GRAPHES (éléments)

Table des matières

1 Introduction aux graphes

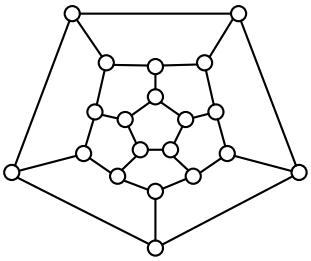

FIGURE 1 – Graphe simple non orienté (voyage autour du monde de Hamilton).

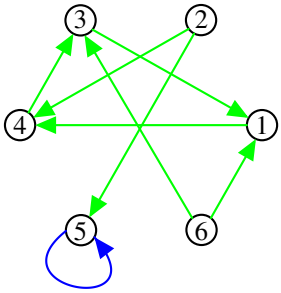

FIGURE 2 – Graphe simple, orienté, avec une boucle.

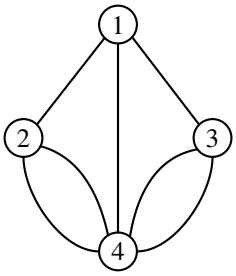

FIGURE 3 – Multi-graphe, non orienté (Euler).

1.1 Graphe simple orienté

Un ***graphe*** (simple) ***orienté***, est un couple $G = (S,A)$ où

- S est l'ensemble des **sommets** ;
- A est l'ensemble des arcs, un ***arc*** étant un couple de sommets (u,v), arc de u vers v, parcouru uniquement dans le sens de u vers v ($u \longrightarrow v$). Un arc (u,u) (de u vers u) est qualifié de ***boucle***.

Le graphe peut être ***pondéré*** (valué) en associant à chaque arc un ***poids*** (ou valuation)

- Pour des raisons de simplicité (homogénéité des algorithmes), on ne considérera que des graphes pondérés, quitte à associer artificiellement le poids 1 à chacun des arcs.

1.2 Graphe simple non orienté

Un ***graphe*** (simple) ***non orienté***, est un couple $G = (S,A)$ où

- S est l'ensemble des ***sommets*** ;
- A est l'ensemble des arêtes, une ***arête*** étant un ensemble de deux sommets $\{u,v\}$, parcourue indifféremment dans le sens u vers v ou v vers u. Une arête réduite à un élément ($\{u,u\} = \{u\}$) est qualifié de ***boucle***.

Le graphe peut être ***pondéré*** (valué) en associant à chaque arête un ***poids*** (ou valuation).

- Pour des raisons de simplicité (homogénéité des algorithmes), on ne considérera que des graphes pondérés, quitte à associer artificiellement le poids 1 à chacun des arcs.
- **On transformera une arête $\{u,v\}$ ($u \neq v$), de poids p, en deux arcs orientés (u,v) et (v,u), chacun de poids p, et une boucle $\{u\}$ de poids p en le seul arc (u,u), de poids p. ce qui assimile un graphe non orienté à un graphe (bi-)orienté symétrique**
  - on peut ainsi préciser un ordre de parcours éventuel, $u \longrightarrow v$ ou $v \longrightarrow u$.
  - les algorithmes traitant des graphes orientés sont applicables aux graphes non orientés.

  Inconvénients :
  - La succession de deux arêtes consécutives, $u \longrightarrow w \longrightarrow v$, peut se décrire par la liste $[(w,u),(v,w)]$ alors que l'on préfererait avoir la liste $[(u,w),(w,v)]$. Solution : introduire un "petit" algorithme de rectification, qui transforme les listes du genre $[(w,u),(v,w)]$ en des listes du genre $[(u,w),(w,v)]$.
  - Un test d'égalité entre deux arêtes doit tenir compte de la possibilité d'inversion : $(u,v) = (v,u)$.

1.3 Multigraphe

Un ***multigraphe***, ou ***multi-graphe***, est une extension de la notion de graphe, où il peut y avoir plusieurs arcs ou arêtes entre deux sommets. Il faut alors accompagner un arc ou une arête d'un élément d'identification (par exemple numéro d'arc ou d'arête) permettant de distinguer entre les différents arcs ou arêtes entre deux sommets.

Un ***p-graphe*** est un multi-graphe dans lequel il y a au plus p arcs d'un sommet à un autre (ou p arêtes entre deux sommets)

Un graphe simple est donc un 1-graphe.
Sauf mention contraire, le mot "graphe" désigne un graphe simple.

1.4 Chemins et circuits, chaînes et cycles

Dans un graphe orienté,

- un ***chemin***, de ***longueur*** $k \geqslant 0$ (ayant k arcs), du sommet x_0 au sommet x_k, passant par les sommets successifs $x_0, x_1, \ldots, x_k$, est une séquence de k arcs concécutifs, $(x_0,x_1), \ldots, (x_i,x_{i+1}), (x_{i+1},x_{i+2}), \ldots, (x_{k-1},x_k)$ [a] .
- un ***circuit*** est un chemin $(x_0,x_1), \ldots, (x_i,x_{i+1}), (x_{i+1},x_{i+2}), \ldots, (x_{k-1},x_k)$ où $x_k = x_0$.

Un chemin, un circuit qui ne passe pas deux fois par un même sommet est qualifié d'***élémentaire***.
Un chemin, un circuit qui ne passe pas deux fois par un même arc est qualifié de ***simple***.

a. Dans le cas d'un multi-graphe, il faut accompagner chaque arc de son identifiant ou alors réduire un arc à son identifiant.

Dans un graphe non orienté,

- Une ***chaîne*** de ***longueur*** $k \geqslant 0$ (ayant k arêtes), du sommet x_0 au sommet x_k, passant par les sommets successifs $x_0, x_1, \ldots, x_k$, est une séquence de k arêtes concécutives, $(x_0,x_1), \ldots, (x_i,x_{i+1}), (x_{i+1},x_{i+2}), \ldots, (x_{k-1},x_k)$ [a] .
- un ***cycle*** est une chaîne $(x_0,x_1), \ldots, (x_i,x_{i+1}), (x_{i+1},x_{i+2}), \ldots, (x_{k-1},x_k)$ où $x_k = x_0$.

Un chaîne ou un cycle qui ne passe pas deux fois par un même sommet est qualifié(e) d'***élémentaire***.
Un chaîne ou un cycle qui ne passe pas deux fois par une même arête est qualifié(e) de ***simple***.

a. Dans le cas d'un multi-graphe, il faut accompagner chaque arête de son identifiant ou alors réduire une arête à son identifiant.

Remarques.

- Dans les graphes simples, on peut aussi caractériser les chaînes, chemins, cycles ou circuits uniquement par la séquence des sommets successifs $x_0, x_1, \ldots, x_k$.
- Une chaîne, de x_0 à x_k, peut être représentée indifféremment par les séquences d'arêtes :

  $(x_0,x_1), (x_1,x_2), \ldots, (x_i,x_{i+1}), (x_{i+1},x_{i+2}), \ldots, (x_{k-1},x_k)$ (forme "naturelle")

  $(x_1,x_0), (x_2,x_1), \ldots, (x_i,x_{i+1}), (x_{i+2},x_{i+1}), \ldots, (x_k,x_{k-1})$ etc ...

  On pourra mettre en place une transformation qui aboutisse à la forme "naturelle".
- Un test d'égalité entre deux arêtes doit tenir compte de la possibilité d'inversion : $(u,v) = (v,u)$.

1.5 Connexité

- Un graphe non orienté est ***connexe*** si tout couple de sommets est reliés par une chaîne.
  Un graphe orienté sera ***connexe*** si le graphe obtenu en supprimant l'orientation est connexe.
- Une ***composante connexe*** d'un graphe non orienté est un sous-graphe connexe maximal.
- Un graphe orienté est ***fortement connexe*** si chaque sommet est accessible à partir d'un sommet quelconque.
- Une ***composante fortement connexe*** d'un graphe orienté est un sous-graphe fortement connexe maximal.

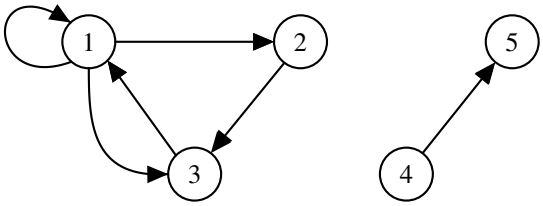

FIGURE 4 – Graphe orienté, avec boucle et circuits, de composantes fortement connexes $\{1,2,3\}$, $\{4\}$ et $\{5\}$.

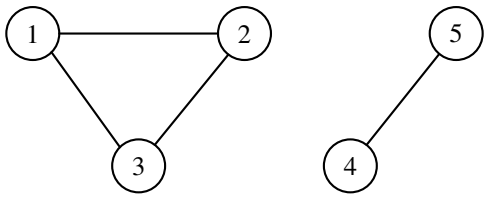

FIGURE 5 – Graphe non orienté, de composantes connexes $\{1,2,3\}$ et $\{4,5\}$, (déduit de celui de la Figure 4 par suppression de l'orientation et des boucles).

1.6 Arbre, arbre couvrant

Un ***arbre*** est un graphe non orienté, non pondéré, acyclique et connexe.

- les sommets de degré 1 sont appelés des ***feuilles*** (ou nœuds "externes")
- les sommets de degré supérieur à 1 sont appelés des ***nœuds*** (avec éventuellement la précision : ***"internes"***)
- les arêtes sont appelées des ***branches***

Si on choisit un sommet r quelconque dans un arbre, il est possible "d'enraciner" l'arbre en r, c'est-à-dire d'orienter toutes les arêtes de sorte qu'il existe un chemin de r à tous les autres sommets. On obtient alors un ***arbre enraciné***, de ***racine*** r.

