Chapitre 8
Graphes

1. Définitions et exemples

Beaucoup de problèmes de la vie courante, tels la gestion de réseaux de communication ou l'ordonnancement de tâches, correspondent à des structures relationnelles que l'on peut modéliser par des graphes. Informellement un graphe est un ensemble d'objets, appelés *sommets*, et de *relations* entre ces sommets.

Exemple 1 : Dans une carte des liaisons aériennes, les villes sont des sommets du graphe et l'existence d'une liaison aérienne entre deux villes est la relation du graphe (figure 1).

Dans cet exemple, la relation est symétrique : on peut raisonnablement supposer que la compagnie aérienne assure des vols aller-retour. Dans ce cas on dit que le graphe est *non orienté*. Si deux sommets $s1$ et $s2$ sont en relation, on dit qu'il existe une *arête* entre $s1$ et $s2$. On peut se poser des questions du type suivant : existe-t-il un moyen d'aller d'une ville A à une ville B ? Quel est le trajet qui nécessite le moins d'escales ? etc.

Exemple 2 : Dans le graphe du flôt de contrôle d'un programme, les boîtes (instructions ou tests) sont les sommets, et les flèches indiquent les enchaînements possibles entre celles-ci.

Dans cet exemple, la relation n'est pas symétrique : $I := I + 1$ s'exécute immédiatement avant $S := S + A[I]$, mais pas l'inverse. Dans ce cas, on dit qu'on a un graphe *orienté*. Si deux sommets $s1$ et $s2$ sont en relation, on dit qu'on a un *arc* de $s1$ vers $s2$.

Exemple 3 : Dans une organisation du travail où certaines tâches doivent être exécutées avant d'autres, on peut schématiser l'ordonnancement des tâches par un graphe où les sommets sont les tâches et où il existe un arc entre deux tâches t_i

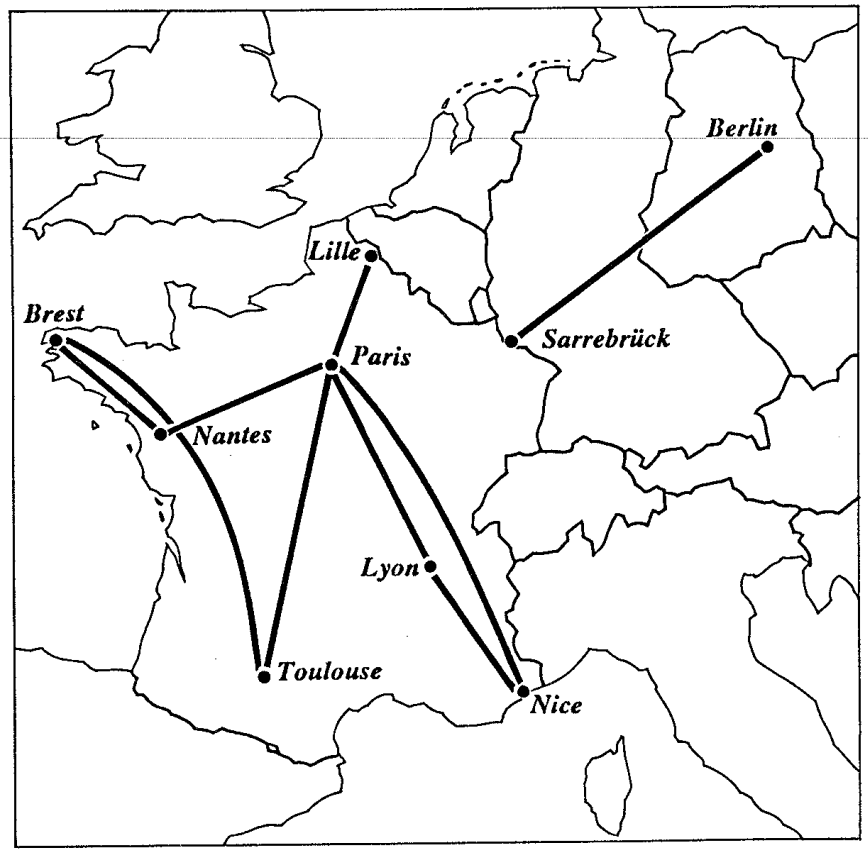


Figure 1. Graphe de liaisons aériennes.

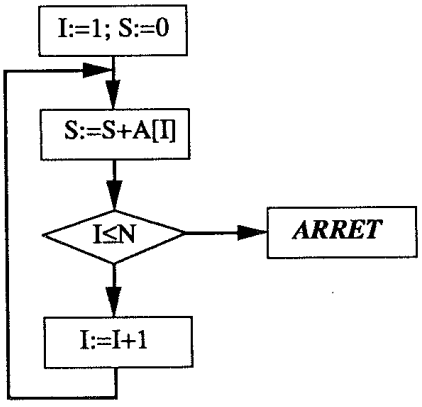


Figure 2. Graphe du flôt de contrôle d'un programme.

et t_j seulement si t_i doit être terminée juste avant d'exécuter t_j. Le graphe de la figure 3 décrit la préparation d'un curry d'agneau.

Un des traitements intéressants sur un tel graphe est un *tri topologique* qui consiste à trouver un ordre des tâches tel que toute tâche t_i soit exécutée avant toute tâche t_j s'il existe un arc de t_i à t_j.

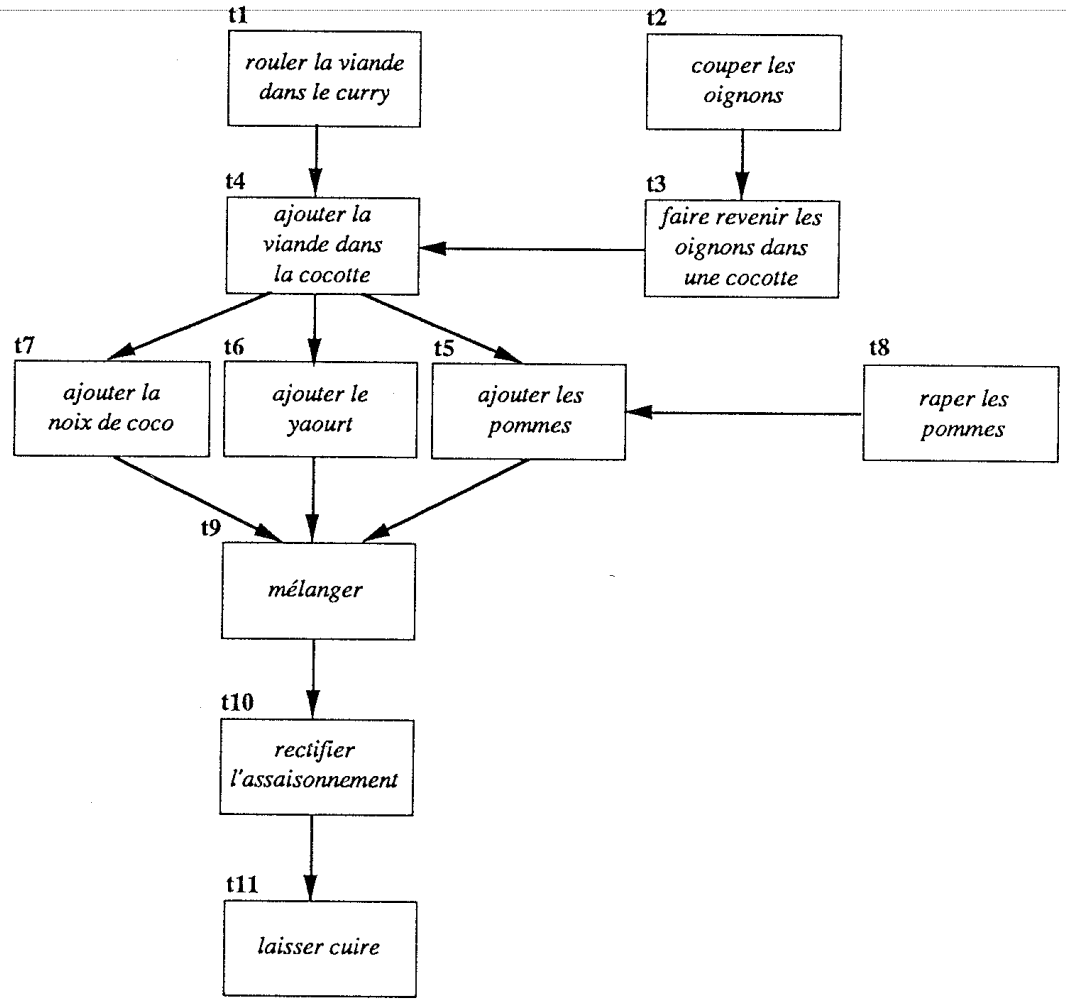


Figure 3. Exemple de graphe d'ordonnancement de tâches.

