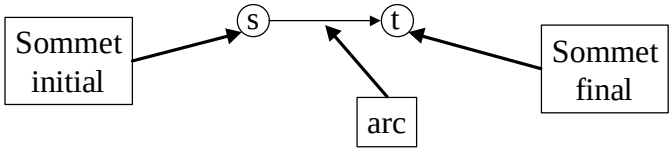
CH.2 GRAPHES

- 2.1 La terminologie
- 2.2 La représentation des graphes
- 2.3 Le plus court chemin
- 2.4 Le parcours en profondeur
- 2.5 Les graphes sans circuit
- 2.6 Les graphes non orientés

2.1 La terminologie

Permettent de représenter des relations quelconques entre des objets.

Sommets ou noeuds, reliés par des *arcs* (orientés).



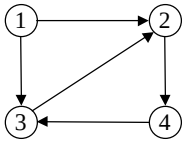
Deux sommets reliés sont *adjacents*.
Nombre d'arcs quittant un sommet : *degré sortant*.
Nombre d'arcs arrivant à un sommet : *degré entrant*.

Les sommets et les arcs peuvent porter des informations (étiquettes).

Chemin d'un sommet à un autre, dont la longueur est le nombre d'arcs.
Circuit : chemin revenant à son point de départ.
Chemin simple : aucun sommet n'est répété (sauf pour un circuit).

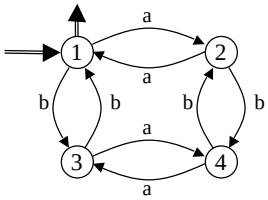
Propriétés :
1. Si deux sommets sont reliés par un chemin, ils sont reliés par un chemin simple ;
2. Si un graphe a n sommets, un chemin simple est de longueur au plus n – 1 si l'origine et l'extrémité sont différentes ; un circuit simple est de longueur au plus n ;
3. Conséquence : si deux sommets différents sont reliés par un chemin, alors ils sont reliés par un chemin de longueur au plus n.

Exemple :



Utilisation pour modéliser des réseaux de communication. Dans ce cas, les étiquettes peuvent porter un distance, un temps, un coût, ...

On peut aussi modéliser un système de transition entre états (automate).



Une suite de a et de b est la suite des étiquettes d'un chemin de 1 vers 1 si et seulement si cette suite contient un nombre pair de a et un nombre pair de b.

2.2 La représentation des graphes

Représentation par matrice d'adjacence :

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

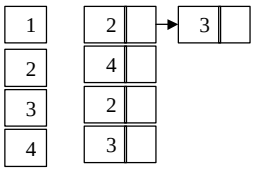
= M

On lit dans la matrice le nombre d'arcs, les degrés entrants et sortants (attention aux boucles !)
Dans M^n, l'élément (s,t) est le nombre de chemins de s à t.

Cette matrice peut aussi comporter les grandeurs (distances, coûts) s'il y a lieu à la place du 1.

Inconvénient : si le graphe a peu ($<< n^2$) d'arcs, la matrice est creuse. Mais sa consultation est en $O(n^2)$ malgré tout.

On peut dans ce cas représenter un graphe par la liste des successeurs. (*cf.* arbres). Les informations supplémentaires peuvent être stockées dans les structures.



La taille requise est le nombre d'arcs. Les degrés sortants sont visibles, mais pas les degrés entrants. Il peut être utile parfois de déduire la représentation par liste des prédécesseurs de celle-ci (exercice).

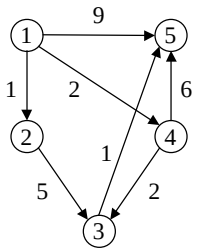
La complexité des algorithmes sur les arbres dépend donc de la façon dont ceux-ci sont représentés.

Un certain nombre d'algorithmes reposent sur l'algorithme d'exploration en profondeur, semblable à celui des arbres et sur les ordres préfixe et postfixe.
Un autre algorithme d'exploration est étudié d'abord.

2.3 Le plus court chemin

Le graphe considéré a ses arcs étiquetés par une grandeur positive qu'on imaginera comme leur longueur. La longueur d'un chemin est donc évidemment la somme des longueurs des arcs du chemin.

