MATH-H-404 : Mise à niveau en algorithmique

Bernard Fortz
Nikita Veshchikov
Version 2011-2012

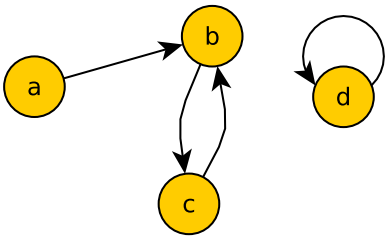
Table des matières

Chapitre 1

Les graphes

1.1 Introduction

Informellement, un graphe se définit comme une structure composée de sommets et soit d'arcs, soit d'arêtes. Nous pouvons, pour donner de l'intuition concernant ce concept, considérer que les sommets représentent des objets et les arcs ou arêtes représentent les relations entre ces objets. Un exemple de graphe est :

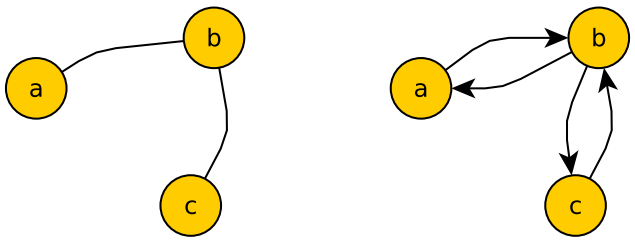


Un arc est une relation dirigée entre deux sommets, il s'agit donc d'une relation non symétrique :

Une arête est une relation non dirigée entre deux sommets, il s'agit ainsi d'une relation symétrique :

Nous pouvons ainsi considérer qu'une arête entre deux sommets est comparable à avoir deux arcs de

sens opposés entre ces deux sommets. Cela nous donne par exemple :



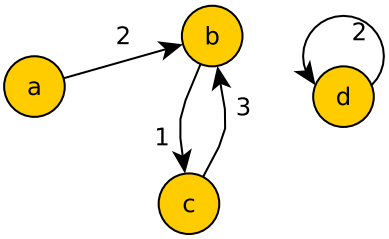
Usuellement, un graphe est composé soit uniquement d'arcs, soit uniquement d'arêtes.

Les graphes représentent un outil fort pratique permettant de modéliser de nombreux systèmes d'informations. Les listes, tableaux, arbres ... peuvent être vus comme des cas particuliers de graphes. Nous pouvons donc voir la notion de graphe comme une structure abstraite d'information (ADT).

Formellement un graphe est défini comme une structure formée de deux ensembles :
– un ensemble S de sommets ($\{a, b, c, d\}$, par exemple) ;
– un ensemble A de couples de sommets ($\{(a, b), (b, c), (c, b), (d, d)\}$, par exemple).

Si l'ordre dans lequel un couple de sommets est exprimé importe on parle alors d'arcs et on considère que la relation est dirigée depuis le premier sommet du couple vers le second sommet du couple. Le graphe résultant est alors dit être dirigé ou orienté. Si l'ordre dans lequel les sommets sont exprimés n'importe pas, on parle alors d'arête et on considère que la relation est non dirigée (dans ce cas (a, b) est équivalent à (b, a)). Le graphe résultant est lui appelé graphe non dirigé.

En outre, les arcs ou les arêtes peuvent posséder une étiquette symbolisant un poids, une valeur, une pondération, etc. Cette étiquette peut être définie à l'aide d'une information additionnelle associée à chaque couple de sommets de l'ensemble A.



Voyons à présent quelques définitions en relation avec les graphes :

– l'ordre d'un graphe est le nombre de sommets qu'il contient ;
– un prédécesseur ou père d'un sommet i est un sommet j tel qu'il existe un arc ou une arête du sommet j vers le sommet i ;
– un successeur ou fils [1] d'un sommet i est un sommet j tel qu'il existe un arc ou une arête du sommet i vers le sommet j ;
– deux sommets i et j qui sont des fils d'un même sommet k, sont dits frères ;
– le degré entrant d'un sommet est le nombre de ses prédécesseurs ;

1. Remarquons que dans le cas d'un graphe non dirigé, deux sommets reliés par une arête sont chacun père et fils l'un de l'autre.

- le degré sortant d'un sommet est le nombre de ses successeurs ;
- deux sommets d'un graphe sont adjacents s'ils sont distincts et s'il existe un arc ou une arête entre les deux sommets ;
- un sommet d'un graphe est isolé s'il n'est adjacent à aucun autre sommet ;
- deux arcs ou arêtes sont adjacents s'ils sont distincts et ont au moins une extrémité commune ;
- lorsque le graphe est tel qu'au plus m arcs ou arêtes vont d'un sommet à un autre sommet, alors le graphe est aussi appelé un m-graphe ;
- une chaîne est une succession non vide d'arêtes contiguës ;
- un chemin est une succession orientée et non vide d'arcs contigus [2] ;
- la longueur d'un chemin ou d'une chaîne est égale au nombre d'arcs ou d'arêtes qui le composent ;
- un chemin où n'apparaît pas deux fois le même arc est appelé un chemin simple ;
- un chemin qui ne passe pas plus d'une fois par chacun de ses sommets est appelé un chemin élémentaire ;
- un cycle, dans un graphe non dirigé, est une chaîne d'un sommet a jusqu'à ce même sommet a qui ne contient pas deux fois le même sommet (autre que a) :
- un cycle dans un graphe dirigé, appelé aussi un circuit, est un chemin d'un sommet a jusqu'à ce même sommet a qui ne contient pas deux fois le même sommet (autre que a) ;
- un chemin d'un sommet a jusqu'à ce même sommet a où chaque arc du graphe n'apparaît qu'une seule fois est appelé un cycle Eulerien ;
- un circuit qui comprend tous les sommets du graphe est appelé un cycle Hamiltonien ;
- un graphe qui ne contient aucun chemin qui part et arrive à un même sommet est appelé un graphe acyclique ;
- un graphe non dirigé qui contient une arête entre chaque paire de sommets est appelé un graphe complet.

1.2 Graphes, arcs et arêtes

Il peut être utile de se rendre compte du nombre de graphes différents que l'on peut construire à partir d'un nombre fixé de sommets. Pour évaluer cela, nous allons déterminer le nombre maximum d'arcs et d'arêtes différents que l'on peut placer dans un graphe de n sommets.

Le nombre maximum d'arêtes que peut contenir un graphe non dirigé de n sommets est $\frac{n(n+1)}{2}$. En effet, un graphe non dirigé, ne contenant qu'un seul sommet, contient au plus une seule arête (de l'unique sommet vers lui-même). De même, un graphe de i sommets, contient au maximum le même nombre d'arêtes qu'un graphe de $i-1$ sommets auquel s'ajoute une arête entre le i^{me} sommet et chacun des $i-1$ autres sommets, ainsi que l'arête du i^{me} sommet vers lui-même. Ainsi, si on note C_i le nombre maximum d'arêtes dans un graphe de i sommets, alors

$$C_i = C_{i-1} + (i-1) + 1 = C_{i-1} + i$$

ce qui revient à écrire :

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

Le nombre maximum d'arcs d'un graphe dirigé de n sommets est n^2. En effet, un graphe dirigé de n sommets contient un arc aller et un arc retour correspondant à chaque arête du graphe non dirigé équivalent, à l'exception des arcs d'un sommet vers lui-même qui n'apparaissent qu'une seule fois. Nous savons qu'un graphe non dirigé de n sommets possède au plus $\frac{n(n+1)}{2}$ arêtes. Nous devons retrancher de

2. Dans la littérature, il arrive qu'une chaîne soit appelée un chemin (ou path en anglais) et un chemin composé d'arcs est alors appelé chemin dirigé (ou directed path en anglais).

ce nombre les n arêtes qui vont d'un sommet vers lui-même, doubler le résultat (puisqu'il y aura dans le graphe dirigé équivalent un arc aller et retour pour chaque arête), et rajouter les n arcs qui vont d'un sommet vers lui-même. Cela donne :

$$2 \cdot \left(\frac{n(n+1)}{2} - n \right) + n = n(n+1) - 2n + n = n^2$$

A partir de cela, sachant qu'un graphe dirigé ou non contient au plus m arcs ou arêtes, pour un nombre de sommets fixé, nous savons que nous pouvons construire 2^m graphes différents, puisque chaque arc ou arête peut être pris ou non dans un graphe.