Définitions :

• Un ***graphe orienté*** G est un couple $< S, A >$, où S est un ensemble fini de sommets et où A est un ensemble fini de *paires ordonnées* de sommets, appelées ***arcs***.

• Un ***graphe non orienté*** G est un couple $< S, A >$, où S est un ensemble fini de *sommets* et où A est un ensemble fini de *paires* de sommets, appelées ***arêtes***.

• Dans de nombreux problèmes, il est naturel d'associer une valeur (on dit aussi un coût ou un poids) aux arcs ou aux arêtes du graphe. Un ***graphe valué***, orienté (resp. non orienté) est un triplet $< S, A, C >$ où S est un ensemble fini de sommets, A un ensemble fini d'arcs (resp. arêtes) et C une fonction de A dans $\mathbb{R}$ appelée fonction de coût.

On peut alors traiter des problèmes tels que : trouver le moyen le plus économique d'aller de Brest à Lyon, connaissant pour chaque ligne aérienne, le prix du billet, ce qui revient à *rechercher un plus court chemin entre deux sommets du graphe.*

Remarque 1 : La distinction entre graphes orientés et graphes non orientés n'est pas aussi catégorique qu'on pourrait le croire : quand on a un graphe orienté, il est parfois commode de ne pas tenir compte de l'orientation si le problème posé est de nature non orientée. Dans ce cas, on se contente de considérer, provisoirement, que la relation est symétrique.

Remarque 2 : Certains graphes admettent des boucles, c'est-à-dire, des arcs (ou des arêtes) qui connectent un sommet avec lui-même. On peut aussi trouver dans un graphe, deux sommets reliés par plusieurs arcs (ou arêtes) correspondant à des relations différentes (cas de *multigraphes*). On parle dans ce cas d'*arcs* (ou d'*arêtes*) *multiples*. Dans ce qui suit, on suppose que les graphes n'ont ni boucles ni arcs (ni arêtes) multiples (*graphes simples*).

2. Terminologie

Nous définissons ici le vocabulaire et les notations usuels sur les graphes.

• On note $x \rightarrow y$ l'arc (x, y) ; x est l'***extrémité initiale*** de l'arc, y est son ***extrémité terminale***. On dit que y est un ***successeur*** de x et que x est un ***prédécesseur*** de y.

De même, on note x—y l'arête $\{x, y\}$; x et y sont les deux ***extrémités*** de l'arête. Par abus de langage, on dit parfois que y est ***successeur*** de x ou que x est successeur de y.

• Soit $G = < S, A >$ un graphe. Le ***sous-graphe de G engendré par $S' \subseteq S$*** est le graphe G' dont les sommets sont les éléments de S' et dont les arcs (resp. arêtes) sont les arcs (resp. arêtes) de G ayant leurs deux extrémités dans S' (cf. figure 4b). Autrement dit, on ignore les sommets de $S - S'$ ainsi que les arcs ayant au moins une extrémité dans $S - S'$.

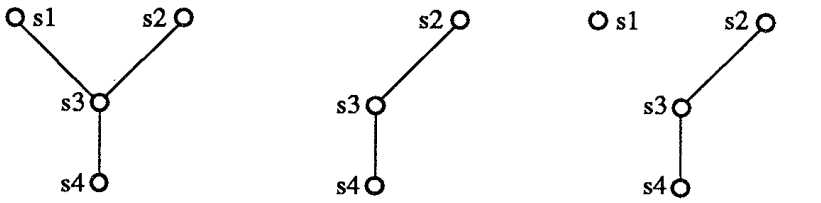
• Soit $G = < S, A >$ un graphe. Le ***graphe partiel de G engendré par $A' \subseteq A$*** est le graphe $< S, A' >$ dont les sommets sont les éléments de S et dont les arcs (resp. arêtes) sont ceux de A'. Autrement dit, on élimine de G les arcs (arêtes) de $A - A'$ (cf. figure 4c).

• Deux ***arcs*** (resp. *arêtes*) d'un graphe orienté (resp. non orienté) sont dits *adjacents* s'ils ont au moins une extrémité commune.

Deux ***sommets*** d'un graphe non orienté sont dits ***adjacents*** s'il existe une arête les joignant.

Dans un graphe orienté, le *sommet* y est dit *adjacent* au sommet x s'il existe un arc $x \rightarrow y$.

• Un graphe orienté (resp. non orienté) est dit ***complet*** si pour tout couple de sommets (x, y), il existe un arc $x \rightarrow y$ (resp. une arête x—y).



a) G (non orienté) **b)** sous-graphe S'={s2, s3, s4} **c)** graphe partiel A'={(s2, s3), (s3, s4)}

Figure 4. Exemple de sous-graphe et de graphe partiel.

• Dans un graphe orienté, si un sommet x est l'extrémité initiale d'un arc $u = x \rightarrow y$, on dit que l'arc u est ***incident à*** x ***vers l'extérieur***. Le nombre d'arcs ayant leur extrémité initiale en x, se note $d^{\circ +}(x)$ et s'appelle le ***demi-degré extérieur*** de x.

On définit de même les notions d'arc ***incident vers l'intérieur*** et de ***demi-degré intérieur*** qui est noté $d^{\circ -}(x)$.

• Dans un graphe orienté (resp. non orienté), on appelle ***degré*** d'un sommet x, et on note $d^{\circ}(x)$, le nombre d'arcs (resp. d'arêtes) dont x est une extrémité. Dans le cas d'un graphe orienté, on a : $d^{\circ}(x) = d^{\circ +}(x) + d^{\circ -}(x)$, pour tout sommet x. Dans l'exemple de la figure 5, le calcul des degrés du sommet $x3$ donne : $d^{\circ +}(x3) = 2; d^{\circ -}(x3) = 3; d^{\circ}(x3) = 5$.

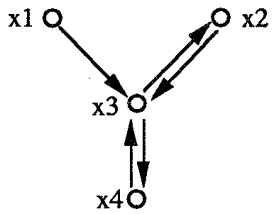


Figure 5. $d^{\circ +}(x3) = 2; d^{\circ -}(x3) = 3; d^{\circ}(x3) = 5$.

• Dans un graphe orienté G (resp. non orienté), on appelle ***chemin*** (resp. ***chaîne***) ***de longueur*** λ, une suite de $(\lambda+1)$ sommets $(s_0, s_1, ..., s_\lambda)$ tels que : pour tout i tel que $0 \leq i \leq \lambda - 1$, $s_i \rightarrow s_{i+1}$ est un arc (resp. une arête) de G. Par convention, on dit qu'il y a un chemin de longueur 0 de tout sommet vers lui-même.

On peut aussi définir de façon récursive un chemin de longueur λ ($\lambda > 0$) allant du sommet x vers le sommet y comme :

- si $\lambda = 1$, un arc de x vers y
- sinon la suite composée d'un arc de x vers un certain sommet z et d'un chemin de z vers y, de longueur $\lambda - 1$.

La définition récursive d'une chaîne de longueur λ est analogue.

• Un chemin (resp. une chaîne) est dit ***élémentaire*** s'il ne contient pas plusieurs fois le même sommet.

• Dans un graphe orienté (resp. non orienté), un chemin (resp. une chaîne) $(s_0, s_1, ..., s_\lambda)$ dont les λ arcs (resp. arêtes) sont tous distincts deux à deux et tel que les deux sommets aux extrémités du chemin (resp. de la chaîne) coïncident, est un ***circuit*** (resp. un ***cycle***).

• Un graphe orienté est dit ***fortement connexe*** si pour toute paire ordonnée de sommets distincts (u, v), il existe un chemin de u vers v et un chemin de v vers u.

Un graphe non orienté est dit ***connexe*** si pour toute paire de sommets distincts $\{u, v\}$, il existe une chaîne reliant u et v.

• On appelle ***composante fortement connexe*** d'un graphe orienté un sous-graphe fortement connexe maximal, c'est-à-dire un sous-graphe fortement connexe qui n'est pas strictement contenu dans un autre sous-graphe fortement connexe.

De même, dans un graphe non orienté, on appelle ***composante connexe*** un sous-graphe connexe maximal. Le graphe de la figure 1 a deux composantes connexes.

Arbres et arborescences en théorie des graphes

On définit en théorie des graphes des structures appelées *arbres* et *arborescences*. Ces structures sont proches de la structure de données qui a été appelée précédemment arbre planaire général. En théorie des graphes, on appelle ***arbre*** un *graphe non orienté, connexe, sans cycle.*

Un certain nombre de propriétés caractéristiques des arbres sont énoncées ci-dessous.