Soit un sommet donné s. Il s'agit de trouver le plus court chemin de s à chacun des autres sommets du graphe.



Le plus court chemin de 1 à 5 passe par 4 puis 3.

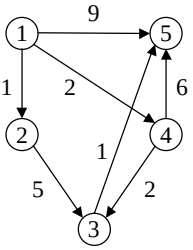
L'algorithme de Dijkstra permet de trouver tous ces chemins. On cherche à chaque étape la meilleure solution. Par chance, cela donne la meilleure solution globale.

On a affaire à un *algorithme glouton*.

L[i, j] est la longueur de l'arc de i à j et ∞ si i et j ne sont pas reliés.

On part du sommet 1 et on note D[i] la longueur du plus court chemin trouvé à un instant donné.

Un ensemble E contient les sommets pour lesquels on a trouvé le plus court chemin définitif. Au départ, E ne contient que 1. A chaque itération, un sommet est ajouté à E et les distances D[i] sont actualisées.



Itér.	E	t	D[2]	D[3]	D[4]	D[5]
0	{1}	—	1	∞	2	9
1	{1,2}	2	1	6	2	9
2	{1,2,4}	4	1	4	2	8
3	{1,2,4,3}	3	1	4	2	5
4	{1,2,4,3,5}	5	1	4	2	5

A chaque itération, on choisit le sommet t qui a le D[t] minimum, puis on réévalue les D[i] susceptibles des successeurs de t.

Pour reconstituer le chemin, on peut tenir à jour un tableau C de prédécesseurs actualisé lorsque D[i] est changé.

```
fonction DIJKSTRA()
{
   E = {1};                                /**********************/
   pour chaque sommet i ≠ 1 faire          /* Initialisation     */
   {                                       /*                    */
      C[i] = 1; D[i] = L[1, i];            /*                    */
   }                                       /**********************/
   pour k = 1 à n - 1 faire                /**********************/
   {                                       /* Iterations         */
      t = sommet ∉ E tel que D[t] est minimum;/*                 */
      E = E ∪ {t};                         /* Sommet ajoute a E  */
      pour chaque i ∉ E successeur de t faire/*                  */
      si (D[t] + L[t, i] < D[i]) alors     /* Actualisation de D */
      {                                    /*                    */
         D[i] = D[t] + L[t, i];            /*                    */
         C[i] = t;                         /* Actualisation de C */
      }                                    /*                    */
    }                                      /**********************/
}
```


Complexité de l'algorithme :

Si le graphe a n sommets et a arcs, l'initialisation est en O(n). Il y a n-1 itérations. La recherche du minimum est en O(n) chaque fois, soit $O(n^2)$ en tout

Quant à l'actualisation, elle nécessite au total l'examen de tous les arcs, soit a ; l'actualisation d'une valeur est en temps constant d'où O(a).

Puisque typiquement $n < a < n^2$, la complexité finale est $O(n^2)$.

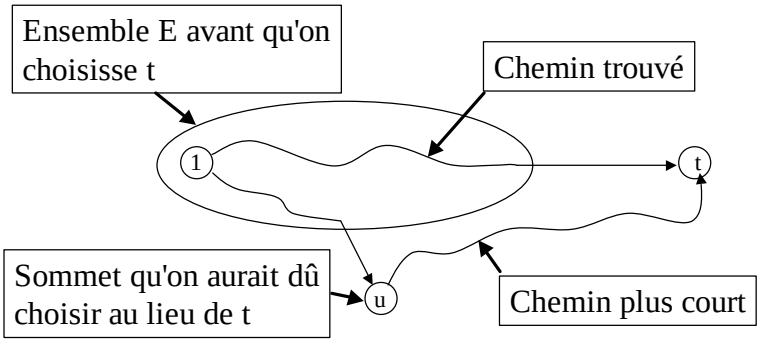
Ceci est vrai si le graphe est représenté par un tableau des successeurs.