Sur base de ce qui précède, le tableau ci-dessous reprend le nombre de graphes dirigés et non dirigés que l'on peut construire sur base de n sommets.

n	nombre de graphes non dirigés	nombre de graphes dirigés
2	8	16
3	64	512
4	1024	65536
5	32768	33554432
$\vdots$	$\vdots$	$\vdots$
i	$2^{\frac{i(i+1)}{2}}$	2^{i^2}

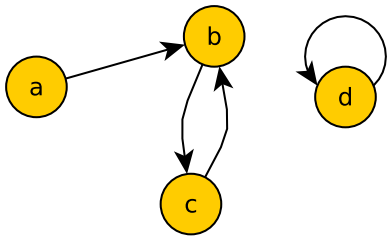
Un graphe qui contient beaucoup moins d'arcs ou d'arêtes que le nombre maximal d'arcs ou d'arêtes qu'il peut contenir est appelé un graphe clairsemé, ou sparse graph. Un graphe qui contient un nombre d'arcs ou d'arêtes proche du nombre maximal d'arcs ou d'arêtes qu'il peut contenir est appelé un graphe dense, ou dense graph en anglais.

1.3 Mise en œuvre

L'implémentation d'un graphe peut être statique, par exemple sous forme d'une matrice d'adjacence ou d'incidence, ou peut être dynamique, par exemple sous forme d'une liste de prédécesseurs et/ou de successeurs.

Une matrice d'adjacence est une matrice carrée $n \times n$, où n est le nombre de sommets du graphe. Cette matrice contient des valeurs booléennes ou des étiquettes. La matrice d'adjacence donne de l'information concernant un lien (via un arc ou une arrête) entre deux sommets.

Ainsi pour le graphe sans étiquette :

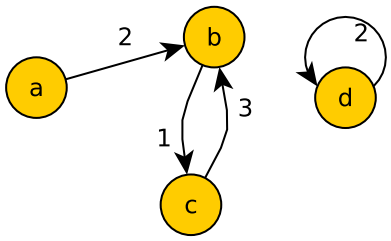


la matrice d'adjacence M correspondante est, où F=faux et V=vrai :

	a	b	c	d
a	F	V	F	F
b	F	F	V	F
c	F	V	F	F
d	F	F	F	V

Remarquons que pour un graphe non dirigé, la matrice M est symétrique.

Pour le graphe :



la matrice d'adjacence M devient, où ici par convention nous utilisons la valeur 0 pour représenter l'absence d'arc ou d'arête entre deux sommets :

	a	b	c	d
a	0	2	0	0
b	0	0	1	0
c	0	3	0	0
d	0	0	0	2

Une autre implémentation possible est la matrice d'incidence. Cette structure donne de l'information sur la relation entre un sommet et un arc ou une arête. L'information se trouvant aux indices i et j indique à la fois si l'arc ou l'arête j pointe vers le sommet i et/ou provient du sommet i.

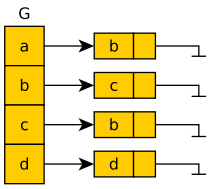
Pour le graphe sans étiquette précédent, la matrice d'incidence I est telle que ci-dessous, où le premier booléen du couple présent en coordonnée (i, j) indique si l'arc ou l'arête j provient du sommet i et où le second booléen de ce couple indique si l'arc ou l'arête j arrive sur le sommet i.

	1	2	3	4
a	(V, F)	(F, F)	(F, F)	(F, F)
b	(F, V)	(V, F)	(F, V)	(F, F)
c	(F, F)	(F, V)	(V, F)	(F, F)
d	(F, F)	(F, F)	(F, F)	(V, V)

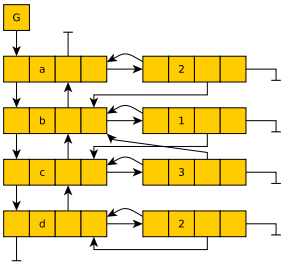
Remarquons qu'une matrice d'incidence pour un graphe non dirigé est telle que chaque couple de booléens de la matrice est composé de deux booléens identiques, il est alors plus économique de n'utiliser qu'un seul booléen pour représenter le couple. Aussi, dans le cas d'un graphe dirigé, si le graphe ne contient aucun arc allant d'un sommet vers lui-même alors les cellules de la matrice d'incidence peuvent contenir des valeurs ternaires, comme par exemple la valeur 1 pour indiquer que l'arc part du sommet,

la valeur -1 pour indiquer que l'arc arrive au sommet et la valeur 0 pour indiquer que l'arc n'est pas liée au sommet.

Une liste de successeurs peut être représentée par une matrice creuse. Nous pouvons aussi considérer une matrice creuse partielle, par exemple par ligne.



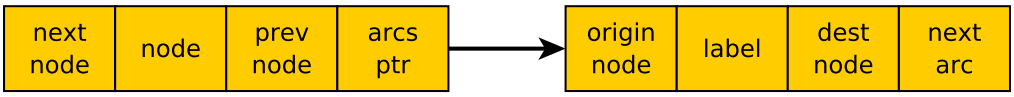
ou



Un graphe dont les arcs ou les arêtes ne portent pas d'étiquette serait alors représenté par un vecteur de pointeurs dont chaque élément représente un sommet de départ d'un arc ou d'une arête et chaque élément pointé par ce vecteur représente un sommet d'arrivée d'un arc ou d'une arête.

Un graphe portant des étiquettes sur ces arcs ou arêtes pourrait être représenté comme sur la figure B ci-dessus. Nous y avons une première liste verticale doublement liée qui représente l'ensemble des sommets de départs des arcs ou des arêtes. De ces sommets sont pointés, sous forme de liste horizontale aussi doublement liée, les éléments qui y sont reliés en y indiquant la valeur de l'étiquette (label) présente sur l'arc ou l'arête ainsi que deux pointeurs vers les sommet origine et destination de la relation ainsi représentée.

La figure ci-dessous reprend la description de ces informations, par manque d'espace nous n'y parlons que d'arcs, il peut s'agir bien sûr d'arcs ou d'arêtes :



On peut, bien entendu, faire de même pour avoir une liste de prédécesseurs. On peut évidemment imaginer encore bien d'autres représentations de graphes.

Quelle que soit le type de représentation utilisée, s'il s'avère que les arêtes du graphe contiennent de nombreuses informations, nous pouvons remplacer, pour chaque arête, le champs correspondant de la structure de donnée représentant le graphe par un pointeur vers une zone de mémoire contenant une description complète de l'arête. Cela permet de ne jamais devoir manipuler et/ou copier ces informations détaillées lorsqu'on construit et lorsqu'on manipule le graphe.

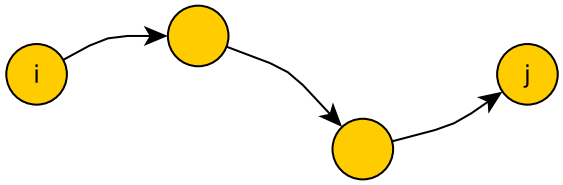
Notons pour finir que la représentation d'un graphe influence l'efficacité des algorithmes le manipulant. Il faut donc choisir la représentation la mieux adaptée aux algorithmes utilisés.

1.4 Accessibilité

Une autre notion importante est celle d'accessibilité. Il s'agit ici de déterminer quels sommets sont accessibles depuis un sommet donné.

Pour ce faire, partons de la matrice booléenne d'adjacence M. L'élément M_{ij} de cette matrice d'adjacence est une valeur booléenne indiquant s'il existe un arc ou une arête allant du sommet i au sommet j.

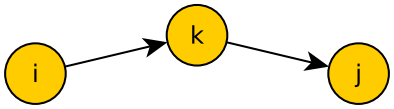
Nous allons construire une matrice reprenant les informations d'accessibilité. Soit A une telle matrice d'accessibilité où A_{ij} indique si le sommet j est accessible à partir du sommet i. Cette matrice répond à la question : existe-t-il un chemin ou une chaîne du sommet i au sommet j :



Pour construire une telle matrice d'accessibilité, partons de notre matrice d'adjacence booléenne M et observons ce que signifie le carré de la matrice d'adjacence. M^2 est telle que pour tout i et j nous avons :

$$M^2_{ij} = (M_{i1} \texttt{ and } M_{1j}) \texttt{ or } \ldots \texttt{ or } (M_{in} \texttt{ and } M_{nj}) = \bigvee_{k=1}^{n} M_{ik} \wedge M_{kj}$$

Si une des expression $M_{ik} \wedge M_{kj}$ est évaluée à vrai alors M^2_{ij} est aussi évaluée à vrai. Si M^2_{ij} est vrai, cela signifie que nous observons, au moins une fois, la configuration :



Il existe ainsi un chemin de longueur deux entre le sommet i et le sommet j.