Un graphe constitué de la réunion d'arbres distincts est une ***forêt***.

On appelle ***arbre couvrant***, d'un graphe $G = (S,A)$ non orienté connexe, tout graphe (S',A') connexe, sans cycle, tel que $S' = S$ et $A' \subseteq A$.

2 Codage des graphes finis

Un graphe est "fini" si les algorithmes de parcours élémentaires permettent d'accéder en un temps acceptable à l'ensemble de ses sommets et de ses arcs, sinon il est (quasiment) "infini".

Pour simplifier le codage, il est préférable d'avoir des sommets qui sont (associés à) des numéros de 1 à n, les arcs étant accompagnés de leur pondération (et de leur identifiant éventuel).
On peut associer, de façon externe au graphe, des informations complémentaires relatives aux sommets et aux arcs.
Par exemple, dans un graphe représentant une carte routière,

- les sommets sont des numéros de ville, et on associe un "nom de ville" à chaque numéro ;
- les arcs sont les liaisons pondérées par la distance entre (numéros de) villes voisines, et on associe un qualificatif ("itinéraire jaune, vert, ...") à chaque arc.

Dans la suite,

- les sommets sont numérotés de 1 à n.
- Un graphe non pondéré est pondéré artificiellement par 1.
- Un graphe non orienté est codé comme un graphe (bi-)orienté symétrique.
  La seule différence entre un arc et une arête sera dans le traitement et l'interprétation : sommets extrémités interprétés dans l'ordre pour un graphe orienté et dans un ordre indifférent pour un graphe non orienté.

2.1 Codage brut par la liste des arcs

Un arc (arête) est codé comme un couple $((x,y),p)$ ou $((x,y),(p,k))$, avec

- x et y les sommets extrémités (origine et extrémité) de l'arc (arête),
- p la pondération de l'arc (arête),
- k identifiant (numéro) de l'arc (arête), dans le cas d'un multi-graphe.

Un multi-graphe orienté est codé comme la liste "brute" [a] de ses arcs, de type (par exemple) :

```
((int * int) * (int * int)) list
```

Un chemin est codé comme une liste d'arcs consécutifs [b], premier arc en tête de liste.

a. Ce n'est pas le codage le plus pertinent, il est même **très mauvais**.
b. Un chemin est alors un (sous-)graphe ("ordonné"), seul avantage de ce codage.

2.2 Codage par liste (ou vecteur) de listes d'adjacence

– La liste **d'adjacence d'un sommet** u est la liste des couples (v,p_{uv}) [a] tels qu'il existe un arc (u,v) : $u \longrightarrow v$ de pondération p_{uv}.
– Un graphe est codé comme la liste (ou le vecteur indicé par les numéros de sommet) des couples (sommet s, liste d'adjacence de s) : **liste d'adjacence** ou **vecteur d'adjacence** du **graphe**.

a. Couples $(v,(p_{uv},k))$, où k est un identifiant d'arc, dans le cas d'un multi-graphe.

Remarque. Pour un graphe non orienté, codé comme un graphe orienté symétrique, si v est dans la liste d'adjacence de u, u est également dans la liste d'adjacence de v, avec la même pondération (il y a redondance).
Dans le cas d'une boucle (u,u), u est dans sa propre liste d'adjacence.

Exemple

- Graphe de la Figure 1 (Hamilton), artificiellement pondéré par 1 :

```
[1,[ 2,1;  5,1;  8,1];  2,[ 1,1;  3,1; 10,1];  3,[ 2,1;  4,1; 12,1];  4,[ 3,1;  5,1; 14,1]; 5,[ 1,1;  4,1;  6,1];
 6,[ 5,1;  7,1; 15,1];  7,[ 6,1;  8,1; 17,1];  8,[ 1,1;  7,1;  9,1];  9,[ 8,1; 10,1; 18,1]; 10,[ 2,1;  9,1; 11,1];
 11,[10,1; 12,1; 19,1]; 12,[ 3,1; 11,1; 13,1]; 13,[12,1; 14,1; 20,1]; 14,[ 4,1; 13,1; 15,1]; 15,[ 6,1; 14,1; 16,1];
16,[15,1; 17,1; 20,1]; 17,[ 7,1; 16,1; 18,1]; 18,[ 9,1; 17,1; 19,1]; 19,[11,1; 18,1; 20,1]; 20,[13,1; 16,1; 19,1] ]
```

Inconvénients :

- obligation de parcours systématique des listes d'adjacence pour obtenir les arcs.
- si le graphe est orienté, non symétrique, il n'est pas immédiat d'obtenir les arcs d'extrémité v.

Avantages :

- on ne stocke que la partie utile de l'information (sauf si l'arête $\{x,y\}$ pondérée par p est représentée par deux arcs : (x,p) est dans la liste d'adjacence de y et (y,p) est dans la liste d'adjacence de x) ;
- pas d'obligation de numéroter les sommets de façon consécutive (mais il est préférable de le faire) ;
- possibilité d'ordonnancement selon divers critères et introduction aisée de nouveaux sommets ou arcs.

2.3 Codage par matrice d'adjacence (sommets numérotés de 1 à n).

La **matrice d'adjacence** d'un graphe simple [a] est une matrice M, carrée, d'indices $0, 1, \ldots, n$, d'ordre $n+1$, telle que, pour $i, j \in \{1, \ldots, n\}$, $M(i,j)$ contienne l'information (existence, pondération) sur l'arc reliant i à j :

- $M(i,0)$ et $M(0,j)$ sont non utilisés ou prennent la valeur du sommet : matrice "bordée" (lisibilité de la matrice) ;
- Pour $i \neq j$, $\begin{cases} M(i,j) = \text{poids de l'arc } (i,j) \text{ s'il existe un arc de } i \text{ à } j, \\ M(i,j) = \infty \text{ ("valeur impossible") s'il n'existe pas d'arc de } i \text{ à } j \end{cases}$
- $M(i,i) = \infty$ (ou $M(i,i) = 0$) s'il n'existe pas de boucle de i à i.

si les poids sont des entiers, ∞ peut être représenté par `max_int`

a. Pour un multi-graphe, il faudrait que $M(i,j)$ puisse contenir la liste des (poids, identifiant) de tous les arcs allant de i à j.

La matrice d'adjacence d'un graphe simple non orienté est symétrique, et on pourrait n'en coder que la partie supérieure.

Exemples.

- Le graphe de la figure Figure 1 (Hamilton), artificiellement pondéré, est représenté par la matrice (bordée) :

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	0	1	∞	∞	1	∞	∞	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
2	1	0	1	∞	∞	∞	∞	∞	∞	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
3	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞	1	∞	∞	∞	∞	∞	∞	∞	∞	∞
4	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞	∞	∞	∞	∞	∞
5	1	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
6	∞	∞	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞	∞	∞	∞
7	∞	∞	∞	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞	∞
8	1	∞	∞	∞	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
9	∞	∞	∞	∞	∞	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞
10	∞	1	∞	∞	∞	∞	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
11	∞	∞	∞	∞	∞	∞	∞	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞	1	∞
12	∞	∞	1	∞	∞	∞	∞	∞	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞	∞
13	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	1
14	∞	∞	∞	1	∞	∞	∞	∞	∞	∞	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞
15	∞	∞	∞	∞	∞	1	∞	∞	∞	∞	∞	∞	∞	1	0	1	∞	∞	∞	∞	∞
16	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	1	0	1	∞	∞	∞	1
17	∞	∞	∞	∞	∞	∞	1	∞	∞	∞	∞	∞	∞	∞	∞	1	0	1	∞	∞	∞
18	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞	∞	∞	∞	∞	∞	1	0	1	∞	∞
19	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞	∞	∞	∞	∞	1	0	1	∞
20	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞	1	∞	∞	1	0	1

avec $M(i,i) = 0$ lorsqu'il n'existe pas de boucle $i \longrightarrow i$.

-

Avantages :

- accès direct à l'information, à coût constant,
- on trouve facilement les arcs d'origine u,
- on trouve facilement les arcs d'extrémité v.

Inconvénients :

- si le graphe comporte beaucoup de sommets et peu d'arcs, ce qui est fréquent, la matrice est "creuse" et on réserve beaucoup d'espace pour peu de données ;
- la représentation à l'aide d'une matrice se prête mal à l'introduction dynamique de nouveaux sommets.

Remarque. On se servira peu de la matrice d'adjacence, d'autant plus que cela peut poser quelques problèmes (complications, acrobaties) selon le choix fait pour le codage de l'absence de boucle $i \longmapsto i$: 0 ou ∞.

2.4 Critères de choix pour le codage

- Certains algorithmes sont plus faciles à réaliser avec une représentation qu'avec d'autres ou spécifiques à une représentation.
- Pour un graphe creux, l'utilisation d'une matrice normale gaspille beaucoup de place.
- Le temps d'exécution peut être plus court avec la représentation matricielle (coût d'accès constant).
- Si un graphe doit subir des modifications (ajouts de sommets), l'utilisation de listes d'adjacence est plus judicieux.