Proposition : Soit $G = < S, A >$ un graphe non orienté, ayant n sommets. Les propriétés suivantes sont équivalentes :

(1) G est un graphe connexe sans cycle,

(2) G est connexe et si on supprime une arête, il n'est plus connexe,
(3) G est connexe avec $(n-1)$ arêtes,
(4) G est sans cycle et en ajoutant une arête, on crée un cycle,
(5) G est sans cycle avec $(n-1)$ arêtes,
(6) tout couple de sommets de S est relié par une chaîne et une seule.

La preuve de cette proposition est laissée en exercice.

Le type de données arbre planaire général, défini au chapitre 7 correspond plutôt à la notion d'*arborescence* définie ci-dessous, car la relation père-fils est orientée.

Etant donné $G =< S, A >$ un graphe orienté, on appelle ***racine*** de G un sommet r de S tel que tout autre sommet de S puisse être atteint par un chemin d'origine r. Notons qu'une racine n'existe pas toujours : le graphe de la figure 6 n'a pas de racine.

Figure 6. Exemple de graphe qui n'est pas une arborescence.

On appelle ***arborescence*** un graphe orienté G admettant une racine, et tel que le graphe non orienté G', obtenu à partir de G en oubliant l'orientation des arcs, soit un arbre. Les propriétés caractéristiques des arborescences sont étudiées en exercice.

Rappelons que dans un arbre planaire général, les successeurs d'un nœud forment une suite ordonnée. Dans une arborescence ils forment, par contre, un ensemble et l'ordre dans lequel on accède aux éléments de cet ensemble n'est que pure convention. Par exemple les deux graphes de la figure 7 sont une même arborescence, mais si on les considère comme des arbres planaires généraux, ceux-ci sont différents.

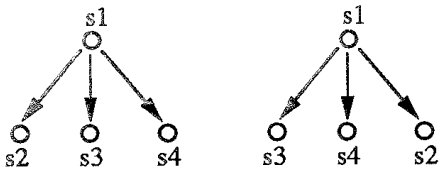


Figure 7. Deux arborescences identiques.

Inversement, si on considère un arbre planaire général comme un graphe, en faisant abstraction de l'ordre des fils de chaque nœud, on obtient une arborescence.

Dans le chapitre sur les structures arborescentes, on a donné la définition d'une forêt d'arbres planaires généraux : c'est une liste de tels arbres. On dit la même chose à propos des arbres et arborescences de la théorie des graphes, mais on ignore

l'ordre des arbres (ou arborescences) : un graphe qui est un ensemble fini d'arbres (ou d'arborescences) disjoints est appelé une ***forêt.***

3. Types abstraits Graphes

Sur les exemples proposés ci-dessus, le graphe est donné une fois pour toutes et les opérations intéressantes sont le test de l'existence d'un arc (d'une arête) entre deux sommets, ou le test de l'existence d'un sommet parmi les successeurs d'un autre sommet, etc. De plus, dans bien des cas, on a besoin d'énumérer les successeurs d'un sommet : il est donc utile de connaître le demi-degré extérieur de tout sommet et le $i^{ième}$ successeur d'un sommet *(l'ordre d'énumération étant arbitraire).*

Mais, le plus souvent, le graphe est évolutif et l'on veut pouvoir lui appliquer des opérations de base qui sont : ajout ou suppression d'un sommet, ajout ou suppression d'un arc, etc. On retrouve toutes ces opérations dans la signature des graphes.

On va avoir deux types abstraits : un pour les graphes orientés, et un pour les graphes non orientés. On commence par donner le type sommet, qui est utilisé par ces deux types abstraits.

Pour distinguer les sommets d'un graphe, on les étiquette, soit par des chaînes de caractères, soit par des numéros. C'est cette deuxième convention qui est suivie dans la spécification ci-dessous.

sorte Sommet
utilise Entier
opérations
som : Entier $\rightarrow$ Sommet
$n°$: Sommet $\rightarrow$ Entier

axiomes
$n°(som(i)) = i$, pour tout entier i

Dans la suite, quand il n'y a pas d'ambiguïté, on ne fait pas la distinction entre un sommet et son numéro : on considère que les sommets sont des entiers.

3.1. Spécification des graphes orientés

Un graphe orienté est un ensemble de sommets et un ensemble d'arcs; les arcs sont des paires ordonnées de sommets. Il y a des similarités entre ce type abstrait et celui vu pour les ensembles au chapitre 7. Il y a cependant des différences, dues essentiellement au fait qu'il faut maintenir la cohérence de deux ensembles : ajouter ou retirer un sommet peut affecter l'ensemble des arcs; ajouter ou retirer un arc peut affecter l'ensemble des sommets.

On a pris les conventions suivantes :

- quand on ajoute un sommet, celui-ci est isolé (il n'a aucun arc incident);
- quand on ajoute un arc, si les sommets adjacents à cet arc n'appartiennent pas au graphe, on les ajoute;
- par contre, quand on retire un arc, les sommets adjacents ne sont pas retirés;
- mais quand on retire un sommet, tous les arcs incidents sont supprimés;
- les opérations d'ajout d'un sommet ou d'un arc ne sont pas définies si le sommet ou l'arc est déjà présent dans le graphe concerné.

La signature du type abstrait Graphe est la suivante :

sorte Graphe {*cas orienté*}
utilise Sommet, Entier, Booléen
opérations

graphe-vide	: → Graphe
ajouter-le-sommet _ à _	: Sommet × Graphe → Graphe
ajouter-l'arc < _,_ > à _	: Sommet × Sommet × Graphe → Graphe
_ est-un-sommet-de _	: Sommet × Graphe → Booléen
< _,_ > est-un-arc-de _	: Sommet × Sommet × Graphe → Booléen
$d^{\circ+}$ *de_ dans_*	: Sommet × Graphe → Entier
_ ème-succ-de _ dans _	: Entier × Sommet × Graphe → Sommet
$d^{\circ-}$ *de_ dans_*	: Sommet × Graphe → Entier
_ ème-pred-de _ dans _	: Entier × Sommet × Graphe → Sommet
retirer-le-sommet _ de _	: Sommet × Graphe → Graphe
retirer-l'arc < _,_ > de _	: Sommet × Sommet → Graphe

Dans ce qui suit les variables s, s', s'' sont de sorte Sommet, la variable g est de sorte Graphe, les variables i, j sont de sorte Entier.

Les domaines de définition des opérations sont spécifiés par les préconditions suivantes.

préconditions

ajouter-le-sommet s à g **est-défini-ssi** s *est-un-sommet-de* g = faux
ajouter-l'arc $< s, s' >$ à g **est-défini-ssi**
$s \neq s'$ & $(< s, s' >$ *est-un-arc-de* $g) =$ faux

$d^{\circ+}$ *de* s *dans* g **est-défini-ssi** s *est-un-sommet-de* g = vrai
i *ème-succ-de* s *dans* g **est-défini-ssi**
$(s$ *est-un-sommet-de* $g) =$ vrai & $(i \leq d^{\circ+}$ *de* s *dans* $g) =$ vrai

$d^{\circ-}$ *de* s *dans* g **est-défini-ssi** s *est-un-sommet-de* g = vrai
i *ème-pred-de* s *dans* g **est-défini-ssi**
$(s$ *est-un-sommet-de* $g) =$ vrai & $(i \leq d^{\circ-}$ *de* s *dans* $g) =$ vrai

retirer-le-sommet s *de* g **est-défini-ssi** s *est-un-sommet-de* g = vrai
retirer-l'arc $< s, s' >$ *de* g **est-défini-ssi** $< s, s' >$ *est-un-arc-de* g = vrai

Les axiomes sont présentés ci-dessous dans l'ordre suivant : définitions successives des observateurs *est-un-sommet-de*, *est-un-arc-de*, $d^{\circ+}$, pour le graphe vide et pour le résultat des opérations d'ajout de sommet et d'arc; puis définitions de ces observateurs sur les opérations de suppression d'un sommet ou d'un arc.

Les axiomes caractérisant le $i^{ième}$ successeur d'un sommet (*ème-succ-de*) sont laissés en exercice, ainsi que ceux caractérisant le demi-degré intérieur ($d^{\circ-}$ *de*) et le $i^{ième}$ prédécesseur d'un sommet (*ème-pred-de*).

Rappelons que l'on a pris la convention au chapitre 4 que les axiomes ont comme prémisses implicites les préconditions, convenablement instanciées, des opérations qui y apparaissent.

