Chapitre 5

PARCOURS DE GRAPHES

1 Représentation informatique d'un graphe

Définition 5.1 — Taille d'un graphe

On appelle *taille* d'un graphe $G = (V, E)$ la quantité $|V| + |E|$.

Remarques

- Cette quantité n'a aucune signification « mathématique » (on additionne des choux et des carottes...) mais sera est pertinente informatiquement : on dira par exemple qu'un algorithme qui s'exécute en temps $O(|E| + |V|)$ est *linéaire en la taille du graphe*.
- La taille est au plus de l'ordre de $|V|^2$ (cas d'un graphe dense), mais est souvent nettement plus petite (graphe creux).

1.1 Représentation par matrice d'adjacence

La manière la plus simple de représenter un graphe est d'utiliser sa matrice d'adjacence : dans le cas assez fréquent où les étiquettes des nœuds sont les entiers de $[0 \ldots n-1]$, le graphe *est* sa matrice d'adjacence. Dans le cas général où les étiquettes sont de type `'a` (où `'a` peut correspondre à une page web, un fichier, une liste d'entiers...), il faudra simplement disposer en plus d'un `'a array` de taille n associant au numéro d'un nœud son étiquette.

Avantages

- Le test d'adjacence de deux sommets se fait en temps constant.
- Dans le cas d'un graphe orienté, obtenir les prédécesseurs n'est pas plus compliqué (ni plus coûteux) que d'obtenir les successeurs.
- Rajouter ou supprimer une arête (ou un arc) peut se faire en temps constant.
- Il est possible de n'utiliser qu'un bit par coefficient de la matrice : dans le cas d'un graphe dense, une telle représentation est donc la plus compacte possible.

Inconvénients

- Ajouter ou supprimer un sommet nécessite un temps proportionnel à n^2.
- Pour récupérer les voisins/successeurs d'un nœud, il faut parcourir toute la ligne de la matrice : l'opération se fait donc en $\Theta(n)$, ce qui est regrettable si le degré sortant du nœud est petit devant n.
- On consomme une mémoire proportionnelle à $n^2 = |V|^2$, même quand la taille du graphe est de l'ordre de n (graphe creux).

En résumé, cette représentation est bien adaptée aux graphes denses, mais presque inutilisable pour les graphes creux.

Exemple 5.1

Si l'on considère le graphe d'un gros réseau social, l'ordre de grandeur du nombre de sommets sera sans doute de 10^9 et celui du degré moyen de 10^2. La matrice d'adjacence aura alors 10^{18} entrées, dont environ 99,999 99% valent zéro : tout stocker n'est pas raisonnable ! Même à raison d'un bit par coefficient, la matrice utiliserait environ 10^{17} octets, c'est-à-dire 100 000 téraoctets.

1.2 Représentation par tableau de listes d'adjacence

Essentiellement, les problèmes liés à la représentation d'un graphe par sa matrice d'adjacence sont simplement ceux liés à la représentation d'une *matrice creuse* (c'est-à-dire majoritairement constituée de zéros). La solution la plus classique à ce problème est de stocker la ligne i de la matrice sous la forme d'une liste `[(j_1, x_1), ..., (j_k, x_k)]` où les j_k sont les indices tels que m_{i,j_k} soit non nul, et les x_k les valeurs correspondantes. Ici, la seule valeur non nulle possible est 1 et il est donc inutile de stocker les x_k : on se retrouve simplement avec la liste des j_k, où les j_k sont les numéros des voisins/successeurs du nœud numéro i.

Un graphe G à n sommets sera donc stocké sous la forme d'un **`int list array`** de longueur n, et G sera non orienté si et seulement si $i \in t.(j) \Rightarrow j \in t.(i)$ pour tous $i, j \in [0 \dots n]$.

Avantages

- La mémoire utilisée est de l'ordre de $|E| + |V|$, ce qui est nettement mieux que $|V|^2$ si le graphe est creux.
- On a directement accès à la liste des voisins d'un nœud : la parcourir prend un temps proportionnel au degré sortant du nœud (et pas à $|V|$).
- Ajouter un nœud peut normalement se faire en temps $|V|$ (si l'ajout n'impose pas de renuméroter les nœuds déjà présents).

Inconvénients

- Si l'on a besoin d'un accès en temps raisonnable aux prédécesseurs d'un nœud (dans le cas orienté), il faut stocker séparément le tableau de listes correspondant.
- Le test d'adjacence ne se fait plus en temps constant (mais en temps proportionnel au degré du nœud).
- Si le graphe est dense, on consommera plus de mémoire (d'un facteur constant) qu'avec une matrice d'adjacence.
- Ajouter ou supprimer une arête n'est pas aussi évident que dans une matrice d'adjacence : suivant l'opération précise que l'on souhaite faire, la complexité peut être unitaire ou proportionnelle aux degrés des nœuds impactés.
- Supprimer un nœud n'est pas pratique : le plus simple est de reconstruire entièrement le graphe (en un temps $\Theta(|E| + |V|)$).