3 Graphes de "grande" taille, fini ou infini (dénombrable)

Dans une telle situation, il n'est pas possible de stocker de façon exhaustive tous les sommets et tous les arcs du graphe. Par exemple, le "jeu d'échec" (très complexe), le "jeu du solitaire" (assez complexe) conduisent à des graphes qui ne sont pas représentables en un temps acceptable.
Un sommet correspond à une configuration donnée (configuration instantanée du jeu), un arc correspond au passage d'un sommet à un autre (règle de jeu). Les sommets seront donc "découverts" (et créés), éventuellement oubliés, au fur et à mesure de l'avancement dans le jeu. Il n'y aura pas à priori de parcours exhaustifs possibles (dans un temps acceptable) pour de tels graphes.

4 Parcours élémentaires d'un graphe fini.

Parcourir un graphe consiste à emprunter **systématiquement** les arcs du graphe pour en visiter les sommets, tout en mémorisant certaines informations sur le graphe.
De nombreux algorithmes utilisent le résultat d'un parcours de graphe pour établir des propriétés dans ce graphe et d'autres sont des adaptations des algorithmes de parcours ou s'en inspirent fortement.

Un parcours de graphe explore TOUT ou PARTIE des chemins possibles, en gardant la possibilité de revenir en arrière (de reculer) pour explorer d'autres chemins, tout en se souvenant des chemins déjà explorés ou des sommets rencontrés. Les algorithmes de parcours utilisent donc fréquemment

- des piles FIFO ou des piles LIFO $\begin{cases} \text{FIFO : First In, First Out, aussi qualifiées de files ;} \\ \text{LIFO : Last In, First Out (piles sous-jacentes aux appels récursifs).} \end{cases}$
- le marquage des sommets (par des couleurs) (éviter un bouclage en cas de présence de cycles ...).

Pour un graphe fini, de petite taille, le parcours peut être exhaustif en un temps fini. Pour un graphe de grande taille, on ne pourra exécuter qu'un parcours partiel en un temps fini, ce qui oblige à introduire des éléments de contrôle (limitation de profondeur) et de choix (stratégie de décision) dans les algorithmes de parcours.

4.1 Parcours en largeur d'un graphe orienté ou non, depuis une origine.

Le parcours en largeur découvre d'abord tous les sommets situés à une distance k (en nombre de sommets rencontrés) du sommet origine s_0 avant de découvrir ceux situés à la distance $k+1$. En fixant une borne pour la valeur de k, on aurait un parcours partiel.

1. Principe : un sommet est introduit dès sa découverte (avant d'être traité) dans une une file FIFO.

   Comme on introduit systématiquement dans la file FIFO les sommets (non déjà traités) de la liste d'adjacence du sommet qui est en tête de la file FIFO, avant d'extraire celui-ci de la file, on traitera toujours en premier les sommets qui sont le plus près du sommet origine.

2. Informations recueillies, dans des vecteurs indicés par les numéros de sommets :
   - $c(s)$ est l'état (couleur "blanc", "gris", "noir") du sommet s : "non traité" (0), "en cours" (1), "traité" (2),
   - $d(s)$ est la distance (en nombre d'arcs) du sommet s à s_0,
   - $p(s)$ est le "père" (prédécesseur) du sommet s dans un plus court chemin (en nombre d'arcs) de s_0 jusqu'à s.

3. Algorithme :

   Initialement , $\begin{cases} \text{la file FIFO } F \text{ contient seulement le sommet } s_0, \\ \text{le vecteur } d \text{ est initialisé à "}\infty\text{" (sauf } d.(s_0)=0), \\ \text{le vecteur } p \text{ est initialisé à "nil" : pas de père connu, ou père invalide } (-1), \\ \text{le vecteur } c \text{ est initialisé à "non traité" (sauf } c.(s_0) = \text{"en cours"}). \end{cases}$

   Tant que la file F n'est pas vide , dans l'ordre,
   - soit u le sommet prêt à sortir de la file F (u est "en cours") ;
   - pour chaque sommet v "non traité" de la liste d'adjacence de u, on affecte les vecteurs c, d, p :

     $c.(v) \leftarrow$ "en cours"
     $d.(v) \leftarrow d.(u)+1$
     $p.(v) \leftarrow u$

     et on introduit v dans F ;
   - on extrait effectivement u de la tête de F et on le marque "traité".

   Remarque. Cet algorithme s'effectue en $O(|S|+|A|)$, (les opérations de gestion de la file FIFO étant en $O(1)$)

4. Exploitation des résultats : on peut déduire des vecteurs d (distance), p (père),
   - un plus court chemin (en nombre d'arcs) d'un sommet donné à un autre,
   - un arbre couvrant (ou une forêt couvrante),
   - le sous-graphe des sommets accessibles depuis un sommet donné,
   - ... .

   Le vecteur c (couleur) peut être utilisé en cours de l'algorithme pour décrire et surveiller l'état d'avancement.

4.2 Parcours en profondeur d'un graphe orienté ou non.

Le parcours en profondeur explore complètement les chemins issus d'un sommet s, dès que ce sommet est découvert.

1. Principe : lorsque tous les sommets issus d'un sommet s ont été découverts, on revient en arrière pour explorer les autres chemins qui partent du sommet à partir duquel s a été découvert.

   Cela conduit naturellement à un algorithme récursif, donc qui exploite, de façon transparente, la pile LIFO (Last In, Last Out) du système.

   Remarque. Dans le cas d'un graphe non connexe, lorsque tous les sommets accessibles à partir d'un sommet origine auront été découverts (ce qui donne une composante connexe), s'il reste des sommets non découverts, on pourra en choisir un qui servira de nouvelle origine pour continuer le parcours.

2. Informations recueillies, dans des vecteurs indicés par les numéros de sommets :
   - $c(s)$ est l'état (couleur) du sommet s : "non traité" (0), "en cours" (1), "traité" (2),
   - $p.(s)$ est le "père" (prédécesseur) du sommet s dans le chemin qui conduit à la découverte de s,
   - $dd.(s)$ est la "date" de début de traitement du sommet s,
   - $df.(s)$ est la "date" de fin de traitement du sommet s.

3. Algorithme (en $O(|S|+|A|)$) :

   Initialement , $\begin{cases} \text{le vecteur } p \text{ est initialisé à "nil" : pas de père, ou père invalide } (-1), \\ \text{le vecteur } c \text{ est initialisé à "non traité",} \\ \text{la date est 0.} \end{cases}$

   Pour chaque sommet s non traité (ou pour un seul sommet s), visiter s, soit, dans l'ordre :
   - marquer s "en cours", incrémenter la date et l'inscrire comme date de découverte de s ;
   - pour chaque sommet v "non traité" de la liste d'adjacence de s,
     affecter s comme père à v puis visiter v ;
   - marquer s "traité", incrémenter la date et l'inscrire comme date de fin de traitement de s.

4. Exploitation des résultats : on peut déduire des vecteurs p (père), dd et df (dates début et fin),
   - la réalisation d'un Tri topologique (ordonnancement linéaire de sommets),
   - la construction d'un arbre couvrant (ou d'une forêt couvrante),
   - la recherche des composantes connexes ou fortement connexes,
   - ... .

   Le vecteur c peut être utilisé en cours de l'algorithme pour décrire et surveiller l'état d'avancement.

Variante de l'algorithme : **limitation en profondeur**
Comme on descend toujours le plus profondément possible dans le graphe, il n'est pas certain que l'on puisse "remonter" dans un temps acceptable !
Une variante du parcours en profondeur consistera à donner une borne k à la profondeur d'exploration.

```
let parcours_profond g =
  (* Non limité, avec reprise sur les sommets non découverts (nouvelle composante connexe) *)
  let n = max_sommet g in     (* sommets de 1 à n *)
  let couleur = Array.make (n+1) 0
  and p = Array.make (n+1) (-1)
  and d = Array.make (n+1) (-1) and f = Array.make (n+1) (-1)
  in
  let temps = ref 0 in
  let rec visiter u =
    couleur.(u) <- 1;
    incr temps; d.(u) <- !temps;
    List.iter (explore u) (List.assoc u g);
    couleur.(u) <- 2;
    incr temps; f.(u) <- !temps
  and    explore u (v, poids) =
    if couleur.(v) = 0 then begin p.(v) <- u; visiter v end
  in
  let ls = sommets g in
  List.iter (fun u -> if couleur.(u) = 0 then visiter u) ls;
  (d,f,p)
;;
(* val parcours_profond : (int * (int * 'a) list) list -> int array * int array * int array *)
```

5 Tri topologique d'un graphe orienté acyclique (sans circuit)

On utilise le tri topologique pour établir des précédences entre événements. Par exemple,

- pour établir l'enchaînement des incidents qui sont survenus lors d'une catastrophe.
- pour établir une "check list", liste ordonnée des protocoles nécessaires à la mise en route d'un processus.

G étant un graphe représenté par listes d'adjacence, on utilise le résultat du parcours en profondeur exhaustif, vu ci-dessus, pour produire une copie G' de G, où les sommets sont ordonnés linéairement, au sens :

si G' contient un arc (u,v), le sommet u apparaît avant v dans toute liste de G'.

On pourra alors dessiner graphiquement G', sur une droite, avec ses sommets alignés dans l'ordre topologique sur cette droite et ses arcs orientés selon ce même ordre.

Remarque. Si G contient des circuits, il n'est pas possible de créer un tel ordre et le résultat est non sensé.

Algorithme : Soit f le vecteur des dates de fin de traitement (f est un des résultat du parcours en profondeur).