Par récurrence nous écrivons l'expression de M^r_{ij}, où M^{r-1}_{ij} est l'information indiquant s'il existe un chemin de longueur $r-1$ entre les sommets `i` et `j` :

$$M^r_{ij} = (M_{i1} \texttt{ and } M^{r-1}_{1j}) \texttt{ or } \ldots \texttt{ or } (M_{in} \texttt{ and } M^{r-1}_{nj})$$

Si M^r_{ij} est vrai cela signifie qu'il existe un chemin de longueur r allant du sommet i vers le sommet j

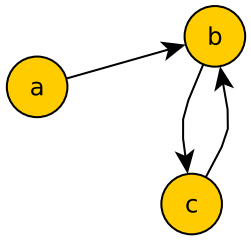
Nous pouvons ainsi définir la matrice d'accessibilité A comme étant la disjonction des puissances successives de la matrice d'adjacence, où n est le nombre de sommets :

$$A = M^0 \vee M^1 \vee M^2 \vee \cdots = \bigvee_{r=0}^{n-1} M^r$$

Nous considérons finalement une disjonction de $n-1$ composantes car pour un graphe de n sommets, s'il existe un chemin entre deux sommets `i` et `j` dans le graphe, alors il existe un chemin entre `i` et `j` qui est d'une longueur au plus égale à $n-1$.

La matrice d'accessibilité est aussi notée M^* et est appelée la fermeture transitive. La fermeture transitive peut être vue comme le graphe résultant de l'adjonction d'un arc entre deux sommets s'il existe un chemin entre ces deux sommets dans le graphe original.

Ainsi, le graphe :



a comme matrice d'adjacence M :

	a	b	c
a	0	1	0
b	0	0	1
c	0	1	0

Pour chaque couple de sommets de ce graphe, il existe un chemin d'une longueur inférieure ou égale à deux (puisque le graphe compte trois sommets), nous avons :

$$M^2 = \begin{array}{c|ccc} & a & b & c \\ \hline a & 0 & 0 & 1 \\ b & 0 & 1 & 0 \\ c & 0 & 0 & 1 \end{array}$$

indiquant qu'il existe un chemin de longueur deux du sommet a au sommet c, du sommet b au sommet b et du sommet c au sommet c. La matrice d'accessibilité, ou fermeture transitive, est alors :

$$A = M^* = M^0 \vee M^1 \vee M^2 = \begin{array}{c|ccc} & a & b & c \\ \hline a & 1 & 1 & 1 \\ b & 0 & 1 & 1 \\ c & 0 & 1 & 1 \end{array}$$

où la première ligne indique qu'il existe un chemin du sommet a au sommet a, du sommet a au sommet b et du sommet a au sommet c, la seconde ligne indique qu'il existe un chemin du sommet b au sommet b et du sommet b au sommet c, enfin la troisième ligne indique qu'il existe un chemin du sommet c au sommet b et du sommet c au sommet c.

1.4.1 Roy-Warshall

L'algorithme de Roy-Warshall permet de construire la matrice d'accessibilité d'un graphe (à laquelle il faudrait encore rajouter M^0 pour être complet). L'algorithme 1.1 présente cette méthode.

Algorithme 1.1 Algorithme de Roy-Warshall

```
A = matrice d'adjacence du graphe

for(k=0 ; k < ordre du graphe ; ++k)
  for(i=0 ; i < ordre du graphe ; ++i)
     if(A[i][k])
       for(j=0 ; j < ordre du graphe ; ++j)
         A[i][j]=A[i][j] or A[k][j]
```

Dans cet algorithme, quand `k=0`, donc au premier tour de la boucle extérieure, nous itérons `i` de `0` à `n-1` et on vérifie pour chaque `i` si `A[i][0]` est vrai, c'est à dire que nous vérifions qu'il existe un arc du sommet `i` au sommet `0`. Si c'est le cas, on itère `j` de `0` jusqu'à `n-1` en calculant `A[i][j]=A[i][j] or A[k][j]` (avec `k`=0). On essaye donc de chercher un chemin de longueur 2 entre les sommets `i` et `j` passant par le sommet `0`, si un chemin de longueur 1 n'existe pas encore.

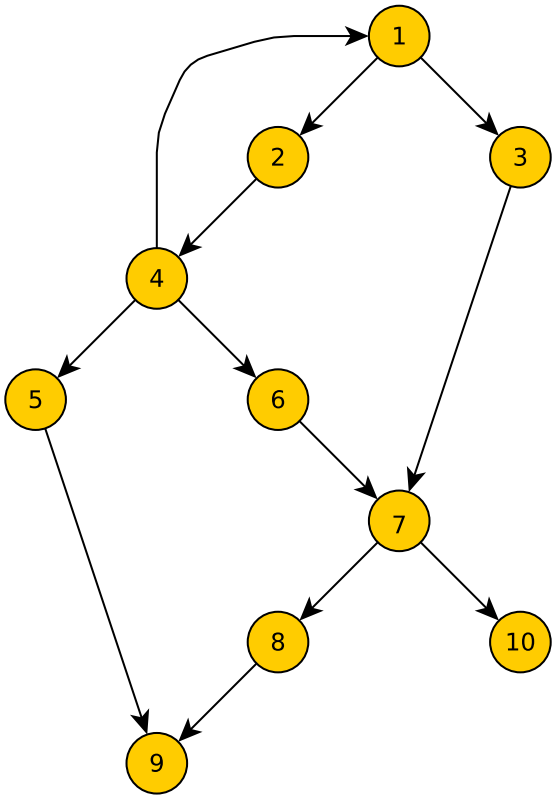
Puis lorsque `k=1`, on cherche un chemin de longueur 2 ou 3 pour rejoindre le sommet `j` depuis le sommet `i`. On a un chemin de longueur 2 si on ne passe pas par le sommet `0` mais bien par le sommet `1` ; et on a un chemin de longueur 3 si on passe à la fois par le sommet `0` et par le sommet `1`. Puis pour `k=2`, on cherche un chemin de longueur 2, 3 ou 4 pour aller de `i` à `j`. Le chemin de longueur 2 ne passe ni par les sommets `0` et `1`, mais bien par le sommet `2` ; le chemin de longueur 3 passe par le sommet `0` ou par le sommet `1`, puis par le sommet `2` ; et le chemin de longueur 4 passe par les sommets `0`, `1` et `2` ...

La complexité maximale de l'algorithme de Roy-Warshall s'exprime en $O(n^3)$ contrairement à $A = \bigvee_{r=0}^{n-1} M^r$ qui est en $O(n^4)$ puisque, pour chaque valeur de r considérée, l'évaluation de M^r consiste en la multiplication de deux matrices [3], à savoir M et M^{r-1}, et cette multiplication matricielle se fait en $O(n^3)$.

1.5 Parcours de graphes

Nous allons à présent examiner différentes manières, en termes d'ordre sur les sommets, pour parcourir un graphe. Pour ce faire, nous allons nous appuyer sur le graphe d'exemple suivant :

3. Comme M_{ij} représente un booléen, la multiplication peut être vue comme un « et logique » (conjonction) et l'addition peut être vue comme un « ou logique » (disjonction).



Il existe, entre autres, deux manières tout à fait classiques pour parcourir un graphe, à savoir le parcours en profondeur et le parcours en largeur. Ces parcours sont aussi respectivement appelés depth-first et breadth-first.

Le parcours en profondeur découvre les sommets du graphe en essayant toujours de descendre, en suivant les arcs ou les arêtes, le plus profondément dans le graphe. Sur base de notre graphe d'exemple ci-dessus, un parcours en profondeur explore les sommets du graphe dans l'ordre suivant :

$$1, 2, 4, 5, 9, 6, 7, 8, 10, 3$$

Nous pouvons noter que l'on revient chaque fois au dernier sommet pour lequel il reste encore des successeurs à visiter.

Le parcours en largeur explore les sommets du graphe par niveau, en découvrant d'abord les sommets à une distance 1, en terme du nombre d'arcs ou d'arêtes, du sommet de départ, puis les sommets à distance 2, etc. Toujours sur base du graphe d'exemple, un parcours en largeur peut se faire dans l'ordre suivant :

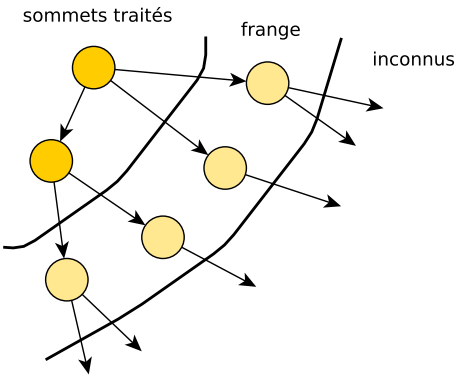
$$1, 2, 3, 4, 7, 5, 6, 8, 10, 9$$

L'ordre précis d'exploration dépend cependant aussi de l'ordre dans lequel on considère les fils d'un sommet (cela peut dépendre de l'implémentation choisie).