C'est cette représentation que nous utiliserons le plus souvent, et c'est aussi celle avec laquelle sont habituellement formulés les algorithmes agissant sur les graphes. Il faut cependant signaler que, en « conditions réelles », on préférerait presque systématiquement des représentations à base de dictionnaires ou d'ensembles (réalisés par des arbres binaires de recherche ou des tables de hachage). En effet, une telle représentation est à la fois plus pratique à utiliser et plus efficace pour les tests d'adjacence et les modifications du graphe.

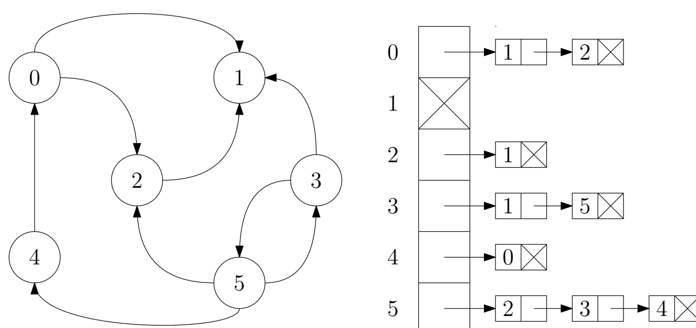


Figure 5.1 – Un graphe orienté et le tableau de listes d'adjacence correspondant.

1.3 Graphes implicites

Le plus souvent, il n'est pas nécessaire de stocker explicitement la structure du graphe : il suffit d'être capable de générer la liste des voisins d'un nœud donné à la demande. De cette manière, les algorithmes présentés dans ce chapitre peuvent être étendus au cas de graphes infinis (ou trop grand pour être stockés) – avec toutefois quelques subtilités supplémentaires.

Exemple 5.2

Les configurations possibles d'un *Rubik's Cube* forment un graphe à environ $4.3 \cdot 10^{19}$ sommets, qu'il serait pour le moins malaisé de stocker explicitement. Cependant, chaque sommet de ce graphe est de degré 12 (s'il on considère qu'un mouvement élémentaire consiste en la rotation de l'une des 6 faces d'un quart de tour, dans un sens ou dans l'autre), et il n'est pas difficile, à partir de l'étiquette d'une configuration[a], de calculer les étiquettes de ses voisins. Àinsi, il est tout-à-fait possible d'opérer effectivement sur ce graphe (par exemple pour chercher le nombre minimal de coups nécessaire à la « résolution » d'une configuration initiale donnée).

a. Les configurations ont un étiquetage naturel par les permutations de $[1 \dots 54]$.

Dans le pseudo-code de ce chapitre, on supposera seulement que l'on dispose, pour un graphe G :

- d'un moyen d'itérer sur les voisins (ou successeurs) d'un sommet x, en un temps proportionnel au degré (sortant) ;
- d'un moyen d'itérer sur l'ensemble des sommets de G, en un temps proportionnel au nombre de sommets ;
- d'un moyen de représenter un ensemble de n sommets de G en espace $\mathcal{O}(n)$, de manière à disposer d'un test d'appartenance efficace ($\mathcal{O}(1)$, ou à la limite $\mathcal{O}(\log n)$).

2 Parcours d'un graphe

Parcourir un graphe est une tâche assez similaire au parcours d'un arbre, avec quelques différences importantes :

- il n'y a pas de « racine » : en règle générale, on parcourt à partir d'un sommet x et l'on ne parcourra bien sûr que les sommets accessibles à partir de x ;
- le plus important : *a priori*, il y a des cycles, et il ne faut pas tourner en rond ! D'une manière ou d'une autre, il faut donc se souvenir des sommets que l'on a déjà visités[1].

2.1 Parcours en profondeur

Essentiellement, on adapte le parcours en profondeur d'un arbre en rajoutant un test pour détecter les nœuds déjà visités. On maintient donc un ensemble *vus* contenant les nœuds que l'on a déjà vus.

Algorithme 1 Parcours en profondeur (*Depth First Search*).

```
1:  fonction DFS(G, v)
2:      vus ← ∅
3:      fonction Explorer(x)
4:          si x ∉ vus alors
5:              vus ← vus ∪ {x}
6:              pré-traitement(x)
7:              pour y ∈ successeurs(x) faire
8:                  Explorer(y)
9:              post-traitement(x)
10:     Explorer(v)
```