3.1.1. Définition de est-un-sommet-de sur le graphe vide et les ajouts

s *est-un-sommet-de graphe-vide* = faux

Le graphe vide correspond à un ensemble vide de sommets.

$s = s' \Rightarrow s$ *est-un-sommet-de* (*ajouter-le-sommet* s' *à* $g) =$ vrai
$s \neq s' \Rightarrow s$ *est-un-sommet-de* (*ajouter-le-sommet* s' *à* $g) =$
s *est-un-sommet-de* g

$s = s' \quad \vee \quad s = s'' \Rightarrow s$ *est-un-sommet-de* (*ajouter-l'arc* $< s', s'' >$ *à* $g)$
$=$ vrai

Quand on ajoute un arc, si les sommets adjacents à cet arc n'appartiennent pas au graphe, on les ajoute.

$s \neq s' \quad \& \quad s \neq s''$
$\Rightarrow s$ *est-un-sommet-de* (*ajouter-l'arc* $< s', s'' >$ *à* $g) =$
s *est-un-sommet-de* g

3.1.2. Définition de est-un-arc-de sur le graphe vide et les ajouts

$< s, s' >$ *est-un-arc-de graphe-vide* = faux

Le graphe vide ne contient aucun arc.

$< s, s' >$ *est-un-arc-de* (*ajouter-le-sommet* s'' à $g) =$
$< s, s' >$ *est-un-arc-de* g

Quand on ajoute un sommet, celui-ci est isolé : on n'ajoute aucun arc.

$s = s''$ & $s' = s'''$
$\Rightarrow < s, s' >$ *est-un-arc-de* (*ajouter-l'arc* $< s'', s''' >$ *à* g) = vrai
$s \neq s''$ $\vee$ $s' \neq s'''$
$\Rightarrow < s, s' >$ *est-un-arc-de* (*ajouter-l'arc* $< s'', s''' >$ *à* g) =
$< s, s' >$ *est-un-arc-de* g

3.1.3. Définition du demi-degré extérieur sur les ajouts

Le graphe vide ne contenant aucun arc, d^{o+} n'est jamais défini sur le graphe vide.

$s \neq s' \Rightarrow d^{o+}$ *de* s *dans* (*ajouter-le-sommet* s' *à* g) = d^{o+} *de* s *dans* g
$s = s' \Rightarrow d^{o+}$ *de* s *dans* (*ajouter-le-sommet* s' à g) = 0

Quand on ajoute un sommet, celui-ci est isolé.

$s \neq s'$ & $s \neq s'' \Rightarrow d^{o+}$ *de* s *dans* (*ajouter-l'arc* $< s', s'' >$ *à* g) =
d^{o+} *de* s *dans* g
$s = s'$ & s *est-un-sommet-de* g = vrai
$\Rightarrow d^{o+}$ *de* s *dans* (*ajouter-l'arc* $< s', s'' >$ *à* g) = (d^{o+} *de* s *dans* g)+1

s' était déjà dans g ; il a un successeur de plus.

$s = s'$ & s *est-un-sommet-de* g = faux
$\Rightarrow d^{o+}$ *de* s *dans* (*ajouter-l'arc* $< s', s'' >$ *à* g) = 1

s' n'était pas dans g ; on l'ajoute ; il a un successeur, s''.

$s = s''$ & s *est-un-sommet-de* g = vrai
$\Rightarrow d^{o+}$ *de* s *dans* (*ajouter-l'arc* $< s', s'' >$ *à* g) = (d^{o+} *de* s *dans* g)

s' était déjà dans g ; il a le même nombre de successeurs.

$s = s''$ & s *est-un-sommet-de* g = faux
$\Rightarrow d^{o+}$ *de* s *dans* (*ajouter-l'arc* $< s', s'' >$ *à* g) = 0

s' n'était pas dans g ; on l'ajoute ; il n'a aucun successeur.

3.1.4. Définition des observateurs sur le résultat de la suppression d'un sommet

$s = s' \Rightarrow s$ *est-un-sommet-de* (*retirer-le-sommet* s' *de* g) = faux
$s \neq s' \Rightarrow s$ *est-un-sommet-de* (*retirer-le-sommet* s' *de* g) =
s *est-un-sommet-de* g

$s = s'' \vee s' = s'' \Rightarrow < s, s' >$ *est-un-arc-de* (*retirer-le-sommet* s'' *de* g) = faux

Quand on retire un sommet, on supprime les arcs incidents.

$s \neq s''$ & $s' \neq s''$
$\Rightarrow < s, s' >$ *est-un-arc-de* (*retirer-le-sommet* s'' *de* g) =
$< s, s' >$ *est-un-arc-de* g

$< s, s' >$ *est-un-arc-de* g = vrai
$\Rightarrow d^{o+}$ *de* s *dans* (*retirer-le-sommet* s' *de* g) = (d^{o+} *de* s *dans* g) -1

$< s, s' >$ *est-un-arc-de* g = faux & s *est-un-sommet-de* g = vrai
$\Rightarrow d^{\circ+}$ *de* s *dans* (*retirer-le-sommet* s' *de* g) = $d^{\circ+}$ *de* s *dans* g

Les autres observateurs sont laissés en exercice.

3.1.5. Définition des observateurs sur le résultat de la suppression d'un arc

s *est-un-sommet-de* (*retirer-l'arc* $< s', s'' >$ *de* g) = s *est-un-sommet de* g
Quand on retire un arc on ne retire pas de sommet.

$s = s''$ & $s' = s'''$
$\Rightarrow <s, s'>$ *est-un-arc-de* (*retirer-l'arc* $< s'', s''' >$ *de* g) = faux

$s \neq s''$ ∨ $s' \neq s'''$
$\Rightarrow <s, s'>$ *est-un-arc-de* (*retirer-l'arc* $< s'', s''' >$ *de* g) =
$< s, s' >$ *est-un-arc-de* g

$s = s' \Rightarrow d^{\circ+}$ *de* s *dans* (*retirer-l'arc* $< s', s'' >$ *de* g) = ($d^{\circ+}$ *de* s *dans* g)−1
$s \neq s' \Rightarrow d^{\circ+}$ *de* s *dans* (*retirer-l'arc* $< s', s'' >$ *de* g) = $d^{\circ+}$ *de* s *dans* g

Les autres observateurs sont laissés en exercice. Ces axiomes terminent la définition du type abstrait Graphe, dans le cas orienté.

De plus, pour des raisons de simplicité dans la description du déroulement de certains algorithmes, on prend la *convention* que les successeurs d'un sommet sont numérotés de façon croissante :

$i < j \Rightarrow n^{\circ}$ (*i ème-succ-de* s *dans* g) $< n^{\circ}$ (*j ème-succ-de* s *dans* g)

3.2. Compléments à la spécification des graphes orientés

A partir de ces opérations de base, on peut définir d'autres opérations qui sont très utilisées dans les algorithmes travaillant sur un graphe orienté :

premsucc : Sommet × Graphe → Sommet

est définie pour tout sommet s et tout graphe g tels que $d^{\circ+}$ *de* s *dans* $g \geq 1$,

et on a :

$$premsucc(s, g) = 1 \text{ ème-succ-de } s \text{ dans } g$$

succsuivant : Sommet × Sommet × Graphe → Sommet

est définie pour tout couple de sommets s, s' et tout graphe g tels que :

il existe $i, 1 \leq i < d^{\circ+}$ *de* s *dans* g, tel que $s' = i$ *ème-succ-de* s *dans* g

et on a alors :

$$succsuivant(s, s', g) = (i + 1) \text{ ème-succ-de } s \text{ dans } g$$

Par ailleurs, on a vu que certains graphes sont valués. Cela signifie qu'on a une fonction de coût, de profil (si les coûts sont des nombres réels) :

coût : Sommet × Sommet × Graphe → Réel

Cette opération est définie pour toute paire ordonnée de sommets (s, s') et tout graphe g tels que $< s, s' >$ *est-un-arc-de* g = vrai.

Il faut modifier dans ce cas le profil de l'opération d'ajout d'un arc, car il faut prévoir le coût de cet arc en opérande :

ajouter-l'arc < _, _ > *de-coût* _ *à* _ :
Sommet × Sommet × Réel × Graphe → Graphe.

Pour programmer certains algorithmes, il est parfois commode de prolonger l'opération de coût à toutes les paires ordonnées de sommets du graphe, y compris celles qui ne correspondent pas à un arc : on lui fait rendre pour ces paires une valeur dont on sait qu'elle ne peut être un coût.