On peut représenter l'ensemble des sommets et les valeurs des tableaux par une autre structure (file de priorité) pour laquelle la recherche du minimum est en O(log n) ainsi que l'actualisation d'une valeur. On a ainsi une complexité O(a log n), bien meilleure si $a < n^2$.

Justification de l'algorithme :

La justification n'est généralement pas indispensable pour l'implémentation ou la compréhension d'un algorithme.

La démonstration peut être schématisée. Supposons que le chemin trouvé de 1 à un sommet t ne soit pas le plus court, et prenons le t le plus proche de 1 pour lequel cela se produit :



2.4 Le parcours en profondeur

S'il on veut explorer un graphe à partir d'un sommet donné (on n'atteindra évidemment que les sommets reliés à celui d'où on part), on peut évidemment appliquer l'algorithme de Dijkstra en mettant à 1 la longueur de chaque arc.

Mais si on veut seulement parcourir tous les sommets atteignables, on peut le faire avec une meilleure complexité par un parcours en profondeur. On procède comme pour les arbres, mais du fait que les sommets peuvent être atteints à partir de plusieurs sommets (et non plus à partir du seul père), on marque les sommets déjà visités pour ne pas les visiter de nouveau.

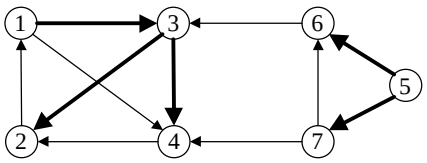
On fabrique ainsi un arbre contenant tous les sommets atteignables à partir du sommet de départ, dont les arcs sont certains des arcs du graphe.

```
fonction INITIALISE_EXPLORATION()
{
   pour chaque sommet t faire explore[t] = 0;
}
fonction EXPLORATION(sommet s)
{
   si (explore[s] = 0) alors
   {
      explore[s] = 1;
      pour chaque successeur t de s faire EXPLORATION(t);
   }
}
fonction PROFONDEUR(sommet s)
{
   INITIALISE_EXPLORATION();
   EXPLORATION(s);
}
```

La fonction principale est essentiellement la même que pour l'exploration des arbres.

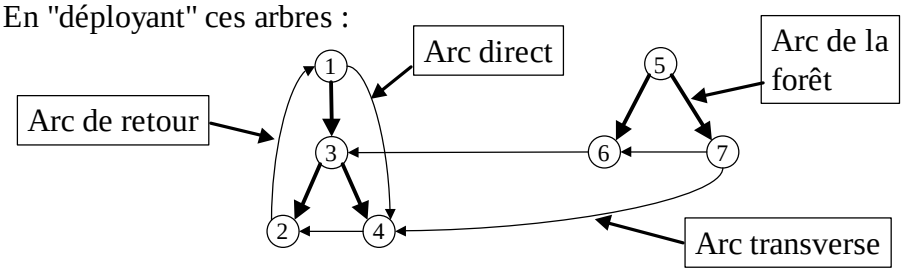
Si on veut visiter tous les sommets, on peut modifier ainsi :


```
fonction DESCRIPTION()
{
   INITIALISE_EXPLORATION();
   pour chaque sommet s faire EXPLORATION(s);
}
```



Si on prend les sommets dans l'ordre croissant, on obtient ici deux arbres, enracinés en 1 et en 5.

En "déployant" ces arbres :



On attribue à chaque sommet un numéro d'ordre (c'est l'ordre préfixe des arbres) :

```
fonction INITIALISE_EXPLORATION()
{
   pour chaque sommet t faire explore[t] = 0;
   compteur = 0;
}
fonction EXPLORATION(sommet s)
{
   si (explore[s] = 0) alors
   {
      explore[s] = 1; ordre[s] = compteur++;
      pour chaque successeur t de s faire EXPLORATION(t);
   }
}
fonction DESCRIPTION()
{
   INITIALISE_EXPLORATION();
   pour chaque sommet s faire EXPLORATION(s);
}
```


Propriété : les arcs transverses relient un sommet à un sommet de numéro plus petit.

Cette propriété est utile pour montrer la validité de certains algorithmes.