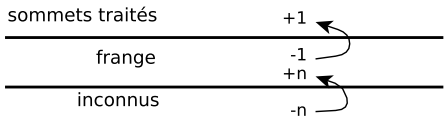
Au cours de ces parcours, à tout moment nous pouvons classifier les sommets du graphe en trois ensembles :

1. l'ensemble des sommets déjà traités (que l'on ne traite qu'une seule fois pour éviter les cycles) ;
2. l'ensemble des sommets frontaliers (on les a vus, en tant que sommets successeurs de sommets déjà traités, mais il ne sont pas encore traités), appelés aussi la frange ;
3. l'ensemble des sommets encore inconnus.

Ces méthodes de parcours de graphes diffèrent dans la manière dont elles déterminent quel sommet doit passer de l'ensemble des éléments frontaliers à celui des éléments traités.



A chaque itération du processus nous sélectionnons un élément de la frange pour être traité et un certain nombre, n, d'éléments inconnus entrent dans la frange :



Ainsi, la manière de choisir les éléments de la frange qui seront traités détermine l'algorithme. Le parcours en profondeur sélectionne le sommet frontalier que l'on a rencontré en dernier au cours du parcours. Par contre, le parcours en largeur sélectionne les sommets frontaliers que l'on a rencontré dans l'ordre dans lequel on les a rencontrés.

Puisque pour un parcours en profondeur nous devons accéder d'abord aux éléments rencontrés le plus récemment, l'usage d'une pile (explicite ou implicite) pour sauvegarder les sommets déjà rencontrés nous sera d'une grande utilité. De la même façon, pour un parcours en largeur, comme nous devons accéder aux éléments dans l'ordre dans lequel on les a rencontrés, l'usage d'une file (ou queue) s'impose pour sauvegarder les sommets rencontrés.

Sur base de cela, nous pouvons écrire récursivement un parcours en profondeur tel que présenté par l'algorithme 1.2. Par souci de concision nous ne parlerons que d'arcs par la suite. Les algorithmes de parcours considérés fonctionnent de la même manière si on manipule des graphes non dirigés.

Au cours de l'exécution de cet algorithme nous traitons un à un les sommets, en partant d'un premier sommet indiqué comme paramètre. Pour éviter de traiter deux fois un même sommet, nous « marquons »

les sommets traités (au moyen d'une information booléenne par exemple). L'algorithme présenté suppose qu'initialement tous les sommets sont démarqués (la marque est initialisée à la valeur booléenne faux, par exemple). L'algorithme, ainsi que les algorithmes suivants, supposent aussi que le graphe est représenté d'une manière telle qu'à partir de chaque sommet nous pouvons accéder successivement à chaque arc dont ce sommet est l'origine, en utilisant les fonctions `1er_arc`, puis `arc_suivant`. De même, nous accédons au sommet se trouvant à l'extrémité d'un arc en utilisant la fonction `extremite`.

Pour chaque sommet traité, nous parcourons récursivement le graphe à partir de chacun de ses fils.

Pour ce faire, nous examinons un à un les fils du sommet courant traité, éventuellement nous réalisons un traitement lié à l'arc ou à l'arête reliant le sommet traité et le fils considéré. Puis, si le fils n'a pas encore été traité, nous continuons, récursivement, le parcours à partir de ce fils.

Lors du retour de l'appel récursif qui a réalisé le parcours à partir d'un fils, nous considérons l'éventuel fils suivant, que l'on peut donc aussi considérer comme étant le frère du sommet à partir duquel on vient de réaliser un parcours. Un parcours sera à nouveau réalisé à partir de ce frère si ce dernier n'a pas encore été traité, et n'est donc pas marqué. Et on continue ainsi pour tous les sommets du graphe.

Algorithme 1.2 Parcours en profondeur récursif

```
void DF(sommet s)
{
  traiter s
  marquer s
  a=s->1er_arc
  while(a existe)
  {
    traiter a
    if(a->extremite n'est pas marquée)
       DF(a->extremite)
    a=a->arc_suivant
  }
}
```

Il est bien sûr possible d'écrire une version itérative du parcours en profondeur, tel qu'illustré par l'algorithme 1.3. L'algorithme présenté suppose qu'initialement tous les sommets sont démarqués.

Cet algorithme utilise une pile d'arcs (ou de pointeurs vers des arcs). La structure de données considérée pour représenter le graphe est telle que, pour un sommet, on accède à ses différents sommets fils en passant directement d'un arc entre le sommet père et le sommet fils aux arcs suivants reliant le sommet père et les autres sommets fils. Nous utilisons, pour réaliser cela, la fonction `arc_suivant`.

L'algorithme parcourt un à un les différents sommets du graphe en sauvegardant dans une pile un accès vers chaque sommet frère du sommet en cours de traitement lors du parcours. Puis l'algorithme passe à un des fils du sommet en cours de traitement ; sommet fils qui devient le nouveau sommet en cours de traitement. Lorsqu'un sommet sans fils non traité est rencontré, cela signifie que nous sommes « descendus » le plus en profondeur dans le graphe, il faut continuer le parcours en remontant au dernier sommet rencontré ayant un frère non encore traité. Pour ce faire on extrait le sommet qui se trouve au sommet de la pile et on recommence le processus à partir de ce nouveau sommet.

Plus précisément, l'algorithme démarre depuis un premier sommet `s` donné comme paramètre. On exécute alors une boucle dont le corps réalise les traitements suivants tant que le sommet `s` considéré n'est pas déjà marqué :

– on marque ce sommet `s` pour l'indiquer comme étant traité ;
– on accède à son premier fils, appelons-le `s'`, et on traite, si nécessaire, l'information liée à l'arc `a` ayant mené à ce fils ;
– on assigne à la variable `a` l'arc suivant, à savoir l'arc partant du sommet `s` et menant à un autre fils de `s`, donc un frère de `s'` ;
– on sauve ce nouvel arc `a`, s'il existe, sur la pile ;
– on sauve `s'` dans `s` ;

Lorsque la boucle précédente se termine, car un sommet `s` déjà traité est rencontré :

– on extrait l'arc se trouvant au sommet de la pile ;
– on traite l'information qui lui est associé ;
– on sauve dans `s` le sommet qui est à l'extrémité de cet arc ;
– on sauve dans `a` l'arc suivant menant au frère suivant ;
– on sauve ce nouvel arc `a`, s'il existe, sur la pile ;

Tout cela est réalisé tant que tous les sommets n'ont pas été marqués ou tant que la pile n'est pas vidée de tous les arcs qui y ont été sauvés.

Algorithme 1.3 Parcours en profondeur itératif

```
void DF(sommet s)
{
  do
  {
     while(s n'est pas marqué)
     {
       traiter s
       marquer s
       a=s->1er_arc
       if(a existe)
       {
         traiter a
         s=a->extremite
         a=a->arc_suivant
         if(a existe) push(a)
       }
     }
     if(stack n'est pas vide)
     {
       a=pop()
       traiter a
       s=a->extremite
       a=a->arc_suivant
       if(a existe) push(a)
     }
  }
  while(s n'est pas marqué or stack n'est pas vide)
}
```

Le parcours en largeur, dont l'algorithme 1.4 propose une implémentation, nécessite l'usage d'une file

de sommets (ou de pointeurs vers des sommets). L'algorithme présenté suppose qu'initialement tous les sommets sont démarqués.

L'algorithme parcourt un à un tous les sommets du graphe en sauvant dans une file tous les fils du sommet en cours de traitement lors du parcours. Puis l'algorithme passe au plus ancien sommet sauvé dans la file, sommet fils qui devient le nouveau sommet en cours de traitement. Ce traitement est réalisé tant que la file n'est pas vidée.

Plus précisément, l'algorithme démarre à partir d'un premier sommet `s` indiqué comme paramètre, traite et marque ce sommet `s` et le sauvegarde dans la file. Puis une boucle est réalisée au cours de laquelle tant que la file n'est pas vidée on réalise les opérations suivantes :

- on retire le plus ancien élément encore dans la file et on le sauve dans `s` ;
- on accède au premier arc `a` qui émane de `s` ;
- tant qu'il y a des arcs sortants de `s`, on réalise les opérations suivantes :
  - on traite l'information associée à l'arc `a` ;
  - on sauve dans `s` le sommet qui est à l'extrémité de l'arc `a` ;
  - si ce nouveau sommet `s` n'a pas encore été traité, on le traite, on le marque et on le sauve dans la file ;
  - on passe à l'arc suivant donnant accès au frère de `s` et on sauve ce nouvel arc dans `a`.