D'autres opérations peuvent être définies à partir de la spécification ci-dessus : par exemple les opérations *nb-sommets* et *nb-arcs* dont la spécification est laissée en exercice.

Très souvent, on rencontre dans les algorithmes sur les graphes le schéma suivant de traitement des successeurs d'un sommet s : «pour chaque sommet y, successeur de s, effectuer un certain traitement sur y». Ce schéma s'écrit en utilisant les opérations de base :

for $i := 1$ **to** d^{o+} *de s dans g* **do**
traiter(*i ème-succ-de s dans g*);

3.3. Spécification des graphes non orientés

On a la signature suivante :

sorte Graphe {*cas non orienté*}
utilise Sommet, Entier, Booléen
opérations

graphe-vide	: → Graphe
ajouter-le-sommet _ *à* _	: Sommet × Graphe → Graphe
ajouter-l'arête < _, _ > *à* _	: Sommet × Sommet × Graphe → Graphe
_ *est-un-sommet-de* _	: Sommet × Graphe → Booléen
< _, _ > *est-une-arête-de* _	: Sommet × Sommet × Graphe → Booléen

d° *de* _ *dans* _ : Sommet × Graphe → Entier
ème-succ-de _ *dans* _ : Entier × Sommet × Graphe → Sommet

retirer-le-sommet _ *de* _ : Sommet × Graphe → Graphe
retirer-l'arête < _, _ > *de* _ : Sommet × Sommet → Graphe

On a des axiomes similaires à ceux du cas orienté en remplaçant partout *arc* par *arête* et d^{o+} par d^o. Il faut cependant modifier et ajouter des axiomes, car on doit avoir pour tous sommets s, s' et tout graphe g :

$$< s, s' > \textit{ est-une-arête-de } g =< s', s > \textit{ est-une-arête-de } g$$

Par exemple, quand on ajoute une arête $\{s, s'\}$, cela a pour conséquence que $< s, s' >$ *est-une-arête-de* $g =$ vrai et $< s', s >$ *est-une-arête-de* $g =$ vrai ; de même, le degré de s et le degré de s' sont tous deux augmentés de 1, ou initialisés à 1 s'il s'agit d'un nouveau sommet.

On définit les opérations *premsucc* et *succsuivant*, et le schéma de traitement des successeurs d'un sommet et la fonction de coût, comme dans le cas des graphes orientés.

Les axiomes de cette spécification sont laissés en exercice.

4. Représentations des graphes

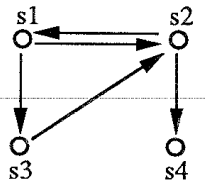
On peut représenter les graphes de plusieurs manières. On peut distinguer deux grandes classes de représentations, selon que l'on privilégie le fait qu'un graphe est un ensemble d'arcs (resp. arêtes) ou un ensemble de sommets.

4.1. Utilisation de matrices

Cette représentation correspond au cas où l'ensemble des sommets du graphe n'évolue pas ; on représente l'ensemble des arcs par un tableau de booléens (cf. chapitre 6) ; comme chaque arc est une paire ordonnée de sommets, le graphe est représenté par une matrice carrée de booléens, dite ***matrice d'adjacence***, de dimension $n \times n$ si le graphe a n sommets. On a donc le type Pascal suivant :

type GRAPHE = **array** $[1..n, 1..n]$ **of** boolean ;

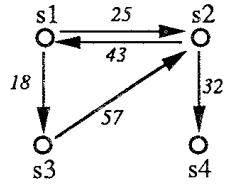
L'élément d'indices i et j de la matrice est vrai si, et seulement si, il existe un arc entre les sommets i et j. La figure 8 donne un exemple de graphe avec sa représentation sous forme matricielle.



	1	2	3	4
1	F	T	T	F
2	T	F	F	T
3	F	T	F	F
4	F	F	F	F

Figure 8. Représentation d'un graphe par une matrice de valeurs booléennes.

Dans le cas où le graphe est non orienté, la matrice est symétrique. Dans le cas où le graphe est valué, on utilise une matrice où l'élément d'indices i et j a pour valeur le poids de l'arc du sommet i au sommet j (resp. de l'arête entre les sommets i et j), si cet arc (resp. arête) existe, et sinon une valeur dont on sait qu'elle ne peut être un poids : par exemple, le plus grand entier utilisable si les poids sont des entiers bornés supérieurement. La figure 9 montre la représentation d'un graphe orienté, valué par des entiers positifs, par une matrice d'entiers où la valeur -1 indique qu'il n'y a pas d'arc entre les sommets correspondants.



-1	25	18	-1
43	-1	-1	32
-1	57	-1	-1
-1	-1	-1	-1

Figure 9. Représentation d'un graphe par la matrice des coûts de ses arcs.

La représentation matricielle est pratique pour tester l'existence d'un arc (ou d'une arête) entre deux sommets : on accède directement à l'élément de la matrice (en un temps constant). De même, il est facile d'ajouter ou de retirer un arc ou une arête. Il est également facile de parcourir tous les successeurs ou prédécesseurs d'un sommet. Pour traiter tous les successeurs du sommet i, on a le schéma suivant, où G est la matrice d'adjacence du graphe :

for $j := 1$ **to** n **do**
  if $G[i,j]$ **then** *traiter* (j);

Cependant, ce schéma demande n tests quel que soit le nombre de successeurs de i. Il en est de même du calcul de $d^{\circ +}$ ou de $d^{\circ -}$. Une consultation complète de la matrice requiert un temps d'ordre n^2, et cette représentation exige un espace mémoire de $\Theta(n^2)$ si le graphe a n sommets, quel que soit le nombre d'arcs ou d'arêtes du graphe. Cela interdit d'avoir des algorithmes d'ordre inférieur à n^2 pour des graphes de n sommets, n'ayant que peu d'arcs. Pour remédier à cet inconvénient, on utilise dans ce cas une représentation appelée «par listes d'adjacence».

4.2. Utilisation de listes d'adjacence

Une autre représentation classique des graphes consiste à représenter l'ensemble des sommets et à associer à chaque sommet la liste de ses successeurs rangés dans un certain ordre. Ces listes sont appelées ***listes d'adjacence.***

Dans le cas où l'ensemble des sommets n'évolue pas, ces listes sont accessibles à partir d'un tableau S, qui contient pour chaque sommet, un pointeur vers le début de sa liste. La figure 10 donne la représentation sous forme de listes d'adjacence du graphe de la figure 8. Le type Pascal correspondant est le suivant :

```
type adr = ↑doublet;
     doublet = record no : 1..n; suiv : adr end;
     GRAPHE = array [1..n] of adr;
```

Il est facile de prendre en compte l'existence de poids sur les arcs en ajoutant un champ au type doublet.

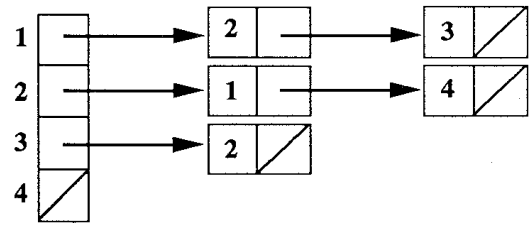


Figure 10. Représentation d'un graphe par listes d'adjacence.

L'intérêt de cette représentation est que l'espace mémoire utilisé est, pour un graphe orienté avec n sommets et p arcs, en $\Theta(n+p)$. Dans le cas d'un graphe non orienté avec p arêtes, l'espace mémoire est en $\Theta(n+2p)$. De plus, quand on a besoin de faire un traitement sur les successeurs d'un sommet s, le nombre de sommets parcourus est exactement le nombre de successeurs de s, soit $d^{o+}(s)$. Le schéma du traitement de tous les successeurs du sommet i est donné ci-dessous :

```
v := S[i];
while v ≠ nil do begin
      j := v↑. no; traiter(j); v := v↑. suiv
end;
```

Le calcul de $d^{\circ+}$ se fait suivant ce schéma.

Un algorithme qui traite tous les arcs d'un graphe de p arcs peut donc être d'ordre p.

En revanche, cette représentation présente l'inconvénient d'exiger, dans le pire des cas, un temps d'ordre n pour tester s'il existe un arc (resp. une arête) entre un sommet donné x et un sommet y (cas où la liste d'adjacence est de longueur $n-1$ et où y est en fin de liste) ou pour l'ajout d'un arc ou d'une arête (avec test de non

répétition). Enfin, on note que cette représentation, contrairement aux matrices, ne permet pas de calculer facilement les opérations relatives aux prédécesseurs ($d^{\circ -}$ et *ème-pred-de*).