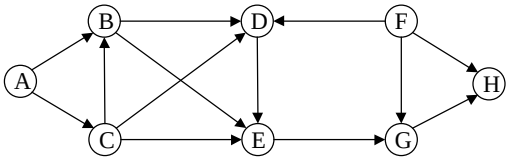
Elle servira en particulier dans la section suivante.

Complexité de l'exploration en profondeur :

Pour chaque sommet, on examine ses successeurs, puis on ne revient pas à ce sommet. On examine chaque arc une fois et une seule. La complexité est donc O(a).

2.5 Les graphes sans circuit

La propriété de n'avoir aucun circuit est indispensable dans certains cas. Par exemple, pour les graphes de priorité.
Certaines tâches doivent être exécutées avant d'autres. Pour qu'une exécution soit possible, il est indispensable qu'aucun circuit n'apparaisse.
Une fois ceci vérifié, il faudra trouvé un ordre possible d'exécution. C'est ce qu'on appelle le *tri topologique*.



Ici, un ordre possible est : A C B F D E G H

Pour tester si un graphe possède un circuit, on utilise une exploration en profondeur modifiée.

Proposition : un graphe possède un circuit si et seulement si, dans une exploration en profondeur, il existe un arc de retour.

S'il existe un arc de retour, il y a bien entendu un circuit :

Supposons qu'il y a un circuit. Nous allons montrer qu'il existe un arc de retour.
Soit un tel circuit, et soit s le sommet de ce circuit qui a le plus petit numéro et soit t son prédécesseur sur ce circuit. Tous les sommets de ce circuit ont donc un numéro plus grand que celui de s.
Si tous sont descendants de s, c'est aussi le cas de t, donc ts est un arc de retour.

Si tous les sommets du circuit ne sont pas descendants de s, soit u le premier dont le successeur v n'est pas descendant de s.

Le sommet v a un numéro plus grand que celui de s. Il est donc visité après s. Mais ce n'est pas un descendant de s. Ce n'est donc pas non plus un descendant de u. Si son numéro était plus petit que celui de u, il serait visité entre s et u, c'est-à-dire pendant que sont visités les descendants de s, ce qui est contraire à l'hypothèse. Le numéro de v est donc plus grand que celui de u. L'arc uv est donc un arc qui n'est pas direct et qui relie un sommet à un autre de numéro plus grand. Ceci aussi est impossible.

Donc tous les sommets du circuit sont descendants de s, et il existe un arc de retour.

L'algorithme consiste donc à chercher s'il existe un arc de retour dans l'exploration en profondeur du graphe.

On procède par marquage, mais les sommets peuvent avoir trois états :

- *avant* la première visite, le sommet est *blanc* ;
- *pendant* qu'on explore ses descendants, le sommet est *gris* ;
- *après* la dernière visite, le sommet est *noir*.

Quand on examine un arc dans le parcours en profondeur,

lorsqu'il s'agit d'un arc qui sera dans la forêt, son extrémité est encore blanche ;

lorsqu'il s'agit d'un arc direct ou d'un arc transverse, son extrémité déjà est noire ;

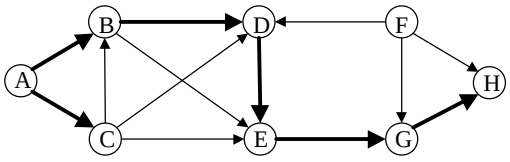
lorsqu'il s'agit d'un arc de retour, son extrémité est grise.

D'où l'algorithme :

```
fonction INITIALISE_TEST_CIRCUIT()
{
   pour chaque sommet t faire explore[t] = 0;/* sommet blanc */
   trouve = 0;
}
fonction TESTE_ARC_RETOUR(sommet s)
{
   si (explore[s] = 0) alors           /* si s est blanc */
   {
      explore[s] = 1;                  /* il devient gris */
      pour chaque successeur t de s tant que trouve = 0 faire
      {
         si (explore[t] = 1) alors trouve = 1;
         sinon TESTE_ARC_RETOUR(t);
      }
      explore[s] = 2; ecrire(s);       /* il devient noir */
   }                                   /* ecrire : tri topologique */
}
fonction TESTE_CIRCUIT()
{
   INITIALISE_EXPLORATION();
   pour chaque sommet s tant que trouve = 0 faire TESTE_ARC_RETOUR(s);
}
```