1. On trie les listes d'adjacence selon l'ordre << défini par : `u << v ssi f.(u) > f.(v)`
2. On trie la liste du graphe selon l'ordre <<< défini par : `(u,Lu) <<< (v,Lv) ssi f.(u) > f.(v)`

```
let tri_topologique g =
  (* tri_topologique g où g est un graphe orienté sans circuit, défini par liste d'adjacence
     renvoie un graphe dont les sommets apparaissent, dans toute liste, selon l'ordre topologique:
     u apparaît avant v s'il existe un arc (u, v)
  *)
  let (d,f,p) = parcours_profond g           (* debut, fin, pere *)
  in
  let ordre2 (s1,_) (s2,_) = f.(s1) > f.(s2)
  and ordre1 (s1,ls1) (s2,ls2) = f.(s1) > f.(s2)
  in
  tri_rapide_list ordre1 (List.fold_left (fun h (s,ls) -> (s,tri_rapide_list ordre2 ls)::h ) [] g)
;;
(* val tri_topologique : (int * (int * 'a) list) list -> (int * (int * 'a) list) list *)
```

6 Arbre couvrant d'un graphe non orienté

6.1 Arbre couvrant par suppression d'arêtes

Dans un graphe non orienté (bi-orienté symétrique), connexe (où deux sommets sont toujours reliés par une chaîne), d'ensemble de sommets $S = \{1,2,3,\ldots,n\}$, on cherche à supprimer des arêtes ou arcs, de façon à obtenir une structure arborescente (arbre couvrant).
Pour accéder facilement aux listes d'adjacence et les modifier facilement, on transforme la liste d'adjacence du graphe en un vecteur d'adjacence du graphe, indicé par les numéros de sommets.
Par exemple, la liste d'adjacence, ordonnée selon les sommets croissants de 1 à n :

```
ladj = [ 1,[2,50; 5,30; 6,40; 8,25]; 2,[1,50; 3,15; 4,5]; 3,[2,15; 4,11; 6,13]; 4, [2,5; 3,11; 5,12; 6,20];
         5,[1,30; 4,12; 6,17]; 6,[1,40; 3,13; 4,20; 5,17; 7,12]; 7,[6,12]; 8,[1,25] ]
```

est transformée en le vecteur d'adjacence, d'indices utiles les numéros de sommets de 1 à n :

```
vadj = [| 0,[]; 1,[2,50; 5,30; 6,40; 8,25]; 2,[1,50; 3,15; 4,5]; 3,[2,15; 4,11; 6,13]; 4,[2,5; 3,11; 5,12; 6,20];
               5,[1,30; 4,12; 6,17]; 6,[1,40; 3,13; 4,20; 5,17; 7,12]; 7,[6,12]; 8,[1,25] |]
```

par l'instruction : `vadj = Array.of_list ((0,[])::ladj)`.
Remarque. `vadj.(s) = s, ls` et l'information `s` est redondante ... on pourrait s'en passer, mais cela facilite le retour à une liste d'adjacence de graphe en final.

On réalise un parcours en profondeur du graphe ("visite"), en comptant le nombre de passage en chaque sommet : si un sommet est vu une deuxième fois (depuis un nouveau père potentiel), on "casse" le lien à ce père et on annule le passage.

```
let arbre_couvrant ladj =
  (* de ladj connexe depuis le sommet 1 *)
  let n = List.length ladj in
  let vadj = Array.of_list ((0,[])::ladj)  (* indices utiles 1..n ; vadj.(s) = (s,ls)*)
  in
  let compteur = Array.make (n+1) 0 in   (* compteur.(s) = nombre de visites du sommet s *)
  let rec visite p f =                   (* visite de f fils de p, excluant remontée directe de f à p *)
    compteur.(f) <- compteur.(f)+1;               (* d = dist(p,f) = dist(f,p) *)
    if  compteur.(f) > 1
    then begin (* f vu deux fois -> rupture du lien p f et annulation de la visite *)
               vadj.(p) <- p, (List.filter (function (u,x) -> u <> f ) (snd vadj.(p)));
               vadj.(f) <- f, (List.filter (function (u,x) -> u <> p ) (snd vadj.(f)));
               compteur.(f) <- 1
         end
    else (* visite des fils de f, à l'exclusion du père p de f *)
         let fils = List.map (function (u,x) -> u) (snd vadj.(f)) in
         List.iter (visite f) (List.filter (function u -> u <> p ) fils)
  in
  visite 0 1;      (* père et (fils,dist), 0 = père fictif de 1 au départ *)
  List.tl (Array.to_list vadj)  (* liste d'adjacence du graphe couvrant, sans le père fictif *)
;;
(* val arbre_couvrant : (int * (int * 'a) list) list -> (int * (int * 'a) list) list = <fun> *)
```

6.2 Arbre couvrant de poids minimal, algorithme de Prim

Voir TP France routière ...

6.3 Arbre couvrant des plus courts chemins d'origine unique, algorithme de Dijkstra

Voir TP France routière ...

7 Algorithmes de plus courts chemins (graphes orientés).

7.1 Plus courts chemins pondérés à origine unique.

7.1.1 Algorithme de Dijkstra

Recherche de tous les plus "courts" chemins **d'origine unique** s_0, dans un graphe orienté pondéré, où les arcs reliant les sommets sont pondérés par des "distances" toutes positives.

1. Principe : l'algorithme de Dijkstra utilise naturellement une **file de priorité** F, contenant les couples (x,u) tels que la distance x (provisoire) du sommet u au sommet s_0 n'a pas atteint la valeur minimale.
   Remarque. **la distance x est en tête du couple (x,u), ce qui permet d'utiliser l'ordre générique `( > )`, c'est à dire l'ordre lexicographique sur les couples, déduit des ordres usuels >.**

   La priorité maximale étant définie par une plus courte distance (provisoire) à s_0, dès qu'un élément arrive en tête de F, on a évalué sa plus courte distance à s_0 et on peut l'extraire de F.

   Pour obtenir le plus court chemin de s_0 à l'élément u prioritaire de F (en tête de F), il suffit d'avoir mémorisé les informations qui ont conduit u à la tête de F.
2. Informations recueillies, dans des vecteurs indicés par les numéros de sommets :
   - $d(s)$ est la distance (provisoire) du sommet s à s_0
   - $p(s)$ est le "père" ou prédécesseur (provisoire) du sommet s dans un plus court chemin (provisoire) du sommet s_0 au sommet s.
3. Algorithme

   Initialement, $\begin{cases} \text{le vecteur } d \text{ est initialisé à "}+\infty\text{" (sauf } d.(s_0)=0), \\ \text{le vecteur } p \text{ est initialisé à "nil" : pas de père ou père invalide } (-1), \\ \text{la file de priorité } F \text{ contient tous les sommets, accompagnés de leur distance} \\ \qquad \text{provisoire à } s_0 \text{ (critère de priorité). Remarque : le couple } (0,s_0) \text{ est en tête de } F. \end{cases}$

   Tant que la file de priorité F n'est pas vide,
   - on extrait la tête (x,u) de F (x est la longueur d'un plus court itinéraire de s_0 à u)
   - **pour chaque couple** (y,v) restant dans F, **tel que** v est directement relié à u :
     si on peut améliorer la distance provisoire y de v à s_0 en passant par u, par la valeur $z=\cdots$,
     - on actualise la valeur de $d.(v)$ en z et la valeur de $p.(v)$ en u (u devient père de v)
     - on augmente la priorité de (y,v) dans F, en prenant comme nouvelle valeur $(d.(v),v)$.

   Finalement, quand la file de priorité F est vide,
   - si u est accessible depuis s_0, $\begin{cases} d.(u) \text{ est la plus courte distance de } s_0 \text{ à } u, \\ p.(u) \text{ est le prédécesseur de } u \text{ dans un plus court chemin de } s_0 \text{ à } u, \end{cases}$
   - si u n'est pas accessible depuis s_0, $\begin{cases} d.(u)=\infty, \\ p.(u)=-1 \end{cases}$

   Remarques.
   - Comme on choisit toujours le sommet le plus proche de s_0 pour l'extraire de F, on dit que cet algorithme utilise une stratégie **gloutonne**.
   - Pour prouver cet algorithme, il suffit de montrer que, lorsque l'on extrait le sommet u de la tête de F, $d.(u)$ représente la plus courte distance de s_0 à u.
4. **Complexité** en $O((|S|+|A|)\times\log(|S|))$, où S est l'ensemble des sommets et A l'ensemble des arcs du graphe.
5. Exploitation des résultats :
   - le suivi des "pères" dans le vecteur p, depuis un sommet u jusquà s_0, permet de retrouver la liste des sommets liant s_0 à u dans un plus court chemin pondéré de s_0 à u.
   - ...

Voir autre version dans le TP France routière ...

7.1.2 Algorithme de Bellman-Ford (pour mémoire)

L'algorithme de Bellman-Ford (1958) utilise le principe de la programmation dynamique pour calculer des plus courts chemins depuis un sommet source donné dans un graphe orienté pondéré. La complexité de l'algorithme est en $O(|S|\,|A|)$.

Contrairement à l'algorithme de Dijkstra, l'algorithme de Bellman-Ford autorise la présence de certains arcs de poids négatif et permet de détecter l'existence d'un circuit absorbant, c'est-à-dire de poids total strictement négatif, accessible depuis le sommet source.

7.2 Plus courts chemins pondérés entre tous couples de sommets.

7.2.1 Algorithme de Floyd-Warshall

On considère ici un graphe G, orienté et valué, d'ensemble de sommets $S = \{1, 2, \ldots, n\}$, représenté par une matrice d'adjacence M, où, pour $i, j \in S$, M_{ij} est le poids de l'arc $(i, j) : i \longrightarrow j$, s'il existe et ∞ sinon.