Remarque 1 : Ces deux méthodes de représentation, matrice d'adjacence et listes d'adjacence, sont valables que les graphes soient orientés ou non. Dans le cas d'un graphe non orienté, on représente l'arête x—y par deux arcs $x \rightarrow y$ et $y \rightarrow x$. En conséquence, la matrice d'adjacence d'un graphe non orienté est symétrique. Dans le cas d'une représentation par listes d'adjacence, chaque fois qu'on trouve le sommet y sur la liste d'adjacence du sommet x, on trouve le sommet x sur la liste d'adjacence du sommet y (s'il y a p arêtes, on a donc $2p$ doublets).

Remarque 2 : Il existe des variantes de ces deux représentations. Par exemple, la représentation par listes d'adjacence peut se faire en utilisant un tableau pour ranger les listes chaînées, comme on l'a vu au chapitre 5 (cf. exercices). Inversement, on peut vouloir représenter le tableau des têtes de listes par une liste chaînée, dans le cas où on doit ajouter ou supprimer des sommets (cf. exercices).

Remarque 3 : On peut aussi représenter un graphe par listes d'adjacence des prédécesseurs de chaque sommet. Pour certains algorithmes, il est même parfois efficace (en temps, mais coûteux en mémoire) d'avoir à la fois la liste des successeurs et la liste des prédécesseurs de chaque sommet. Le tableau de la figure 10 contient alors deux têtes de liste.

5. Parcours de graphes

Beaucoup de problèmes sur les graphes nécessitent un examen exhaustif des sommets et des arcs (ou arêtes) du graphe. On en verra des exemples comme le tri topologique au chapitre 18 et les connexités au chapitre 19. On va étudier deux types de parcours qui correspondent à des stratégies d'exploration très générales :

• ***Le parcours en profondeur*** (en anglais depth-first search) consiste, à partir d'un sommet donné, à suivre un chemin le plus loin possible, puis à faire des retours en arrière pour reprendre tous les chemins ignorés précédemment. On verra qu'un tel parcours se conçoit naturellement de façon récursive.

• ***Le parcours en largeur*** (en anglais breadth-first search) consiste à explorer le graphe niveau par niveau, à partir d'un sommet donné. Ce parcours, quant à lui, est intrinsèquement itératif.

5.1. Parcours en profondeur : version récursive

On considère un graphe orienté dont tous les sommets sont initialement non marqués. Le parcours en profondeur consiste à choisir un sommet de départ s, à le marquer et

à suivre un chemin issu de s, aussi loin que possible, en marquant les sommets au fur et à mesure qu'on les rencontre. Lorsqu'on est en fin de chemin, on revient au dernier choix fait et on prend une autre direction. On donne ci-après la procédure récursive de parcours en profondeur *prof*. Lors de l'exécution de la procédure *prof* appelée pour le sommet s, tous les sommets qui n'ont pas été marqués avant s et qui peuvent être atteints à partir de s sont visités, et rien qu'eux.

Pour obtenir le parcours total du graphe, il faut un programme principal qui appelle la procédure pour un sommet non marqué, et qui recommence à l'issue de ce parcours tant qu'il reste des sommets non marqués.

5.1.1. Algorithme abstrait

Les programmes qui suivent font appel aux opérations et au schéma de traitement des successeurs du type abstrait donné au paragraphe 3.

Programme principal :

```
var i : integer; gr : Graphe; marque : array [1..n] of boolean;
...
begin
    for i := 1 to n do marque [i] := false;
    for i := 1 to n do
        if not (marque [i] ) then prof(i, gr, marque)
end;
```

Procédure récursive :

```
procedure prof(s : integer; G : Graphe; var M : array [1..n] of boolean);
{s est un sommet }
var j, v : integer; {v est un sommet }
begin
    M[s] := true;
1   {première rencontre de s}
    for j := 1 to d°+ de s dans G do
    begin
      v :=j ème-succ-de s dans G; {rencontre de l'arc (s,v) à l'aller}
      if not (M[v]) then prof(v, G, M); {rencontre de l'arc (s,v) au retour}
    end
2   { dernière rencontre de s}
end prof;
```

On illustre cet algorithme en montrant son fonctionnement sur l'exemple de la figure 11.

Exemple : On suppose que l'ordre d'exploration des successeurs d'un sommet du graphe de la figure 11, est l'ordre des numéros croissants.

• Si on insère dans la procédure *prof*, une instruction d'écriture de s, au moment de la première rencontre, ligne n° 1, on obtient l'impression des sommets dans l'ordre : $s1, s3, s2, s6, s5, s7, s4, s9, s8$.

• Par contre, si cette instruction d'écriture est insérée au moment de la dernière rencontre, ligne n° 2, on obtient : $s2, s6, s3, s5, s7, s1, s9, s4, s8$.

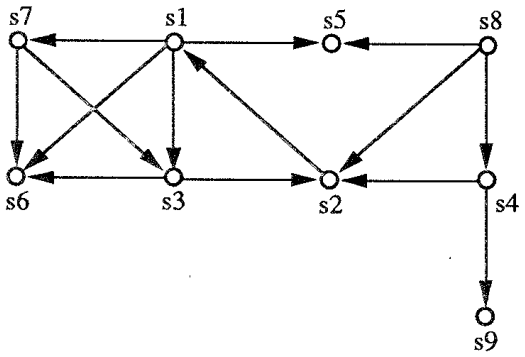


Figure 11.

Pour un parcours donné, tous les appels de *prof* se font avec les mêmes paramètres G et M. Dans tout ce qui suit, lorsqu'il n'y a pas d'ambiguïté, *prof*(s, G, M) est abrégé par *prof*(s).

L'algorithme commence par l'appel de *prof*$(s1)$ qui marque $s1$, le premier sommet du graphe puis choisit $s3$, le premier successeur de $s1$. Comme $s3$ est non marqué, il y a appel récursif de *prof*$(s3)$. Le sommet $s3$ est alors marqué, et son premier successeur $s2$ est choisi. Comme $s2$ est non marqué, il y a appel de *prof*$(s2)$. Le sommet $s2$ est marqué et son premier successeur $s1$ est choisi. Comme $s1$ est déjà marqué, et que $s2$ n'a que ce seul successeur, la procédure *prof*$(s2)$ s'arrête et la visite des sommets reprend au deuxième successeur de $s3$, le sommet $s6$; $s6$ n'ayant pas de successeur, *prof*$(s6)$ marque $s6$ et s'arrête tout de suite. Tous les successeurs de $s3$ ont été explorés, donc *prof*$(s3)$ se termine. *prof*$(s1)$ se poursuit par le choix de $s5$, qui n'est pas encore marqué. L'appel *prof*$(s5)$ marque $s5$ mais se termine tout de suite, $s5$ n'ayant pas de successeur. Le successeur suivant de $s1$ est $s6$ qui est déjà marqué ; *prof*$(s1)$ considère donc le dernier successeur $s7$ de $s1$ qui reste non marqué. La procédure *prof*$(s7)$ marque $s7$ et se termine tout de suite, les successeurs de $s7$ ayant déjà été marqués. *prof*$(s1)$ est maintenant terminée : il y a retour dans le programme principal. Le prochain sommet non encore marqué étant $s4$, il y a appel de *prof*$(s4)$. Le premier successeur de $s4$, soit $s2$ étant déjà marqué, *prof*$(s4)$ marque $s4$ et appelle *prof*$(s9)$, qui marque $s9$ et se termine ensuite. La procédure *prof*$(s4)$ étant terminée, il y a retour dans le programme principal et le sommet $s8$ est choisi. *prof*$(s8)$ marque $s8$ et se termine ensuite, les successeurs $s2, s4, s5$ de $s8$ étant déjà marqués. Tous les sommets étant marqués, l'algorithme se termine.

On donne ci-dessous deux versions de l'algorithme de parcours en profondeur selon que le graphe est représenté par matrice d'adjacence ou par listes d'adjacence.