Algorithme 1.4 Parcours en largeur

```
void BF(sommet s)
{
  traiter s
  marquer s
  enqueue(s)
  while(queue n'est pas vide)
  {
    s=dequeue()
    a=s->1er_arc
    while(a existe)
    {
      traiter a
      s=a->extremite
      if(s n'est pas marqué)
      {
        traiter s
        marquer s
        enqueue(s)
      }
      a=a->arc_suivant
    }
  }
}
```

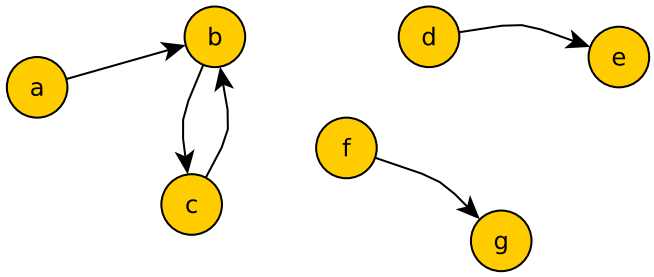
Il est important de se rendre compte que ces algorithmes de parcours peuvent être écrits de multiples façons. Par exemple, nous pouvons considérer le parcours en largeur illustré par l'algorithme 1.5 où un vecteur `val` est utilisé pour gérer le marquage. Cet algorithme fait, bien entendu, usage d'une file (une file d'entiers dans ce cas) et suit le même canevas que le parcours présenté par l'algorithme 1.4.

Les deux types d'algorithmes de parcours, en profondeur et en largeur, passent en revue, lors du parcours, tous les sommets et tous les arcs. Ainsi la complexité maximale de ces algorithmes s'exprime en $O(|S|+|A|)$, où S et A représentent respectivement l'ensemble des sommets et des arcs ou arêtes et où $|S|$ et $|A|$ dénotent la taille de ces ensembles.

Algorithme 1.5 Parcours en largeur au moyen d'un vecteur

```
void BF()
{
  n=0
  val=(0,...,0)
  enqueue(1)
  while(queue n'est pas vide)
  {
    ++n
    k=dequeue()
    val[k]=n
    t=1er successeur du sommet k
    while(t existe)
    {
      if(val[t] == 0)
      {
         enqueue(t)
         --val[t]
      }
      t=successeur suivant de t
    }
  }
}
```

Il est en pratique tout à fait possible qu'un graphe possède plusieurs sommets origine. On dit alors qu'il est formé de plusieurs composantes. Comme par exemple :



Lors d'un parcours il faut alors parcourir chacune de ces composantes. Pour ce faire, il est nécessaire d'envisager une boucle générale qui lance les parcours considérés ci-dessus pour chaque sommet origine de chaque composante formant le graphe.

Une façon de faire, basée uniquement sur la recherche de sommets non marqués au sein du graphe après chaque parcours partiel, est présentée par les algorithmes 1.6 et 1.7, réalisant respectivement un parcours en profondeur et en largeur.

Algorithme 1.6 Parcours en profondeur généralisé

```
void DepthFirst(graphe g)
{
  démarquer g
  n=ordre de g
  for(s=1 ; s <= n ; ++s)
     if(sommet s n'est pas marqué)
        DF(s)
}
```

Algorithme 1.7 Parcours en largeur généralisé

```
void BreadthFirst(graphe g)
{
  démarquer g
  n=ordre de g
  for(s=1 ; s <= n ; ++s)
     if(sommet s n'est pas marqué)
        BF(s)
}
```

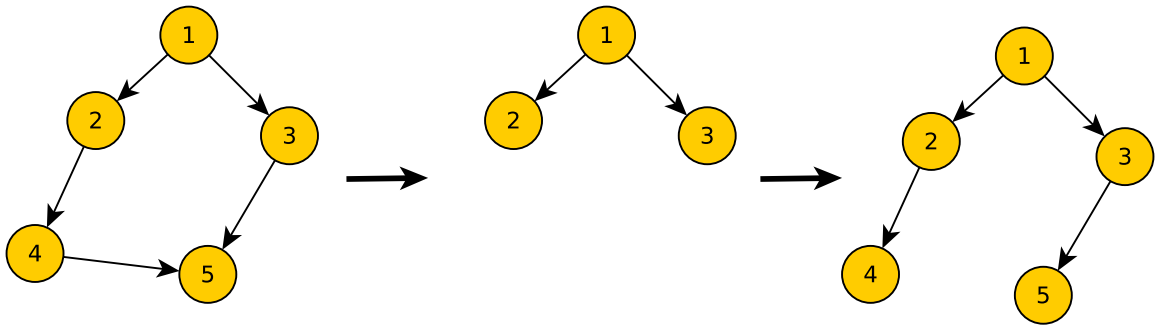
Chapitre 2

Les plus courts chemins

2.1 Introduction

Trouver le plus court chemin, ou shortest path en anglais, entre deux sommets d'un digraphe pondéré consiste en la recherche du chemin dont la somme des pondérations des arcs le composant est minimale, parmi tous les chemins reliant les deux sommets donnés.

Nous pouvons d'ores et déjà identifier le cas particulier consistant en un digraphe où tous les poids sont égaux. Le parcours en largeur nous permet, dans ce cas, de trouver le plus court chemin entre un sommet donné et tous les autres sommets du graphe. En effet, il s'agit alors de chercher les chemins comptant le moins d'arcs possibles entre les sommets considérés. Nous pouvons illustrer cela par l'exemple suivant :

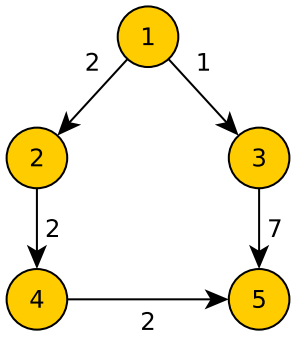


2.2 Plus courts chemins dans un digraphe

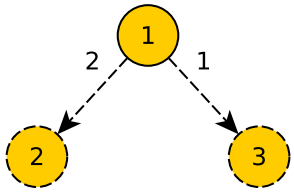
D'une manière générale, on pose donc le problème consistant à trouver les plus courts chemins reliant un sommet x d'un digraphe donné à tous les autres sommets de ce digraphe. Nous considérons dans un premier temps que les pondérations sur les arcs sont toutes positives. Un chemin le plus court ne pouvant alors « tourner » sur lui-même, sous peine de se rallonger, il n'y a jamais de cycle dans de tels plus courts chemins entre deux sommets. Ainsi, la construction des plus courts chemins depuis un sommet du graphe conduit à produire un arbre recouvrant du digraphe.

On construit l'arbre des plus courts chemins à partir d'un sommet x en ajoutant, à chaque étape de cette construction, le sommet de la frange le plus proche du sommet x.

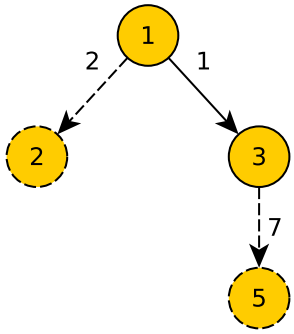
Illustrons cela en partant du digraphe suivant [1] :



Partons du sommet `1` :

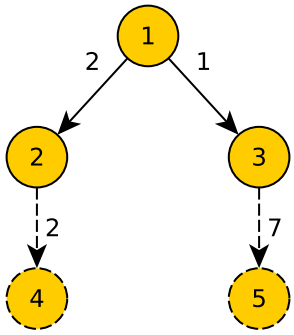


Le sommet le plus proche du sommet `1` est le sommet `3` :

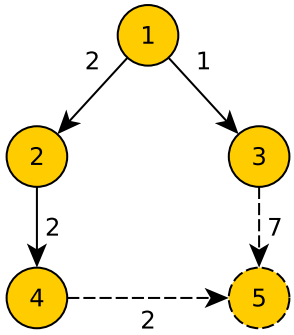


Cette fois le sommet le plus proche, toujours du sommet `1`, est le sommet `2` (distant de 2 unités, et non le sommet `5` distant de 8 unités) :

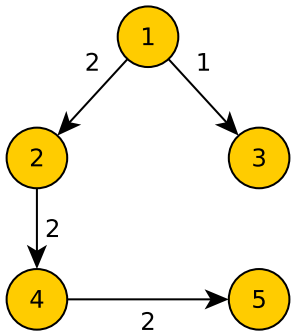
1. Les figures suivantes représenteront en pointillés les éléments se trouvant dans la frange.