Le poids d'un chemin entre deux sommets est la somme des poids sur les arcs constituant ce chemin. Les arcs du graphe peuvent avoir des poids négatifs, mais **le graphe ne doit pas posséder de circuit de poids strictement négatif.**

On supprime les boucles éventuelles en donnant, pour tout $i \in S$, la valeur 0 ou ∞ à M_{ii} (0 ou ∞, choix sans incidence ici).

Remarque. Prendre $M_{ii} = 0$ ou $M_{ii} = \infty$ pour tout $i \in S$, revient à supprimer ou ignorer les boucles (i, i), $i \in S$, du graphe, puisqu'elles n'interviennent pas dans le problème : circuits élémentaires de poids positif, qui ne peuvent qu'augmenter les distances pondérées.

Calcul des distances minimales :

Pour $0 \leqslant k \leqslant n$, on note W_{ij}^k le poids minimal d'un chemin du sommet i au sommet $j \neq i$, n'empruntant que des sommets intermédiaires dans $S_k = \{s,\ s \leqslant k\}$ (on a $S_0 = \{\}$, $S_1 = \{1\}, \ldots, S_n = \{1, 2, 3, \ldots, n\}$) et, si un tel chemin n'existe pas, on pose $W_{ij}^k = \infty$. Pour $k = 0$, il s'agit de chemins sans intermédiaires (réduits à un arc) et $W_{i,j}^0 = M_{ij}$.

On note W^k la matrice des W_{ij}^k (W_{ii}^k à préciser). En posant $W_{ii}^0 = M_{ii}$, $W^0 = M$, matrice d'adjacence du graphe G.

1. S'il existe un chemin p entre les deux sommets distincts i et j, de poids minimal, ses sommets intermédiaires, s'ils existent, sont dans un ensemble $S_k = \{s,\ s \leqslant k\}$, avec ~~$0 \leqslant k \leqslant n$~~ $1 \leqslant k \leqslant n$, et
   - soit p n'emprunte pas le sommet k,
   - soit p emprunte exactement une fois le sommet k (car les circuits sont de poids positifs ou nuls) et p est la concaténation de deux chemins, entre i et k et k et j respectivement, tous deux de poids minimal, dont les sommets intermédiaires, s'il existent, sont dans S_{k-1},

   ce qui conduit à la relation : pour $1 \leqslant k \leqslant n$, $W_{ij}^k = \min\left(W_{ij}^{k-1},\ W_{ik}^{k-1} + W_{kj}^{k-1}\right)$ (relation de récurrence sur k).
2. S'il n'existe pas de chemin entre les deux sommets distincts i et j, pour tout $k \geqslant 1$, $W_{ij}^k = \infty$ et on a, soit $W_{ik}^{k-1} = \infty$, soit $W_{kj}^{k-1} = \infty$, et la relation ci-dessus reste valable si on convient que pour tout x, éventuellement égal à ∞,
   $$\begin{cases} \min(x, \infty) = \min(\infty, x) = x \\ x + \infty = \infty + x = \infty \end{cases}$$
3. Pour $j = i$, la relation ci-dessus reste valable, $\begin{cases} \text{si on a fait le choix initial } M_{ii} = 0, \text{ avec } W_{ii}^k = 0 \text{ pour tout } k, \\ \text{si on a fait le choix initial } M_{ii} = \infty, \text{ avec } W_{ii}^k = \infty \text{ pour tout } k. \end{cases}$

En introduisant de nouvelles opérations sur les nombres, avec les conventions vues ci-dessus :

`+?` définie par `x +? y = min(x,y)` associative, d'élément neutre $+\infty$

`*?` définie par `x *? y = x + y` associative, d'élément absorbant $+\infty$, d'élément neutre 0

la relation de récurrence sur k devient : $\begin{cases} k = 0 : \text{pour } i, j \in S, & W_{ij}^0 = M_{ij} \\ k > 0 : \text{pour } i, j \in S, & W_{ij}^k = W_{ij}^{k-1} +? \left(W_{ik}^{k-1} *? W_{kj}^{k-1}\right) \end{cases}$

En final, $W_{i,j}^n$ sera la plus courte distance des chemins de i à j ($W_{i,j}^n = \infty$ s'il n'y a pas de chemin de i à j).

Plus courts chemins :

On adapte l'algorithme précédent, avec une matrice P telle que $P_{i,j}$ est le père de j dans un meilleur chemin de i à j. Lorsqu'on découvre un meilleur chemin de i à j passant par k, on remplace le père P_{ij} de j par le père P_{kj} de j.

$$\begin{cases} k = 0 : \text{pour } i, j \in S, & \begin{cases} W_{ij}^0 = M_{ij} \\ P_{ij} = \begin{cases} -1 & \text{si } i = j \text{ ou si } M_{ij} = \infty \\ i & \text{sinon (père de } j \text{ dans le chemin réduit à l'arc } i \longrightarrow j \text{)} \end{cases} \end{cases} \\ k > 0 : \text{pour } i, j \in S, & newD = W_{ik}^{k-1} *? W_{kj}^{k-1} \begin{cases} \text{si } newD < W_{ij}^{k-1}, & \begin{cases} W_{ij}^k = newD \\ P_{ij} = P_{kj} \quad (P_{kj} \text{ meilleur père de } j) \end{cases} \\ \text{sinon,} & \begin{cases} W_{ij}^k = W_{ij}^{k-1} \\ P_{ij} \quad \text{est inchangé.} \end{cases} \end{cases} \end{cases}$$

En final, $\begin{cases} P_{i,j} \text{ sera le père de } j \text{ dans un plus court chemin de } i \text{ à } j, \text{ de longueur } W_{ij}^n. \\ \text{avec } P_{i,j} = -1 \ , W_{ij}^n = \infty \quad \text{s'il n'y a pas de chemin de } i \text{ à } j. \end{cases}$ **et pour obtenir les sommets successifs dans un plus court chemin de i à j, il suffira de suivre (à reculons) les $P_{i,k}$ depuis $k = j$ jusqu'à $k = i$.**

Complexité : l'algorithme de Floyd-Warshall est de complexité $O(|S|^3)$

(Dijkstra est de complexité $O\big((|S| + |A|) \log |S|\big)$, mais il ne calcule les chemins que pour une seule origine).

7.2.2 Puissance n-ième de la matrice d'adjacence

On considère ici un graphe G, orienté et valué, d'ensemble de sommets $S = \{1, 2, \ldots, n\}$, représenté par une matrice d'adjacence M, où, pour $i, j \in S$, M_{ij} est le poids de l'arc $(i, j) : i \longrightarrow j$, s'il existe et ∞ sinon.

Le poids d'un chemin entre deux sommets est la somme des poids sur les arcs constituant ce chemin. Les arcs du graphe peuvent avoir des poids négatifs, mais **le graphe ne doit pas posséder de circuit de poids strictement négatif.**

On supprime les boucles éventuelles en donnant, pour tout i, la valeur 0 à M_{ii} (∞ **ne convient pas ici !**).

Remarque. Prendre $M_{ii} = 0$ pour tout $i \in S$, revient à supprimer ou ignorer les boucles (i, i), $i \in S$, du graphe, puisqu'elles n'interviennent pas dans le problème : circuits élémentaires de poids positif, qui ne peuvent qu'augmenter les distances pondérées.

Préalables :

Les indices utiles pour les matrices sont les numéros de sommets (de 1 à n) et on borde les matrices par les numéros de sommets : $M_{i0} = i$, $M_{0i} = i$ (simple confort de lecture).

- La matrice d'adjacence M est arrangée ainsi : pour $i, j \in S$,
  M_{ij} est la pondération de l'arc (i, j), avec $\begin{cases} M_{ij} = \infty & \text{s'il n'y a par d'arc } (i,j) \text{ (pour } i \neq j) \\ \underline{\underline{M_{ii} = 0}} & \underline{\textbf{(l'algorithme ne marche pas avec } M_{ii} = \infty \textbf{)}} \end{cases}$
- On convient que, pour tout nombre x, éventuellement égal à ∞, $\begin{cases} \min(x, \infty) = \min(\infty, x) = x \\ x + \infty = \infty + x = \infty \end{cases}$

Lemmes (dans les conditions énoncées ci-dessus) :

- Les sous-chemins d'un plus court chemin pondéré sont eux mêmes des plus courts chemins.
- Si le graphe contient n sommets, un plus court chemin pondéré comprends au plus $n - 1$ arcs.

Algorithme :

On note $D^{(m)}$ la matrice des longueurs des plus courts chemins <u>d'au plus</u> m arcs entre deux sommets.

- Initialement, pour $i, j \in S$, $D^{(1)}_{ij} = M_{ij}$.
- Pour $m > 1$, un plus court chemin <u>d'au plus</u> m arcs entre i et j (éventuellement de longueur ∞) se décompose en
  - un plus court chemin d'au plus $m - 1$ arcs de i à k (éventuellement de longueur ∞),
  - un arc de k à j (éventuellement de longueur ∞),

  d'où la relation :

$$\text{pour } i, j \in S, \quad D^{(m)}_{ij} = \min\left(D^{(m-1)}_{ij} \,,\ \min_{\substack{k=1\cdots n \\ k\neq i \ k\neq j}} \left(D^{(m-1)}_{ik} + M_{kj} \right) \right)$$
$$= \min_{k=1\cdots n} \left(D^{(m-1)}_{ik} + M_{kj} \right) \quad \text{puisque } M_{ii} = M_{jj} = 0$$

(calculs étendus aux nombres infinis, d'après les conventions introduites ci-dessus)

vérifiée valable pour $i = j$, avec $D^{(m)}_{ii} = 0$ pour tout $m \geqslant 0$.