Le même algorithme permet d'effectuer un *tri topologique*.
Un tri topologique est un ordre dans lequel, s'il existe un arc de s vers t, alors s reçoit un numéro d'ordre plus petit que t.
Considèrons les sommets écrits dans l'ordre postfixe d'exploration en profondeur dans un graphe sans circuit.
Les sommets sont alors numérotés dans l'ordre où ils deviennent noirs.
L'ordre *inverse* est un tri topologique des sommets.
Il faut vérifier que, s'il existe un arc de s vers t, alors le numéro de s est *plus grand* que celui de t.
Or dans un graphe sans circuit on a :
• des arcs de la forêt, connectant un père à son fils ;
• des arcs directs, connectant un sommet à un descendant ;
• des arcs transverses connectant un sommet encore gris à un déjà noir.
Dans ces trois cas, la condition d'ordre est satisfaite.

Ceci est réalisé en même temps que le test par une écriture.



L'ordre postfixe est ici : H G E D B C A F, correspondant à l'ordre topologique F A C B D E G H.

2.6 Les graphes non orientés

Certains problèmes ne font pas intervenir l'orientation des arcs.

On a alors affaire à un *graphe non orienté*. Une autre façon de voir un tel graphe est d'imaginer qu'un arc st implique l'existence d'un arc ts. Dans la structure de données par liste de successeurs, il faut donc vérifier ces conditions.

Les problèmes classiques sont la recherche d'un arbre recouvrant de poids minimum, la recherche de la connexité (simple), de la connexité multiple, des points d'articulation, des circuits eulériens.
Les problèmes précédents, applicables par exemple aux réseaux, sont résolubles par des algorithmes polynomiaux.

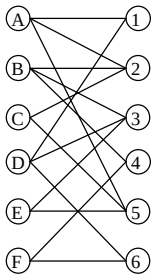
D'autres problèmes de coloriage, de recherche de circuits hamiltoniens (voyageur de commerce), ne sont résolubles que par des algorithmes

exponentiels, donc quasiment impossibles à résoudre de façon pratique. On doit donc, dans ce cas, recourir à des algorithmes approchés, fondés sur des hypothèses raisonnables (heuristiques) ou sur des méthodes de perturbations (recuit simulé).

Nous allons étudier un problème d'appariement qui, dans le cas général, est de complexité exponentielle mais qui, dans le cas particulier des *graphes bipartis*, est polynomial.

On considère deux ensembles de sommets et on suppose que les arêtes relient des sommets de l'un à des sommets de l'autre.

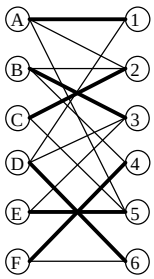
On peut voir les premiers comme des personnes, les seconds comme des tâches et les arêtes comme des capacités d'affectation des personnes aux tâches.



Est-il possible d'affecter toutes les personnes à toutes les tâches ?

Sinon, quel est le nombre maximum d'affectations pouvant être réalisées ?

Un placement mal commencé peut aboutir à un blocage :
A-1, B-2, C-5 (forcé), D-3, impossible pour E.



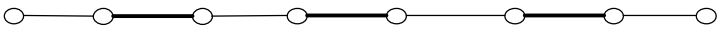
On peut affecter toutes les personnes, à condition de ne pas affecter B à 2. Il y a plusieurs solutions.

L'algorithme requiert un peu de terminologie.
Une affectation (choix d'arêtes) est un *couplage*. Il est caractérisé par le fait que deux arêtes choisies n'ont pas d'extrémité commune. Les arêtes choisies seront appelées *rouges*, les autres *bleues*.
On cherche à réaliser un *couplage maximum*.
Un sommet est *affecté* lorsqu'il est extrémité d'une arête choisie. Sinon il est *libre*.
Les sommets du premier ensemble sont des *lettres*, les autres des *chiffres*.