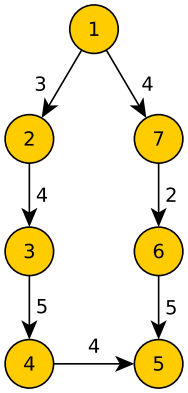
Le sommet le plus proche du sommet `1` est à présent le sommet `4` :



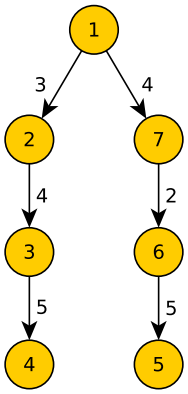
Le sommet le plus proche du sommet `1` est maintenant le sommet `5`, en passant par le sommet `4` (et non pas par le sommet `3`) :



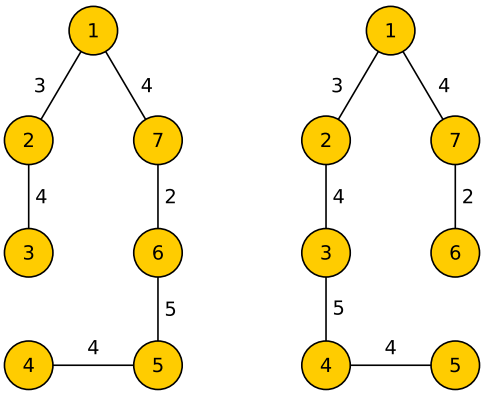
Considérons un second exemple en partant du digraphe suivant :



L'arbre des plus courts chemins partant du sommet initial `1` est :



Remarquons que cet arbre est bien différent de l'arbre sous-tendant minimal du graphe non dirigé équivalent, qui peut être pour cet exemple un des deux arbres suivants :



Ces arbres ont tous deux un poids total égal à 22 (alors que l'arbre des plus courts chemins est d'un poids total de 23). Dans l'arbre de gauche le chemin du sommet `1` au sommet `4` est de longueur 15, et non de longueur 12 comme l'est le plus court chemin. Pour l'arbre de droite, le chemin du sommet `1` au sommet `5` est de longueur 16, alors que le plus court chemin entre ces deux sommets est de longueur 11.

Pour réaliser le calcul des plus courts chemins à partir d'un sommet initial, nous pouvons modifier l'algorithme de Prim en stockant dans un vecteur, que nous appellerons `dist`, la distance de chaque sommet avec le sommet initial `x`. Quand on rajoute un sommet `k` dans l'arbre en construction, on met à jour la frange en parcourant la liste de successeurs de ce sommet `k`. Pour chaque sommet `t` de cette liste de successeurs, la distance du sommet `x` aux sommets de cette liste est égale à
`dist[k] + distance de k à t`.

La propriété fondamentale sur laquelle se base cet algorithme est que, dans un digraphe, le chemin le plus court entre un sommet `a` et un sommet `c` est composé du plus court chemin entre le sommet `a` et un sommet `b` et du plus court chemin entre ce sommet `b` et le sommet `c`, et ce pour tout sommet `b` du digraphe se trouvant sur le plus court chemin entre les sommets `a` et `c`.

L'algorithme 2.1 réalise cette recherche des plus courts chemins depuis un sommet initial `s` jusqu'à tous les autres sommets du digraphe. Pour réaliser cela, nous manipulons un ensemble `M` de sommets contenant initialement tous les sommets du graphe à l'exception du sommet initial `s`.

Algorithme 2.1 Algorithme de Dijkstra

```
M=ensemble vide
for(i=1; i<= ordre du graphe; ++i)
{
  dist[i]=poids arc(s,i)
  prec[i]=s
  rajouter le sommet i à l'ensemble M
}
retirer le sommet s de l'ensemble M
while(l'ensemble M n'est pas vide)
{
  m=sommet x appartenant à M tel que dist[x] est minimum
  retirer le sommet m de l'ensemble M
  if(dist[m] == infini)
    M=ensemble vide
  else
  {
    for(a=1er arc sortant de m ; a existe ; a=arc suivant)
    {
      y=extremité de a
      if(y appartient à M)
      {
        v=dist[m]+poids arc(m,y)
        if(v < dist[y])
        {
          dist[y]=v
          prec[y]=m
        }
      }
    }
  }
}
```

Nous manipulons aussi deux vecteurs, `dist` et `prec`, tels que pour chaque sommet `i` distinct du sommet `s`, la valeur contenue en `dist[i]` indique la plus courte distance connue entre le sommet `s` et le sommet `i`, et `prec[i]` reprend le numéro du sommet précédant le sommet `i` sur le plus court chemin en construction entre les sommets `s` et `i`.

Initialement, `dist[i]` contient le poids présent sur l'arc entre le sommet `s` et le sommet `i`. Si cet arc n'existe pas, le poids est supposé égal à l'infini. Les entrées correspondantes du vecteur `prec` valent toutes initialement `s`, puisque nous ne considérons dans un premier temps que les chemins ne comportant qu'un seul arc entre le sommet `s` et tous les autres sommets.

Par la suite, en découvrant de nouveaux arcs, nous essayons d'allonger, en nombre d'arcs, les chemins, en constatant éventuellement que nous améliorons, en terme de distance, la situation existante.

Pour ce faire, nous réalisons une boucle qui retire de l'ensemble `M` le sommet `m` dont la valeur correspondante dans le vecteur `dist`, à savoir `dist[m]`, est la plus petite. Pour chaque sommet `y` successeur de ce sommet `m`, nous regardons s'il appartient à l'ensemble `M`. Si le sommet `y` n'y appartient pas, cela

signifie qu'il est déjà traité. Sinon, nous mettons à jour les vecteurs `dist` et `prec` si, en passant par le sommet `m`, le sommet `y` est plus proche du sommet initial `s`.

Si `dist[m]` valait l'infini lors du retrait du minimum de l'ensemble `M`, cela signifie que le sommet `m` appartient à une autre composante que le sommet `s`. Cela sera alors aussi le cas pour tous les autres sommets potentiellement encore présents dans `M`. Dans cette situation, la recherche des plus courts chemins depuis le sommet `s` peut s'arrêter. Ces sommets encore présents dans l'ensemble `M` sont à une distance infinie du sommet `s`, valeur déjà présente pour ces sommets dans le vecteur `dist`.

L'ensemble `M` peut être géré au moyen d'un heap de sommets dont la priorité de chaque sommet est inversement proportionnelle à la distance courante entre le sommet `s` initial et le sommet considéré. Ainsi, dans l'algorithme 2.1, l'instruction `rajouter le sommet i à l'ensemble M` est d'une complexité maximale qui s'exprime en $O(\log(|S|))$. Cette instruction est réalisée $|S|$ fois.

La boucle `while` tourne au plus $|S|$ fois et réalise, entre autres, l'extraction du minimum du heap. Cette opération s'exprime au pire en $O(\log(|S|))$.

De plus, par les deux boucles imbriquées `while` et `for` nous parcourons, au total, une fois chaque arc du graphe, et pour certains de ces arcs, nous modifions le heap en réalisant l'instruction `dist[y]=v` qui modifie la priorité du sommet `y`. Cette modification s'effectue avec une complexité maximale s'exprimant en $O(\log(|S|))$. Au pire, cela donne donc, pour l'ensemble des arcs considérés, une complexité de l'ordre de $O(|A|\log(|S|))$.

Au total, nous avons une complexité maximale qui s'exprime en :

$$O(|S|\log(|S|) + |S|\log(|S|) + |A|\log(|S|)) = O(|S|\log(|S|) + |A|\log(|S|))$$

Et comme, au pire, $|A| = |S|^2$, la complexité maximale peut s'exprimer en :

$$O(|A|\log(|S|))$$

2.3 Tous les plus courts chemins

Nous avons vu à la section précédente une technique donnant comme résultat, pour un digraphe, tous les plus courts chemins à partir d'un sommet de départ. Néanmoins, on peut aussi désirer connaître, a posteriori, les plus courts chemins depuis n'importe quel sommet du digraphe vers tous les autres sommets.

Pour réaliser ce calcul, nous pourrions réaliser une recherche des plus courts chemins, tel que réalisé par l'algorithme de Dijkstra, en partant tour à tour de chaque sommet du digraphe.

Une autre approche consiste en l'usage de la technique calculant la fermeture transitive du graphe, comme défini à la section 1.4.1. Nous rajoutons ici au graphe une information relative aux sommets `x` et `y` qui indique s'il existe un chemin allant du sommet `x` au sommet `y`. Des informations peuvent ainsi être spécifiées pour tous les couples de sommets du digraphe.