Si on introduit de nouvelles opérations sur les nombres, <u>avec les conventions vues ci-dessus</u> :
`+?` définie par `x +? y = min(x,y)` associative, d'élément neutre ∞
`*?` définie par `x *? y = x + y` associative, d'élément absorbant ∞, d'élément neutre 0,
on obtient une structure de (quasi) anneau et l'égalité ci dessus devient :

$$\text{pour } i, j \in S, \quad D^{(m)}_{ij} = \sum_{k=1}^{n} D^{(m-1)}_{ik} \ *?\ M_{kj} \qquad \text{noté} \quad D^{(m-1)} *! \ M$$

et on reconnaît $D^{(m)}$ comme le "produit $*!$" de la matrice $D^{(m-1)}$ par la matrice $M = D^{(1)}$.

- Pour $m \geqslant n - 1$, on aura $D^{(m)} = D^{(n-1)}$, puisque un plus court chemin pondéré contient <u>au plus</u> $n - 1$ arcs.

On calcule $D^{(n-1)}$, puissance $(n-1)$-ième de la matrice $D = M$, au sens des opérations `+?` et `*?` :

- par l'algorithme d'exponentiation rapide (au sens du produit `*!`),
- en calculant la suite des $D^{(2m)} = D^{(m)} \ *! \ D^{(m)}$, à partir de $m = 1$, jusqu'à ce que $2m \geqslant n - 1$ et alors $D^{(2m)} = D^{(n-1)}$,

pour avoir dans $D^{(n-1)}(i, j)$ la longueur d'un plus court chemin de i à j.
Ensuite, on exploite le contenu de $D^{(n-1)}$ pour obtenir un plus court chemin de i à j (voir TP Hamilton).

Remarque. L'algorithme de Floyd-Warshall est plus performant !

8 Exemples de problèmes de graphes (fini).

8.1 Multi-graphes eulériens.

Dans un (multi-)graphe non orienté, une ***chaîne eulérienne*** est une chaîne empruntant une fois et une seule chaque arête et un ***cycle eulérien*** est une chaîne eulérienne dont les extrémités coïncident.

Remarque. Une chaîne eulérienne peut passer plusieurs fois par un même sommet.

Exemple 8.1. Le problème des sept ponts de Königsberg.

La ville de Königsberg est traversée par la rivière Pregel et possède 7 ponts (Figure 6).
Un piéton peut-il, en se promenant, parcourir chaque pont une seule fois ?
Si oui, peut-il le faire en revenant à son point de départ ?

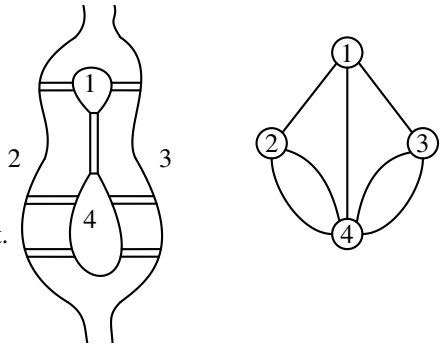

FIGURE 6 – Les 7 ponts de Koenigsberg et le multigraphe associé.

Théorème 8.1. d'Euler *(Euler-Hierholzer, 1736-1873)*

Un multi-graphe non orienté connexe admet une chaîne eulérienne si et seulement si le nombre de sommets de degré impair est
- *0 (existence d'un cycle eulérien dont les extrémités sont un même sommet de degré pair),*
- *ou 2 (existence d'une chaîne eulérienne dont les extrémités sont les deux sommets de degré impair).*

Preuve. On remarque tout d'abord qu'on peut supprimer les sommets isolés (de degré 0, pair) d'un (multi)graphe sans modifier les chaînes eulériennes et sans modifier le nombre de sommets de degré impair. D'autre part, un (multi)graphe non connexe, à sommets non isolés, n'admet pas de chaîne eulérienne.

On vérifie que la condition est nécessaire.
Dans une chaîne eulérienne qui n'est pas un cycle, les deux sommets extrémité sont les seuls à avoir un degré impair et dans un cycle eulérien, il n'y a aucun sommet de degré impair.

On montre que la condition est suffisante, en raisonnant par récurrence sur le nombre m d'arêtes.
La propriété est vérifié pour un (multi)graphe n'ayant qu'une arête.
On suppose que la propriété est vraie pour un (multi)graphe connexe ayant au plus $m \geqslant 1$ arêtes.
Soit G un (multi)graphe connexe, ayant $m+1$ arêtes et 0 ou 2 sommets de degré impair.
- si G possède 2 sommets impairs, on note a et b ces sommets
- si G possède 0 sommets impairs, on choisit a un sommet quelconque et on prends $b = a$

Soit une chaîne partant de a et n'utilisant pas deux fois la même arête. Si, en suivant cette chaîne, à un instant donné on arrive à un sommet $x \neq b$,
- si $x \neq a$, x est de degré pair et comme on n'aura utilisé qu'un nombre impair d'arêtes incidentes à x, on pourra continuer avec une arête non encore utilisée.
- si $x = a$, x est de degré impair ($a = x \neq b$) et comme on n'aura utilisé qu'un nombre pair d'arêtes incidentes à x, on pourra continuer avec une arête non encore utilisée.

Ainsi il existe une chaîne L allant de a à b, qui n'utilise pas deux fois la même arête.
Si on a utilisé toutes les arêtes de G, L est une chaîne eulérienne.
Sinon, soit G' le (multi)graphe constitué par toutes les arêtes non utilisées et les sommets incidents à ces arêtes (G' n'est pas forcément connexe !), de composantes connexes $G'_1, \cdots, G'_p$. Chacun des G_i comporte au plus m arêtes, n'a pas de sommet de degré impair (ce serait contradictoire avec l'hypothèse et le choix de a et de b) et, d'après l'hypothèse de récurrence, admet un cycle eulérien μ_i.
Comme G est connexe, le chemin L rencontre chacun des G_i en un ou plusieurs sommets. La chaîne L', déduite de L par le remplacement d'un seul des sommets de rencontre avec chaque G_i par le cycle μ_i d'extrémités ce sommet, est une chaîne eulérienne dans G.
D'après le principe de récurrence, on en déduit le théorème.

Remarque. Ce théorème est fondamental
- preuve (simple) de l'existence d'une solution (c'est assez rare en théorie des graphes !)
- de plus, la démonstration est constructive : on en déduit un algorithme de construction d'une chaîne eulérienne dans un graphe eulérien à m arêtes (algorithme en $O(m)$).

Définition 8.1.

Un (multi-)***graphe eulérien*** est un multi-graphe non orienté connexe, ayant 0 ou 2 sommets de degré impair.

8.2 Le problème du postier chinois

Objectif : Dans un (multi)graphe non orienté connexe, dont les arêtes sont valuées positivement (chaque arête u est munie d'une longueur $\ell(u) \geqslant 0$), déterminer un plus court chemin qui passe au moins une fois par chaque arête du graphe et revient à son point de départ.
Le problème se pose dans un graphe qui n'admet pas de cycle eulérien, puisqu'un cycle eulérien (empruntant une et une seule fois chaque arête), est une solution optimale du problème.

Lemme 8.2.

Dans un (multi)graphe, le nombre de sommets de degré impair est pair.

En effet, la somme des degrés des sommets est le double du cardinal de l'ensemble des arcs et la somme des degrés des sommets pairs est paire.

L'idée principale pour résoudre le problème du postier chinois est de dupliquer un certain nombre d'arêtes pour obtenir un multigraphe sans sommets de degré impairs, tout en minimisant la somme des longueurs des arrêtes supplémentaires.

Dans l'exemple ci-contre, on obtient un cycle passant
deux fois par l'arête (B,D)
deux fois par l'arête (A,C) :
$A \longrightarrow B \longrightarrow C \longrightarrow D \longrightarrow B \longrightarrow D \longrightarrow A \longrightarrow C \longrightarrow A$

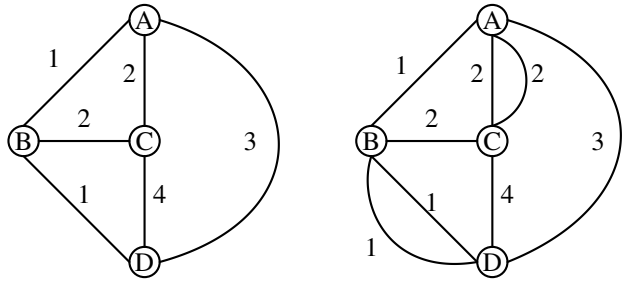

FIGURE 7 – Résolution d'un problème du postier chinois, par duplication d'arêtes.

8.3 Graphes Hamiltoniens

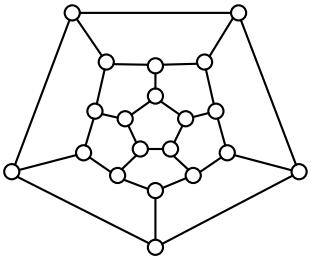

FIGURE 8 – Voyage autour du monde de Hamilton.