Une chaîne améliorante pour un couplage donné (non maximum) est un chemin dont les extrémités, une lettre et un chiffre, sont libres, et dont les arêtes sont alternativement bleues et rouges. La première et la dernière arête sont donc bleues. Un telle chaîne est de longueur impaire, contenant k arêtes rouges et k + 1 arêtes bleues.

Si un couplage possède une chaîne améliorante, il peut être amélioré.

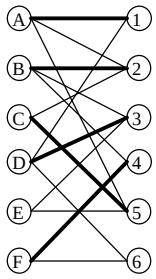
Il suffit d'intervertir arêtes rouges et bleues sur la chaîne. Le nombre total d'arêtes rouges a augmenté d'une unité. Deux sommets auparavant libres sont devenus affectés. Les sommets intermédiaires sont restés affectés.



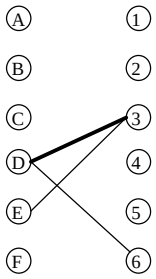
Chaîne améliorante avant l'amélioration

Chaîne améliorante après l'amélioration

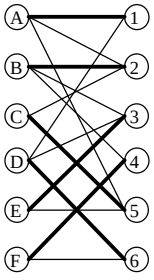
Dans notre exemple, on obtient la configuration suivante :



Chaîne améliorante :



Couplage amélioré :

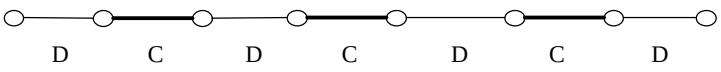


On a montré :
tout couplage ayant une chaîne améliorante peut être amélioré.

On a aussi la réciproque :
tout couplage non maximum possède une chaîne améliorante.

Démonstration (combinatoire) :
Soit C un couplage et D un couplage maximum. Supposons que C soit moins bon que D. Nous allons trouver une chaîne améliorante pour C.

Considérons le graphe G = C ⊕ D. C'est-à-dire que les arêtes de G sont celles de C pas dans D et celles de D pas dans C. D'un sommet peut partir : aucune arête, une et une seule arête de C, une et une seule arête de D ou une arête de chaque
Les composantes connexes de G sont donc soit des cycles où les arêtes de C et de D alternent, soit des chaînes où elles alternent également.
Mais les arêtes de G provenant de D sont strictement plus nombreuses que celles provenant de C. Donc il existe dans G au moins une chaîne contenant plus d'arêtes de D que de C. Cette chaîne est de la forme :



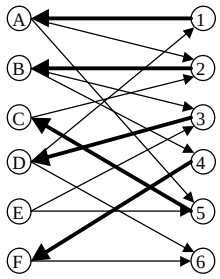
C'est une chaîne améliorante de C.

L'algorithme de recherche d'un couplage maximum sur un graphe biparti utilise la propriété précédente.

- Partir d'un couplage quelconque (éventuellement nul !) ;
- Chercher une chaîne améliorante pour ce couplage ;
- Améliorer le couplage grâce à cette chaîne ;
- Recommencer jusqu'à ce qu'on ne trouve plus de telle chaîne.

Le point est donc de trouver une (éventuelle) chaîne améliorante.

Pour cela, à partir d'un couplage C déjà construit, on fabrique un graphe orienté en orientant des lettres vers les chiffres les arêtes qui ne sont pas dans C et des chiffres vers les lettres les arêtes qui sont dans C. Puis on cherche, dans ce graphe, un chemin allant d'une lettre libre à un chiffre libre. Un tel chemin est exactement une chaîne améliorante. On la trouve donc par une exploration de graphe en profondeur.



On cherche un chemin de E vers 6. L'exploration en profondeur donne par exemple :
E-3-D-6 (chemin choisi précédemment), ou bien :
E-3-D-1-A-2-B-4-F-6 ou d'autres.

Si le graphe comporte 2n sommets et a arêtes, la complexité de chaque recherche de chaîne est O(a), complexité de l'exploration en profondeur.

Chaque chaîne améliore de 1 le couplage. Un couplage maximum contient au plus n arêtes, donc nécessite n itérations de recherche de chaîne améliorante.

La complexité globale de cet algorithme est donc O(na).