Nous avons déjà vu l'algorithme 1.1, présenté à la section 1.4.1, qui calcule la fermeture transitive d'un graphe. Nous allons exploiter cette technique de construction de la fermeture transitive en considérant que s'il existe un moyen pour aller du sommet `x` au sommet `y` en n'utilisant que des sommets d'indice inférieur ou égal à `k-1` et s'il existe un moyen pour aller du sommet `k` au sommet `y`, alors il existe un moyen pour aller du sommet `x` au sommet `y` en n'utilisant que des sommets d'indice inférieur ou égal à `k`. Il est ainsi espéré que le nouveau chemin considéré, passant par le sommet `k`, soit plus court que tous ceux déjà explorés. Il faut donc passer en revue tous les chemins possibles, sans cycle, entre chaque paire de sommets.

Sur base de cela, nous pouvons construire, par exemple de la manière proposée par l'algorithme 2.2, un tableau reprenant le plus court chemin de tout sommet `x` à tout sommet `y` du digraphe considéré.

Algorithme 2.2 Algorithme de Floyd

```
for(k=0 ; k < ordre du graphe ; ++k)
  for(x=0 ; x < ordre du graphe ; ++x)
    if(mat[x][k] < infini)
      for(y=0 ; y < ordre du graphe ; ++y)
        if(mat[k][y] < infini)
          if(mat[x][k]+mat[k][y] < mat[x][y])
            mat[x][y]=mat[x][k]+mat[k][y]
```

Le graphe est représenté, dans cet algorithme, par une matrice d'adjacence `mat` qui sera modifiée par l'algorithme de manière à finalement contenir les plus courts chemins entre tous les couples de sommets du digraphe. Initialement, nous trouvons en `mat[x][y]` le poids de l'arc allant du sommet `x` au sommet `y`. Si cet arc n'existe pas, l'entrée correspondante de la matrice vaut l'infini.

Il est bien sûr nécessaire d'ajouter à l'algorithme de base utilisé, l'algorithme 1.1, un test détectant si en passant par le sommet `k`, le chemin entre les sommets `x` et `y` s'améliore. C'est uniquement en cas d'amélioration que la valeur de `mat[x][y]` est modifiée. Ce test est réalisé par l'avant-dernière instruction de l'algorithme.

A la fin de l'exécution de l'algorithme, pour tout couple de sommets `x` et `y` du digraphe, `mat[x][y]` contient la taille du plus court chemin allant du sommet `x` au sommet `y`.

Le trajet du plus court chemin entre deux sommets peut être reconstruit en utilisant une matrice supplémentaire qui retient en coordonnées `(i,j)` le nom du sommet précédant le sommet `j` sur le chemin du sommet `i` au sommet `j`.

La complexité maximale de cet algorithme s'exprime, comme nous l'avons déjà vu, en $O(|S|^3)$.

2.4 Plus courts chemins dans un DAG

Considérons à présent le cas particulier où un plus court chemin doit être trouvé dans un digraphe acyclique. Dans ce cas, nous allons constater que réaliser un tri topologique peut nous être d'une grande aide.

Pour déterminer le plus court chemin depuis un sommet initial `s` jusqu'à un sommet `x`, il faut connaître le plus court chemin du sommet `s` jusqu'à chacun des prédécesseurs du sommet `x`. Ainsi, on ne traite un sommet que si tous ses prédécesseurs sont traités. Nous sommes donc bien confrontés à un tri topologique.

L'idée est qu'une fois que tous les plus courts chemins entre le sommet de départ `s` et tous les prédécesseurs d'un sommet `x` sont connus, il est facile de déterminer le plus court chemin du sommet `s` au sommet `x`. Il suffit en effet, pour chaque prédécesseur `y` du sommet `x`, d'additionner la taille du plus court chemin entre les sommets `s` et `y` à la distance reprise sur l'arc entre les sommets `y` et `x`. Le résultat le plus petit indiquera le plus court chemin entre `s` et `x`.

L'algorithme de Bellman, illustré par l'algorithme 2.3, utilise donc un tri topologique. Pour réaliser celui-ci, nous devons pouvoir extraire le sommet le plus proche de l'origine, à partir de l'ensemble des sommets traitables (c'est à dire ceux dont tous les prédécesseurs ont été traités).

Algorithme 2.3 Algorithme de Bellman

```
dist[0 ... n-1] = +infini
dist[s] = 0
renuméroter les sommets dans l'ordre topologique
for(int k=1 ; k < ordre du graphe ; ++k)
{
   j = i tel que pour tous les arcs (i,k) dans le graphe
         on ait dist[i]+mat[i][k] qui soit minimum
   dist[k] = dist[j]+mat[j][k]
   pred[k]=j
}
```

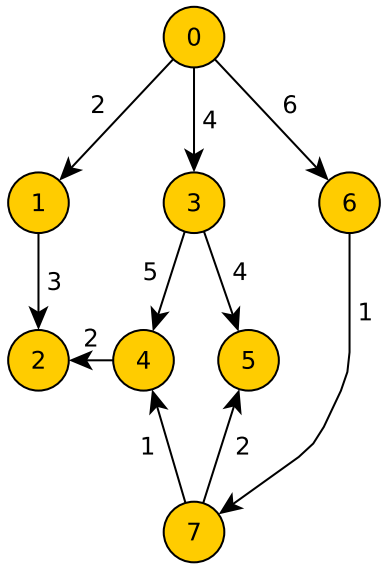
La réalisation du tri topologique nécessaire à la renumérotation des sommets est d'une complexité maximale s'exprimant en $O(|S| + |A|)$.

Au cours de la boucle `for`, nous passons en revue tous les sommets du DAG et, pour chaque sommet, nous regardons les arcs qui y mènent. Nous observons donc, au cours de cette boucle, au total une fois tous les arcs et une fois tous les sommets du DAG, la complexité maximale s'exprime donc en $O(|S| + |A|)$.

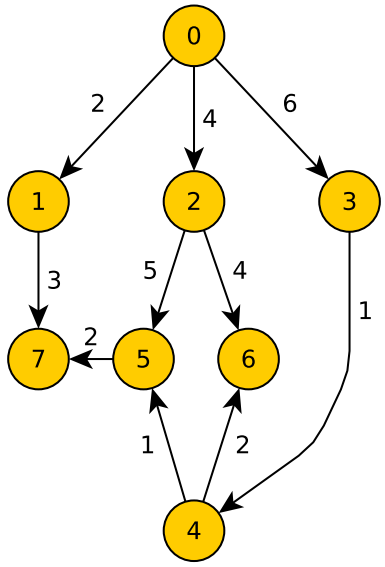
Cela nous donne bien une complexité maximale totale de l'algorithme s'exprimant en $O(|S| + |A|)$.

Si $|S|$ est approximativement égal à $|A|$, ce qui semble une hypothèse raisonnable pour un DAG, alors cet algorithme pour la recherche de plus courts chemins dans un DAG est d'une complexité inférieure à celle de Dijkstra.

Nous pouvons illustrer cette technique au moyen du graphe d'exemple suivant :



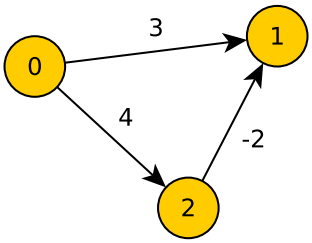
Après renumérotation nous obtenons le graphe :



On accède donc tour à tour aux sommets dans l'ordre de leur numérotation. Ensuite nous considérons, dans cet ordre, un à un les sommets du digraphe. Pour chaque sommet `k` ainsi considéré, nous sélectionnons l'arc `(j,k)` dont le poids plus la distance du sommet initial au sommet `j` est minimal. Nous indiquons alors que ce sommet `k` est accessible depuis le sommet initial par un chemin de longueur `dist[j]+mat[j][k]`, et que le prédécesseur du sommet `k` sur ce plus petit chemin est le sommet `j`.

2.5 Digraphes ayant des poids négatifs

Si des poids négatifs apparaissent sur les arcs du digraphe, alors l'algorithme de Dijkstra ne fonctionne plus. Par exemple, considérons le digraphe suivant :



Nous constatons que l'algorithme de Dijkstra trouverait un chemin du sommet `0` au sommet `1` de longueur 3 et un chemin du sommet `0` au sommet `2` de longueur 4. Alors que pour aller du sommet `0` au sommet `1` le plus court chemin passe par le sommet `2` et est de longueur 2.