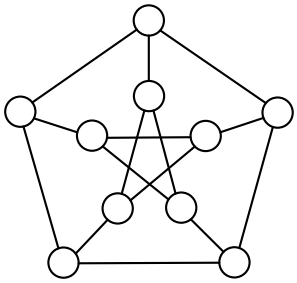

FIGURE 9 – Petersen (non hamiltonien).

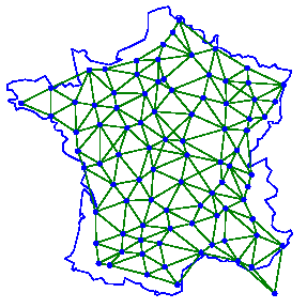

FIGURE 10 – France routière (hamiltonien).

Dans un graphe orienté (/non orienté), fortement connexe (/connexe),

- un ***chemin hamiltonien*** (/une ***chaîne hamiltonienne***) est un chemin (/une chaîne) passant une fois et une seule par chacun de ses sommets ;
- un ***circuit hamiltonien*** (/un ***cycle hamiltonien***) est un circuit (/cycle) passant une fois et une seule par chacun de ses sommets.

Un graphe est ***hamiltonien*** s'il contient un circuit (/cycle) hamiltonien passant par tous les sommets.

Exemple 8.2. Le voyage autour du monde de Hamilton.

Visiter une et une seule fois 20 villes, sommets d'un dodécaèdre régulier (Figure 8, graphe hamiltonien) et revenir au point de départ.
Variante : un joueur marque cinq sommets consécutifs quelconques du dodécaèdre et l'autre joueur essaie de compléter ce chemin pour former un cycle qui passe une et une fois par chaque sommet.

Exemple 8.3. Le problème du voyageur de commerce (ou du ramassage scolaire)

Visiter une et une seule fois n villes (Figure 10), en revenant à son point de départ et en minimisant la distance parcourue.

Remarques.

- On ne connaît pas de condition nécessaire et suffisante d'existence d'un cycle ou d'un circuit hamiltonien passant par tous les sommets (il existe un certain nombre de conditions suffisantes).
- Le problème de la recherche d'un circuit ou cycle hamiltonien passant par tous les sommets dans un graphe orienté on non est un problème **NP complet** (on ne connaît pas d'algorithme polynomial), mais une solution peut être validée en un temps polynomial (en $O(|S|^2)$) par un algorithme.

8.4 Le problème du sac à dos (tout ou rien).

Un randonneur (ou un voleur) doit choisir, parmi un ensemble de n objets possibles à emporter dans son sac, un sous-ensemble dont le poids n'excède pas un poids donné et qui soit la meilleure sélection possible en terme de valeur.
On pourrait envisager un problème avec plus d'une contrainte (poids, volume, longueur, etc..) comme dans le cas du chargement d'un bateau, d'un avion, d'un camion avec la cargaison la plus profitable.

Il s'agit ici de la variante "tout ou rien" du problème du "sac à dos", la variante "fractionnaire", où le randonneur (ou voleur) peut emporter des fractions d'objets, étant élémentaire à résoudre.

Le problème unidimensionnel (une seule contrainte) du "sac à dos tout ou rien" peut se résoudre par

- un algorithme de **programmation dynamique**, si les poids sont des entiers, pseudo-polynomial en $O(n \times (b+1))$ où n est le nombre d'objets et b est le poids maximal (entier) ;
-

8.4.1 Algorithme de programation dynamique. (Gilmore et Gomory, 1966)

Cet algorithme **suppose que les poids des n objets sont des entiers (positifs)** [1] et évalue toutes les possibilités entières de poids pour le sac, de 0 à la valeur maximale b autorisée. On aura donc une complexité temporelle en $O(n \times (b+1))$, et une complexite spatiale en $O(n \times (b+1))$.
Si on ne veut que la valeur du chargement optimal, la complexité spatiale peut se réduire avec un calcul par recouvrement, mais pour avoir aussi la composition du chargement optimal, il faut rester avec une complexité spatiale en $O(n \times (b+1))$.

Les n objets sont numérotés de 1 à n, on note p_i le poids (entier positif) de l'objet i et v_i sa valeur.
On note b (borne) le poids maximal (entier) à ne pas dépasser.

Le problème est le suivant : $\begin{cases} \text{maximiser } \sum_{j=1}^{n} v_j x_j \\ \text{avec } \sum_{j=1}^{n} p_j x_j = b \end{cases}$ où $x_j = 1$ ou 0 suivant que l'objet i est pris ou non.

On "plonge" le problème dans la famille des $n \times (b+1)$ problèmes : $\mathscr{P}_k(y)$: $\begin{cases} \text{maximiser } \sum_{j=1}^{k} v_j x_j \\ \text{avec } \sum_{j=1}^{k} p_j x_j = y \end{cases} \quad \begin{vmatrix} k = 1 \dots n \\ y = 0 \cdots b \end{vmatrix}$

En notant $f_k(y)$ la valeur d'une solution optimale du problème $\mathscr{P}_k(y)$, on a, pour $k > 1$:

$$f_k(y) = max\{ f_{k-1}(y) , v_k + f_{k-1}(y - p_k) \}$$

D'où l'algorithme, ici avec calcul par recouvrement et avec mémorisation de la composition du sac pour chaque poids :

```
let sac_a_dos_dyn v p b =       (* valeurs, poids, borne; sommets de 1 à n *)
  let n = Array.length v -1
  and f = Array.make (b+1) 0    (* valeur du sac pour le poids y (y = 0 .. b) *)
  and l = Array.make (b+1) []   (* composition du sac pour le poids y *)
  in
  for k = 1 to n
  do for y = b downto p.(k)                (* downto primordial ! *)
     do let u = v.(k) + f.(y - p.(k)) in
        if u > f.(y) then
        begin f.(y) <- u;                  (* valeur du sac = max ...  *)
              l.(y) <- k::l.(y-p.(k));     (* composition du sac       *)
         end
    done
  done;
  let v2 = List.fold_left (fun u k -> u + v.(k)) 0 l.(b) in  (* valeur du sac (re calculé) *)
  let p2 = List.fold_left (fun u k -> u + p.(k)) 0 l.(b) in  (* poids  du sac *)

  v2, p2, l.(b)    (* attendu: v2 = f.(b). *)
;;
(* val sac_a_dos_dyn : int array -> int array -> int -> int * int * int list = <fun> *)
```

Exemple :

```
let poids  = [| 0; 7; 6; 10;  8;  9; 5 |] and valeurs = [| 0; 10; 8; 13; 10; 10; 5 |];;
sac_a_dos_dyn choix poids 39;;   (* (48, 39, [6; 5; 4; 3; 1]) *)
sac_a_dos_dyn choix poids 38;;   (* (46, 36, [6; 4; 3; 2; 1]) *)
sac_a_dos_dyn choix poids 12;;   (* (15, 12, [6; 1]) *)
```

1. Si les poids des objets sont décimaux, il faut multiplier les poids des objets et la capacité du sac afin de les rendre entiers.

Algorithme sans recouvrement, avec calcul de la composition du sac en final :

```
let sac_a_dos_dyn_wiki v p b =
  let n = Array.length v -1 in
  let t = Array.make_matrix (n+1) (b+1) 0  (* historique complet *)
  in
  for k = 1 to n
  do for y = 0 to b
     do if y >= p.(k)
        then t.(k).(y) <- max t.(k-1).(y)  (v.(k) + t.(k-1).(y-p.(k)))
        else t.(k).(y) <- t.(k-1).(y)
     done
  done;
  (*-------------- retrouver la composition du sac ----*)
  let l = ref []
  and k = ref n
  and y = ref b in
  while !y > 0 && t.(!k).(!y) = t.(!k).(!y-1) do y := !y-1 done;
  while (!y > 0) && (!k > 0)
  do begin
        while (!k > 0) && (t.(!k).(!y) = t.(!k-1).(!y)) do k := !k-1 done;
        y := !y - p.(!k);
        if !y >= 0 then l := !k::!l;
        k := !k-1
     end;
  done;
  (*-------------*)
  let v2 = List.fold_left (fun u k -> u + v.(k)) 0 !l in  (* valeur du sac (re-calcul) *)
  let p2 = List.fold_left (fun u k -> u + p.(k)) 0 !l in  (* poids du sac *)

  v2, p2, !l   (* t.(n).(b) *)      (* attendu: v2 = t.(n).(b). *)
;;
(* val sac_a_dos_dyn_wiki : int array -> int array -> int -> int * int * int list = <fun> *)
```

9 Metaheuristiques, algorithmes d'approximation

9.1 Recherche d'un cycle hamiltonien de longueur (valuation) minimale

A partir d'un ou plusieurs cycles hamiltonien initiaux, de longueurs (en poids) non minimales, on peut chercher à utiliser ces cycles pour en déduire d'autres cycles hamiltoniens qui soient de longueurs inférieures aux précédents.

Remarque. Il n'est pas toujours évident de trouver un ou plusieurs cycles hamiltoniens initiaux et l'amélioration, si elle se produit, pourrait ne pas aboutir à une solution optimale (existence de minima locaux).

9.1.1 Algorithme des fourmis

Des fourmis circulent sur le graphe, de sommet en sommet, en suivant les arcs du graphe.

