longueur (chemin valué)
somme des poids des arêtes le long du chemin

Longueur (chemin)
nombre d'arêtes sur le chemin

P

Prim
algorithme pour l'arbre couvrant de poids minimum

Père
voisin d'un sommet sur son chemin à la racine d'un arbre

R

Recherche (algorithme de)
Algorithme d'exploration d'un graphe

Réseau
graphe orienté avec une valuation positive (capacité) sur ses arcs

Réseau résiduel
Réseau associé à un flot F sur un réseau N, composé des arcs forward et bacward

S

Saturation (chemin dans un réseau)
un chemin dans un réseau est saturé si le flot passant sur l'un de ces arcs est égal à sa capacité

Sous-graphe
graphe induit par une partie des sommets d'un autre graphe

Stable
sous-graphe sans arête

V

Voisin

deux sommets sont voisins si ils sont reliés par une arête (synonime d'adjacent)