L'algorithme de Dijkstra ne voit donc pas que pour aller du sommet `0` au sommet `1` le plus court chemin passe par le sommet `2`. La présence de poids négatifs peut avoir comme effet que les plus courts chemins ont tendance à avoir plus d'arcs que les plus courts chemins dans des graphes dont les arcs n'ont pas de poids négatif.

En effet, dans un graphe n'ayant que des poids positifs sur ses arcs, quand un sommet `t` est considéré, par l'algorithme de Dijkstra, comme faisant partie d'un plus court chemin, nous sommes sûr d'avoir trouvé la plus courte distance du sommet d'origine à ce sommet `t`. Si ce n'était pas le cas, cela signifierait qu'il existerait un autre chemin allant du sommet d'origine au sommet `t` en passant par un sommet `u` non encore considéré. Or ce cas de figure ne peut se présenter car comme nous extrayons toujours le minimum de l'ensemble `M`, le sommet `u` est plus éloigné du sommet d'origine que l'est le sommet `t`. Passer par le sommet `u` pour aller du sommet d'origine au sommet `t` rallongerait donc la distance totale. Par contre, dans le cas de figure où des poids négatifs apparaissent sur les arcs, même si un sommet `u` est plus éloigné du sommet d'origine que le sommet `t`, il peut être avantageux de passer par ce sommet `u` puisque le plus court chemin du sommet `u` au sommet `t` pourrait être d'un poids total négatif.

Si le poids d'un cycle est la somme des poids des arcs qui le composent, alors un graphe ayant un cycle de poids négatif n'a pas de plus court chemin puisque tant que l'on tourne dans le cycle, le poids total du chemin, décroît alors que sa taille, en nombre d'arcs, augmente.

Nous considérons donc des graphes ayant des arcs de poids négatifs, mais sans cycle de poids négatif. Bellman et Ford ont proposé l'algorithme 2.4 qui calcule le plus court chemin depuis un sommet initial du digraphe et qui retourne un booléen `false` si un cycle de poids négatif est présent dans le graphe.

L'algorithme fonctionne en associant les sommets et les arcs à des poids. Le poids d'un arc représente, comme précédemment, la distance entre les sommets aux extrémités de l'arc, et le poids d'un sommet représente la distance entre le sommet initial considéré du digraphe et le sommet en question.

L'algorithme passe en revue tous les arcs `(u,v)` du digraphe, en comparant le poids du sommet `u` additionné au poids de l'arc `(u,v)` avec le poids du sommet `v`. Si l'addition produit une valeur inférieure au poids du sommet `v`, nous modifions alors le poids du sommet `v` et nous indiquons dans un vecteur `pred` que le prédécesseur du sommet `v` sur le plus court chemin est le sommet `u`.

Ce traitement est réalisé autant de fois qu'il y a de sommets dans le digraphe moins un. En effet, dans un tel graphe, le chemin le plus long, si on néglige les cycles, est au plus égal au nombre de sommets dans le digraphe moins un. Donc au premier tour de la boucle englobante on considère tous les chemins composés d'un arc. Puis, au tour suivant, on considère les chemins composés de deux arcs, puisqu'on ajoute un arc aux chemins déjà considérés au tour précédent si la somme des poids de ces arcs donne un résultat inférieur, et donc une longueur de chemin inférieure, à ce qui était précédemment obtenu. Nous continuons à rajouter un arc à chaque tour de la boucle englobante, jusqu'à considérer les chemins potentiels les plus longs, à savoir passant par tous les sommets, si cela est possible.

Algorithme 2.4 Algorithme de Bellman et Ford

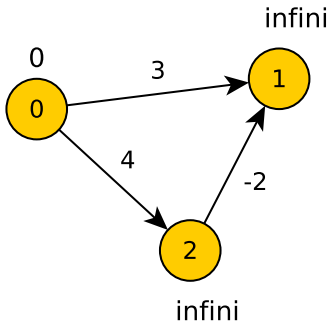
```
mettre le poids de tous les sommets à +infini
mettre le poids du sommet initial à 0

for(int i=1 ; i < ordre du graphe ; ++i)
{
  for((u,v)=1er arc du graphe; (u,v) existe ; (u,v)=arc suivant)
  {
    val=poids du sommet u + poids de l'arc (u,v)
    if(val < poids du sommet v)
    {
      pred[v]=u
      poids du sommet v=val
    }
  }
}
for((u,v)=1er arc du graphe; (u,v) existe ; (u,v)=arc suivant)
  if(poids du sommet u + poids de l'arc (u,v) < poids du sommet v)
    return(false)
return(true)
```

La dernière boucle de l'algorithme permet de détecter si le digraphe possède un cycle de poids négatifs. Pour ce faire, cette boucle simule le rajout d'un arc de plus sur les chemins et vérifie si les sommets ainsi atteint ne sont pas plus proche du sommet initial après ce rajout. Comme la partie précédente de l'algorithme a considéré les chemins les plus longs, en nombre d'arcs, envisageables au cas où il n'y avait de cycle de poids négatif dans le digraphe, si cette dernière vérification indique la construction d'un chemin plus court, en terme de longueur, cela signifie bien qu'un cycle de poids négatif est détecté.

L'initialisation est d'une complexité maximale en $O(|S|)$, puis les deux boucles imbriquées tournent de l'ordre de $O(|S|)$ fois pour la boucle englobante et de l'ordre de $O(|A|)$ pour la boucle imbriquée. Au total cela nous donne une complexité maximale s'exprimant en $O(|S| + |S||A|)$. Ce qui s'approxime donc par $O(|S||A|)$.

Si nous considérons à nouveau le graphe d'exemple :

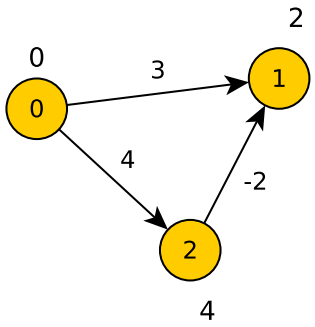


Au premier tour de boucle, nous considérons tour à tour :

– l'arc `(0,1)` et nous changeons le poids du sommet `1` par le poids du sommet `0` additionné au poids

de l'arc `(0,1)`, ce qui fait $0 + 3 = 3 < \infty$;
– l'arc `(0,2)` et nous changeons le poids du sommet `2` par le poids du sommet `0` additionné au poids de l'arc `(0,2)`, ce qui fait $0 + 4 = 4 < \infty$;
– l'arc `(2,1)` et nous changeons le poids du sommet `1` par le poids du sommet `2` additionné au poids de l'arc `(2,1)`, ce qui fait $4 - 2 = 2 < 3$.

A la fin de ce premier tour, les sommets du graphes sont donc pondérés comme ci-dessous :



Au second tour de boucle, nous considérons tour à tour :

– l'arc `(0,1)` et nous ne changeons pas le poids du sommet `1` car le poids du sommet `0` additionné au poids de l'arc `(0,1)` donne $0 + 3 = 3 > 2$, où le poids du sommet `1` vaut 2 ;
– l'arc `(0,2)` et nous ne changeons pas le poids du sommet `2` car le poids du sommet `0` additionné au poids de l'arc `(0,2)` donne $0 + 4 = 4 = 4$ et le sommet `2` a déjà un poids égal à 4 ;
– l'arc `(2,1)` et nous ne changeons pas le poids du sommet `1` car le poids du sommet `2` additionné au poids de l'arc `(2,1)` donne $4 - 2 = 2$ et le sommet `1` a déjà un poids égal à 2.

Ainsi, le plus court chemin entre le sommet `0` et le sommet `1` est de longueur 2 et passe par le sommet `2` ; et le plus court chemin entre le sommet `0` et le sommet `2` est de longueur 4.

Remarquons que si nous obtenons dans cet exemple les plus courts chemins dès la fin du premier tour de la boucle, cela est dû à l'ordre dans lequel nous avons considéré les différents arcs. Si nous avions considérés les arcs dans l'ordre suivant `(0,1)`, `(2,1)` et `(0,2)`, nous aurions alors eu besoin des deux tours pour retrouver les plus courts chemins.

2.6 Dernières remarques

Nous terminerons ce chapitre par une définition et une dernière remarque.

Le couple de sommets dans un digraphe qui est tel que le chemin entre les sommets de ce couple est le plus grand du digraphe définit le diamètre du digraphe. La valeur de ce diamètre est donc la longueur de ce plus long chemin.

Remarquons enfin que si un DAG possède plusieurs sommets initiaux sans prédécesseur, nous pouvons le modifier en lui ajoutant une source unique qui a comme successeurs les sources initiales et dont les arcs ainsi ajoutés sont de poids nuls.