- Les fourmis qui ont réussi à effectuer un cycle hamiltonien, y déposent suffisamment de phéromones pour inciter leurs congénères à parcourir à leur tour tout ou partie des arcs de ce cycle.
- Des fourmis élitistes renforcent les dépôts de phéromones sur les meilleurs cycles.
- Pour essayer d'éviter de s'enfermer dans des minima locaux,
  - les traces de phéromones s'évaporent au fils du temps,
  - le choix d'un arc par une fourmi ne dépends pas uniquement de la quantité de phéromones qui y est déposée : des fourmis ("aventureuses" ou exploratrices) peuvent être tentées d'emprunter des arcs nouveaux ou non fortement marqués.

Cet algorithme dépends de divers paramètres : nombre de fourmis, amplitude des dépôts, évaporation, ampleur de la contribution des fourmis élitistes, probabilité de choisir un arc plutôt qu'un autre, probabilité d'emprunter un arc faiblement marqué, etc ..., ces paramètres devant être ajustés selon les graphes.

- en fin de tour réussi, dépôts de phéromones et gestion de l'évolution de ces dépôts,
- pendant un parcours, choix aléatoire d'un arc, dépendant des traces de phéromones,
- introduction de perturbations : arcs exclus de l'augmentation des dépôts de phéromones, choix aléatoire d'arcs du meilleur cycle hamiltonien qui seront coupés temporairement ou affaiblis en dépôts de phéromone, etc ... (perturbations qui pourraient éviter de s'enfermer dans des minima locaux).
- éventuellement, mélange de deux approximations, obtenues avec des cycles initiaux différents, pour continuer à partir d'un nouveau cycle initial (ce qui rejoint un peu l'algorithme génétique ci-dessous).

Ce genre d'algorithme peut donner des résultats intéressants (découverte, amélioration notable, pas forcément optimale), même pour des graphes de grande taille,

Voir le TP Hamilton.

9.1.2 Algorithme génétique (données modifiées génétiquement)

Algorithme qui s'inspire de la théorie de l'évolution : ici, un individu est un cycle hamiltonien, l'ADN de l'individu est la liste des sommets dans l'ordre du parcours, un gène étant le sommet où l'on passe à une certaine position dans le parcours. L'adaptation d'un individu à l'environnement est représentée par l'évaluation de la longueur (en poids) du cycle et on fait des "croisements" et de la "sélection" à l'intérieur de la population :

- on combine deux solutions pour en produire une nouvelle, cette combinaison pouvant
  - subir des mutations (ce qui peut éviter de s'enfermer dans des extrema locaux),
  - être l'objet de réparations de son ADN pour redevenir valide ;
- on élimine les solutions les moins adaptées (attention, il faut quand même laisser des canards boiteux !).

En répétant ce processus, l'adaptation moyenne de la population devrait augmenter et on peut espérer se rapprocher d'une solution optimale du problème. Là encore, il y a de nombreux paramètres à introduire et à ajuster selon les graphes ... .

Ce genre d'algorithme peut aussi donner des résultats intéressants (découverte, amélioration notable, pas forcément optimale), même pour des graphes de grande taille, et, apparemment, à condition de disposer de plusieurs cycles hamiltoniens initiaux, il serait équivalent en termes de performances à l'algorithme des Fourmis.

9.1.3 Autres : Recuit simulé, algorithmes par partitions, ...

WTSP (World Traveling Salesmam Problem) with 1904711-city instance. The tour of length 7515772212 was found on May 24, 2013, by Keld Helsgaun using a variant of his LKH heuristic algorithm. The current best lower bound on the length of a tour for the World TSP is 7512218268, established by the Concorde TSP code (June 5, 2007),

10 Annexes

10.1 Interface du module Prior (files de priorités)

```
(* Priors -- Prior.mli (interface to module Prior) *)
(* OCaml *)
(*-------------------------------------------------------------------------
This module implements prior (file of priority), with in-place modification
The implementation uses balanced binary trees, as tas, and therefore searching and
insertion take time logarithmic in the size of the prior.
Warning! This module is delicate to use and may be not always safe especially for functions
prior_modify, prior_iter_modify, prior_iter_augment that must be tested again and again.
For advanced users only.
-------------------------------------------------------------------------
Files: Prior.cmi ; Prior.cmo ; testPrior.ml
Use:
#load "Prior.cmo";;
open Prior;;
-------------------------------------------------------------------------
Author: antoine MOTEAU
cible: Ocaml
date: 13/12/2017
version:
-------------------------------------------------------------------------*)

type 'a prior;;  (* The type of priors containing elements of type 'a. *)

exception Empty;;  (* Raised when take is applied to an empty prior. *)
exception Full;;   (* Raised when add is applied to a full prior. *)

val prior_new : ('a -> 'a -> bool) -> int -> 'a -> 'a prior;;
(* prior_new o n a return a new empty prior of maximum length n, containing elements of
same type as a, using o as total ordering function over the type of elements.
The exact value of the argument a is not signifiant. *)

val prior_length : 'a prior -> int;;
(* Return the length (number of elements) of the given prior. *)

val prior_contains : 'a prior -> 'a -> bool;;
(* prior_contains p a is true if and only if a is structurally equal (see module eq)
to  an element of p. *)

val prior_take : 'a prior -> 'a;;
(* prior_take q removes and returns the highest element in prior p, according the
ordering function of p, or raises Empty if the prior is empty. *)

val prior_add : 'a prior -> 'a -> unit;;
(* prior_add p a insert the element a in the prior p, according the ordering
function of p, or raises Full if the prior is full. *)

val prior_modify : 'a prior -> 'a -> 'a -> unit;;
(* prior_modify p a b replace a by b in the prior p, rearranging p according to the
ordering function of p, or raise Not_found if the element a is not in p. *)

val prior_iter_modify : ('a -> 'a) -> 'a prior -> unit;;
(* prior_iter_modify f p replace all element a in p with element f a, rearranging p
according to the ordering function of p. *)

val prior_iter_augment : ('a -> 'a) -> 'a prior -> unit;;
(* prior_iter_augment f p replace all element a in p with element f a, only if a is
less (or equal) f a, according to the ordering function of p, rearranging p according
to the ordering function of p. Raise Failure "Diminution" if a is not less than f a.
Comment: prior_iter_augment is faster then prior_iter_modify. *)

(*== End of Prior.mli ==*)
```

Lexique :

***boucle* :** une *boucle* est un arc ou une arête allant d'un sommet à lui même.

***degré sortant* :** le *degré sortant* d'un sommet u est le nombre d'arcs qui partent de u.

***degré rentrant* :** le *degré rentrant* d'un sommet u est le nombre d'arcs qui arrivent en u.

***degré* :** le *degré* d'un sommet u est le nombre d'arcs ou d'arêtes partant de u ou aboutissant en u (un arc ou une arête (u,u) compte pour deux dans le calcul du degré de u).

***poids* :** le *poids* (ou la *valuation*) d'un chemin, d'une chaîne, est la somme des pondérations de ses arcs, de ses arêtes.

***longueur* :** la *longueur* d'un chemin, d'une chaîne est le nombre de ses arcs, de ses arêtes.

***accessible* :** un sommet v est *accessible* à partir de u s'il existe un chemin de u vers v, ou une chaîne entre u et v.

***acyclique* :** un graphe (orienté) est dit *acyclique* s'il ne contient pas de cycle.

***connexe* :** un graphe non orienté est dit *connexe* si tout couple de sommets est reliés par une chaîne.
Un graphe orienté sera dit *connexe* si le graphe obtenu en supprimant l'orientation est connexe.

***composante connexe* :** une *composante connexe* d'un graphe non orienté est un sous-graphe connexe maximal.

***fortement connexe* :** un graphe orienté est *fortement connexe* si chaque sommet est accessible à partir d'un sommet quelconque.

***composante fortement connexe* :** une *composante fortement connexe* d'un graphe orienté est un sous-graphe fortement connexe maximal.

***arbre* :** un *arbre* est un graphe non orienté, non pondéré, sans circuits.

arbre enraciné* :** un *arbre enraciné* est un arbre que l'on a orienté de telle façon que les sommets soient accessibles depuis un unique sommet initial appelé ***racine.

***forêt* :** une *forêt* est la réunion d'arbres distincts.

Ouvrages de référence

[CM95] G.Cousineau, M.Mauny. Approche fonctionnelle de la programmation, Edisciences international Paris 1995.
Pas mal d'informations autour de Caml, avec une programmation parfois compliquée .... Quelques éléments sur les graphes, pas mal de choses sur les arbres et rien sur les automates. Les fonctions Caml un peu élaborées sont souvent complexes, inutilement. On cherche à faire "joli" au détriment de la simplicité, alors que l'on pourrait souvent faire plus simple et plus performant.

[lA] Luc Albert, Ouvrage collectif. Cours et exercices d'informatique. Vuibert 1998
Surtout de bonnes choses (complètes et un peu complexes) sur les automates Rien sur les arbres ni sur les graphes. Logique et circuits combinatoires. Pas de Caml (ou si peu).

[CLR94] T.Cormen, C.Leiserson, R.Rivest. Introduction à l'algorithmique Dunod, 1994.
Introduction to Algorithms, MIT Press 1990.
Très complet sur les problèmes algorithmiques et en particulier sur les graphes et arbres. Peu de choses sur les automates. Un chapitre intéressant sur les algorithmes parallèles et sur les circuits combinatoires (mais assez complexe).

[GM79] M.Gondran, M.Minoux. Graphes et algorithmes, Eyrolles Paris 1979.
Comme le titre le laisse espérer, c'est assez complet sur les graphes, dans un esprit d'application à des problèmes concrets difficiles. (L'édition est déjà un peu ancienne).

$< \mathscr{FIN} >$ Graphes (éléments).