5.1.2. Représentation par matrice d'adjacence

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

    procedure prof(i : integer; G : GRAPHE; var M : array [1..n]
    of boolean);
    var j : integer;
    begin
        M[i] := true;
        for j := 1 to n do
            if G[i, j] and not (M[j]) then prof(j, G, M)
    end prof;
```

5.1.3. Représentation par listes d'adjacence

```
type adr = ↑doublet;
    doublet = record no : 1..n; suiv : adr end;
    GRAPHE = array [1..n] of adr;
    procedure prof(i : integer; G : GRAPHE; var M : array [1..n]
    of boolean);
    var j : integer; s : adr;
    begin
        M[i] := true;
        s := G[i];
        while s ≠ nil do begin
                j := s↑.no;
                if not (M[j]) then prof(j, G, M);
                s := s↑.suiv
        end
    end prof;
```

5.1.4. Analyse de la complexité du parcours en profondeur

Généralement, dans les algorithmes de traitement des graphes, on mesure la *complexité en fonction de deux paramètres,* d'une part le nombre n de sommets du graphe, d'autre part le nombre p d'arcs (ou d'arêtes) du graphe.

Supposons qu'on effectue un parcours en profondeur sur un graphe orienté ayant n sommets et p arcs ($p \leq n^2$). Chaque sommet est marqué une et une seule fois, si bien qu'il y a n appels de la procédure *prof.* L'examen des marques se fait exactement n fois dans le programme principal; dans la procédure *prof,* il se fait

pour chaque successeur y d'un sommet x, et ceci pour tous les sommets x. Dans le cas de la représentation par listes d'adjacence, cela revient à parcourir l'ensemble des listes de successeurs : le temps requis est en $\Theta(p)$. Dans le cas de la représentation par matrices d'adjacence, la consultation de la matrice exige un temps en $\Theta(n^2)$. En conséquence, avec une représentation par listes d'adjacence, le temps total requis est en $\Theta(\max(n,p))$, et avec une représentation par matrices d'adjacence, il est en $\Theta(n^2)$.

5.2. Parcours en profondeur : version itérative

Dans ce paragraphe, on décrit une version itérative de la procédure *prof* du parcours en profondeur d'un graphe. Cet algorithme utilise les opérations des types abstraits Graphe et Pile.

Comme l'appel récursif dans *prof* se trouve dans une boucle, il faut garder le sommet visité, pour savoir où reprendre la boucle. A la place de chaque appel de *prof* sur v, il faut empiler ce sommet v. On remarque que si deux sommets s et v se suivent dans la pile, c'est que v est un successeur de s. Lorsqu'on a épuisé la visite le long d'un chemin en profondeur des descendants d'un sommet (booléen *poursuite* mis à false), il faut revenir en arrière. Pour cela, il faut mémoriser le sommet de pile, soit v; puis dépiler pour obtenir le sommet s. Le parcours se poursuit alors en considérant le successeur de s qui vient après v, pour une visite en profondeur de ses successeurs. Pour obtenir le parcours du graphe tout entier, il faut ajouter le programme principal donné au §5.1. en remplaçant l'appel de *prof* par l'appel de *iterprof*.

```
procedure iterprof(s: integer; G: Graphe; var M: array [1..n] of boolean);
{s est un sommet}
var i, j, d : integer; poursuite : boolean; Q : Pile;
{i et j sont des sommets}
begin
   i := s; Q := pile-vide; poursuite : = false; M[i] := true;
       {1ère rencontre de i}
   Q := empiler (Q, i);
   if d°+ de i dans G <> 0 then begin
     poursuite : = true; j := premsucc(i, G)
   end;
   while not (est-vide (Q)) do begin
     while poursuite do
           if not (M[j]) then begin {j n'est pas marqué : on progresse}
             M[j] := true; { 1ère rencontre de j}
             Q := empiler (Q, j); i := j; {on passe aux successeurs}
             if d°+ de i dans G > 0 then j := premsucc(i, G)
             else poursuite : = false {pas de successeurs}
           end
```

```
                else begin {j est marqué : on regarde les autres successeurs de i}
                d := d°+ de i dans G;
                if j <> d ème-succ-de i dans G then j := succsuivant(i, j, G)
                else poursuite : = false {il n'y a plus d'autres successeurs de i}
                end;
        {retour arrière :}
        j := sommet(Q); Q := dépiler(Q); {dernière rencontre de j}
        if not (est-vide(Q)) then begin
                i := sommet(Q);
                d := d°+ de i dans G;
                if j <> d ème-succ-de i dans G then begin
                   j := succsuivant(i, j, G); poursuite : = true
                end
                {sinon on a marqué tous les successeurs de i;
                dernière rencontre de i; on continue le retour arrière}
        end
    end
end iterprof;
```

Dans le cas d'une représentation du graphe par une matrice d'adjacence ou par des listes d'adjacence, on obtient des programmes plus simples (voir exercices).

5.3. Forêt couvrante associée au parcours en profondeur

Durant le parcours en profondeur d'un graphe orienté, on distingue différents types d'arcs.

• Les arcs $x \rightarrow y$ tels que *prof*(x) appelle *prof*(y) sont nommés ***arcs couvrants.*** Les arcs couvrants constituent une ***forêt couvrante d'arborescences de recherche en profondeur.***

Soit r un sommet tel que *prof*(r) est appelée par le programme principal ; considérons le sous-graphe G' des arcs couvrants résultant de l'exécution de *prof*(r). Le graphe non orienté sous-jacent à G' est clairement connexe. Comme la procédure *prof* est appelée exactement une fois pour chaque sommet s de G' qui n'est pas r, $d^{\circ -}$ *de* s *dans* $G' = 1$ pour $s \neq r$; de plus, $d^{\circ -}$ *de* r *dans* $G' = 0$ (car *prof*(r) est appelée par le programme principal). Or, un graphe orienté G' dont le graphe non orienté sous-jacent est connexe et tel que tous ses sommets sont de demi-degré intérieur égal à 1, sauf un, qui est lui, de demi-degré intérieur nul, est une arborescence de racine r (voir exercices). Par exemple, sur la figure 12, l'arc $s1 \rightarrow s3$ est un arc couvrant.

• Les arcs dont l'extrémité terminale est un ascendant (au sens de la terminologie des arbres planaires) de l'extrémité initiale, dans la forêt sont appelés ***arcs en arrière.*** Par exemple, sur la figure 12, l'arc $s2 \rightarrow s1$ est un arc en arrière.

• Les arcs dont l'extrémité terminale est un descendant, dans la forêt, de l'extrémité initiale sont appelés ***arcs en avant.*** Par exemple, sur la figure 12, l'arc $s1 \rightarrow s6$ est un arc en avant.

• Les arcs pour lesquels il n'existe pas de chemin entre leurs extrémités dans la forêt sont appelés des ***arcs croisés.*** Sur la figure 12, $s8 \rightarrow s5$ est un arc croisé.

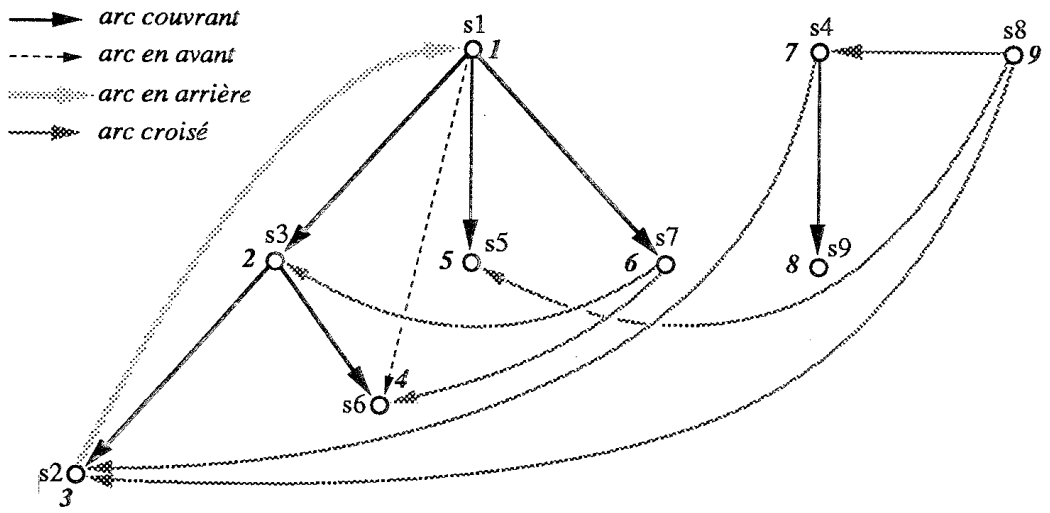


Figure 12. Forêt couvrante du graphe de la figure 11 et numérotation préfixe.

Si l'on prend la convention de dessiner les différents arcs de la forêt au fur et à mesure qu'ils sont empruntés et les différentes arborescences de la gauche vers la droite, au fur et à mesure qu'elles sont constituées par l'algorithme, les arcs croisés sont nécessairement dirigés de la droite vers la gauche.

En effet, raisonnons sur la figure 12. Supposons par exemple, qu'au lieu d'avoir un arc croisé $s7 \rightarrow s3$, on ait un arc $s3 \rightarrow s7$, et que par conséquent, $s7$ soit un successeur de $s3$. Comme l'arc $s1 \rightarrow s7$ a été dessiné après l'arc $s1 \rightarrow s3$, au moment où on visite $s3$, le sommet $s7$ n'est pas encore visité. Comme $s7$ est un successeur de $s3$, *prof*($s3$) doit comporter la visite de $s7$, et l'arc $s3 \rightarrow s7$ devrait être un arc couvrant et non pas un arc croisé.

Considérons une numérotation des sommets qui respecte l'ordre dans lequel ils sont marqués lors du parcours. Pour cela, il suffit d'ajouter dans la procédure *prof*, avant la boucle **for** (ligne 1), les instructions de comptage suivantes :

$compt[i] := cpt;$
$cpt := cpt + 1;$

et d'initialiser à 1 le compteur *cpt* dans le programme principal. La numérotation du sommet i est donnée alors par $compt[i]$. On obtient les propriétés suivantes :

- un sommet y est l'un des k descendants d'un sommet x dans la forêt si, et seulement si : $compt[x] < compt[y] \leq compt[x] + k$
- si l'arc $x \rightarrow y$ est un arc couvrant alors $compt[x] < compt[y]$
- si l'arc $x \rightarrow y$ est un arc en avant alors $compt[x] < compt[y]$
- si l'arc $x \rightarrow y$ est un arc en arrière alors $compt[x] > compt[y]$
- si l'arc $x \rightarrow y$ est un arc croisé alors $compt[x] > compt[y]$

La démonstration de ces propriétés est laissée en exercice.

Sur la figure 12 on a indiqué les valeurs de *compt* pour chaque sommet. Cette numérotation correspond à l'ordre de première rencontre des sommets, qui a été donné pour les sommets du graphe de la figure 11. Il s'agit de l'ordre préfixe sur la forêt couvrante (considérée comme une forêt d'arbres planaires). De même, l'ordre suffixe sur cette forêt couvrante correspond à l'ordre de dernière rencontre des sommets. Pour l'obtenir il faut mettre les instructions de numérotation après la boucle **for** (ligne n° 2).

5.4. Parcours en profondeur d'un graphe non orienté

L'algorithme de parcours en profondeur donné est valable aussi bien dans le cas orienté que dans le cas non orienté. La complexité de l'algorithme dans le cas non orienté est du même ordre : si le graphe a p arêtes, cela revient à considérer qu'il a $2p$ arcs. La complexité est donc en $\Theta(\max\ (n, 2p))$ pour une représentation par listes d'adjacence, et en $\Theta(n^2)$ pour une représentation par matrice d'adjacence.

Notons que le sens du parcours implique une orientation des arêtes : on parle donc encore d'arcs pour la forêt couvrante. La forêt couvrante associée à un parcours en profondeur d'un graphe non orienté comporte autant d'arborescences que le graphe a de composantes connexes. On ne distingue plus dans ce cas que deux types d'arcs :

- les ***arcs couvrants*** : ce sont les arcs $x \rightarrow y$ tels que *prof*(x) appelle directement *prof*(y).
- les ***arcs en arrière*** : ce sont les arcs $x \dashrightarrow y$ tels que x est un descendant de y dans la forêt. (Attention si l'arête y—x est devenue un arc couvrant, on ne doit pas considérer l'arête x—y quand on construit les arcs en arrière. L'examen de l'arête x—y n'a pas été suivi par un appel récursif de *prof*.)

Il n'y a plus d'arcs en avant, puisque les arêtes ne sont pas orientées.

Il n'y a pas non plus la notion d'arc croisé. En effet, supposons qu'il y ait un arc croisé (au sens des graphes orientés) de y vers x, c'est-à-dire que x et y ne soient pas descendants l'un de l'autre mais soient adjacents, et raisonnons par l'absurde. Si x est visité avant y, *prof*(x) est appelée avant *prof*(y) ; comme la procédure *prof*(x)

ne peut se terminer sans avoir visité tous les sommets accessibles à partir de x, nécessairement y est visité lors de $prof(x)$, si bien que y est un descendant de x dans l'arborescence; d'où la contradiction.

Dans la figure 13, on a donné le graphe non orienté associé au graphe de la figure 11 et on a dessiné la forêt couvrante obtenue par un parcours en profondeur du graphe (toujours avec la convention que les sommets et les successeurs sont choisis par ordre d'indices croissants). Comme le graphe est connexe, on obtient une seule arborescence.

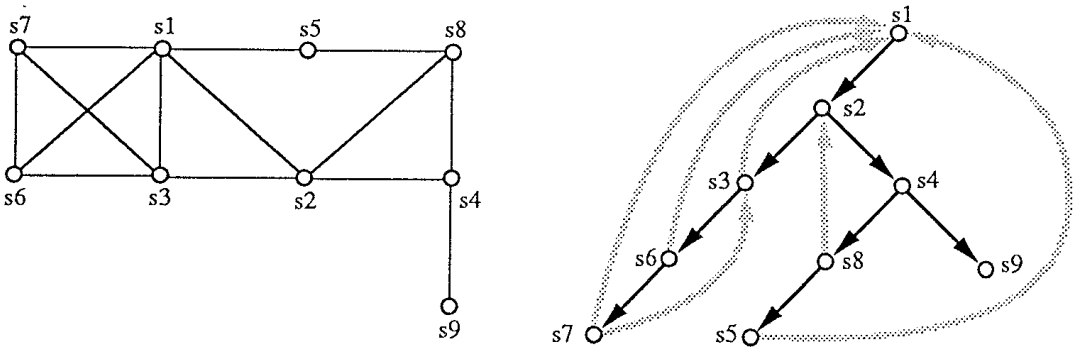


Figure 13. Exemple de forêt couvrante associée à un graphe non orienté.

5.5. Parcours en largeur

On appelle ce parcours, ***parcours en largeur***, car pour un sommet de départ s on commence par visiter tous les successeurs de s avant de visiter les autres descendants de s. Appelons distance de s à y, la longueur d'un plus court chemin issu de s et allant vers y. Le parcours en largeur consiste à visiter d'abord tous les sommets qui sont à la distance 1 de s, puis ceux qui sont à la distance 2, puis à la distance 3 et ainsi de suite.

Un parcours en largeur du graphe de la figure 11, à partir du sommet $s1$, visite dans l'ordre :

$s1$ (sommet de départ)

$s3, s5, s6, s7$ (sommets dont la distance à $s1$ vaut 1)

$s2$ (sommets non encore visités dont la distance à $s1$ vaut 2)

Pour programmer l'algorithme de parcours en largeur de façon itérative, on utilise une structure de file : en effet, lorsque à partir d'un sommet s, on visite ses successeurs non marqués, il est nécessaire de les ranger successivement dans une file

(FIFO) puisque la recherche au niveau suivant repartira de chacun des successeurs de s, à partir du premier.

L'algorithme qui suit, utilise les opérations des types abstraits donnés pour les graphes et le type File.

```
procedure larg(s : integer; G : Graphe; var M : array [1..n] of boolean);
{s est un sommet}
var v, w, i : integer; F : File;
{v et w sont des sommets}
begin
    F := file-vide; M[s] := true;
    F := ajouter(F, s);
    while not (est-vide(F)) do begin
        v := premier(F); F := retirer(F);
        for i := 1 to d^o+ de v dans G do begin
            w := i ème-succ-de v dans G;
            if not (M[w]) then begin
                M[w] := true; F := ajouter(F, w)
            end
        end
    end
end larg;
```

Comme pour la procédure *prof*, lors de l'exécution de *larg*, appelée pour s, on visite tous les sommets qui peuvent être atteints à partir de s et qui n'ont pas encore été marqués, et eux seulement.

Pour parcourir tout le graphe, il faut ajouter dans le programme principal, une boucle sur tous les sommets du graphe, analogue à celle du programme principal du parcours en profondeur. Les sommets du graphe de la figure 11 sont alors visités dans l'ordre suivant : $s1, s3, s5, s6, s7, s2, s4, s9, s8$.

Les procédures obtenues lorsque le graphe est implémenté par matrice d'adjacence ou listes d'adjacence s'écrivent aisément : elles sont laissées en exercice.

La complexité du parcours en largeur est la même que celle du parcours en profondeur.

Comme pour le parcours en profondeur, on peut associer au parcours en largeur une forêt couvrante. Cette forêt couvrante peut être utilisée, par exemple, pour déterminer les composantes connexes dans des graphes non orientés.

Selon les applications on utilise soit le parcours en profondeur (tri topologique, plus courts chemins, connexités...), soit le parcours en largeur (algorithmes de diffusion, algorithmes distribués...).