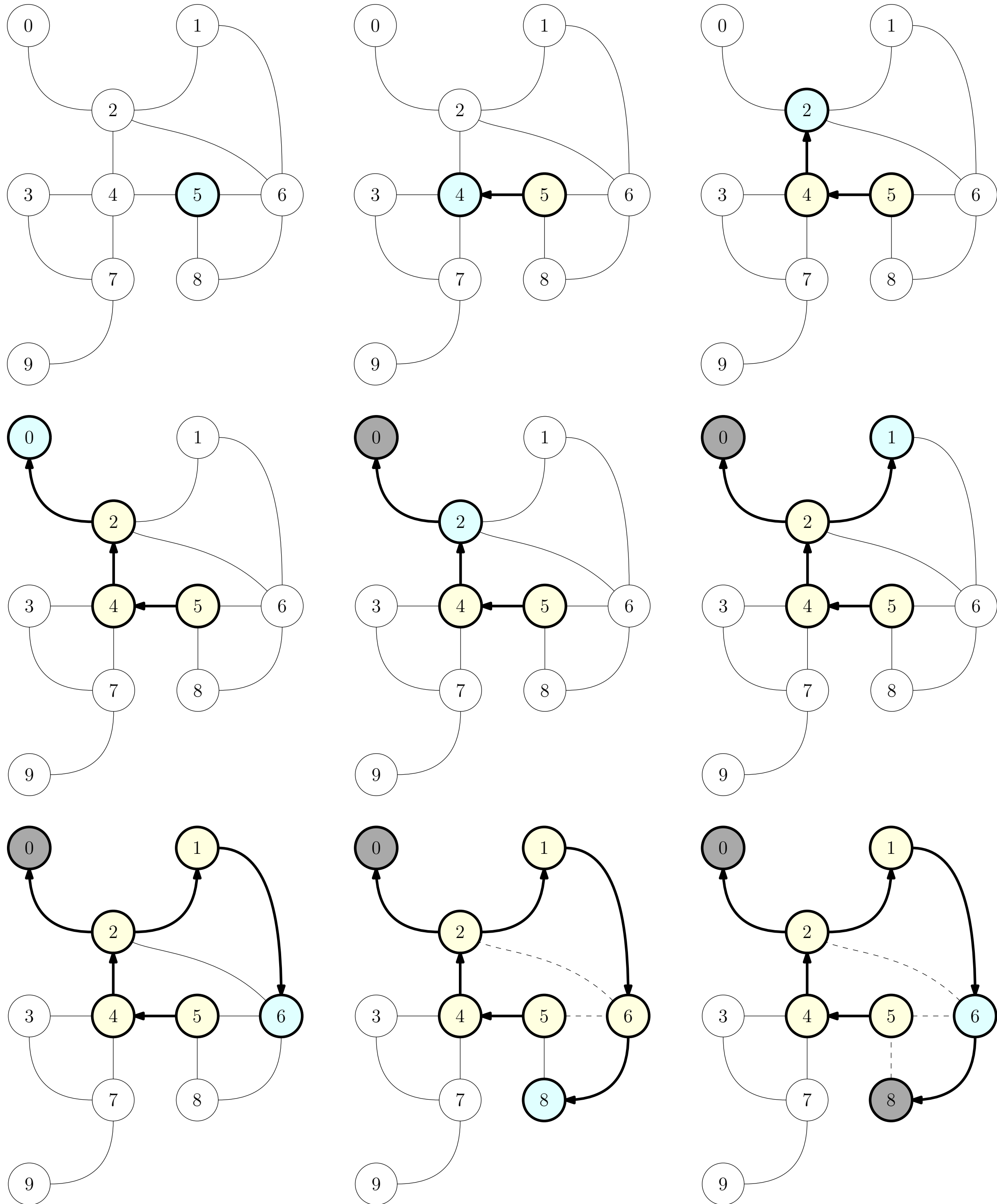
```
1:  fonction DFS-complet(G)
2:      vus ← ∅
3:      fonction Explorer(x)
4:          si x ∉ vus alors
5:              vus ← vus ∪ {x}
6:              pré-traitement(x)
7:              pour y ∈ successeurs(x) faire
8:                  Explorer(y)
9:              post-traitement(x)
10:     pour v ∈ V faire
11:         Explorer(v)
```

Remarques

- Dans la version DFS-complet, il est indispensable que tous les appels à Explorer partagent le même ensemble vus.
- Il est possible de remplacer le test « si $x \notin$ vus » par un test « si $y \notin$ vus » avant l'appel Explorer(y). Attention dans ce cas à modifier en conséquence la boucle principale de la fonction DFS-complet.
- Si le graphe est non orienté, on parlera des voisins de x plutôt que des successeurs de x.

1. Si le graphe est trop gros, ce n'est pas forcément possible mais nous ignorerons ce problème pour l'instant.

- Assez souvent, on souhaite pouvoir arrêter le parcours avant d'avoir exploré tous les nœuds accessibles (dès qu'on atteint un certain nœud, par exemple). Il faudra alors légèrement modifier l'algorithme.
- L'ensemble `vus` peut être réalisé de plusieurs manières (arbre binaire de recherche, table de hachage...). Le plus simple est d'utiliser un tableau de n booléens (où $n = |V|$) initialisés à `false`[2].



2. En plus d'être simple, cette solution est efficace, *sauf si l'on ne compte explorer qu'une petite partie du graphe.*

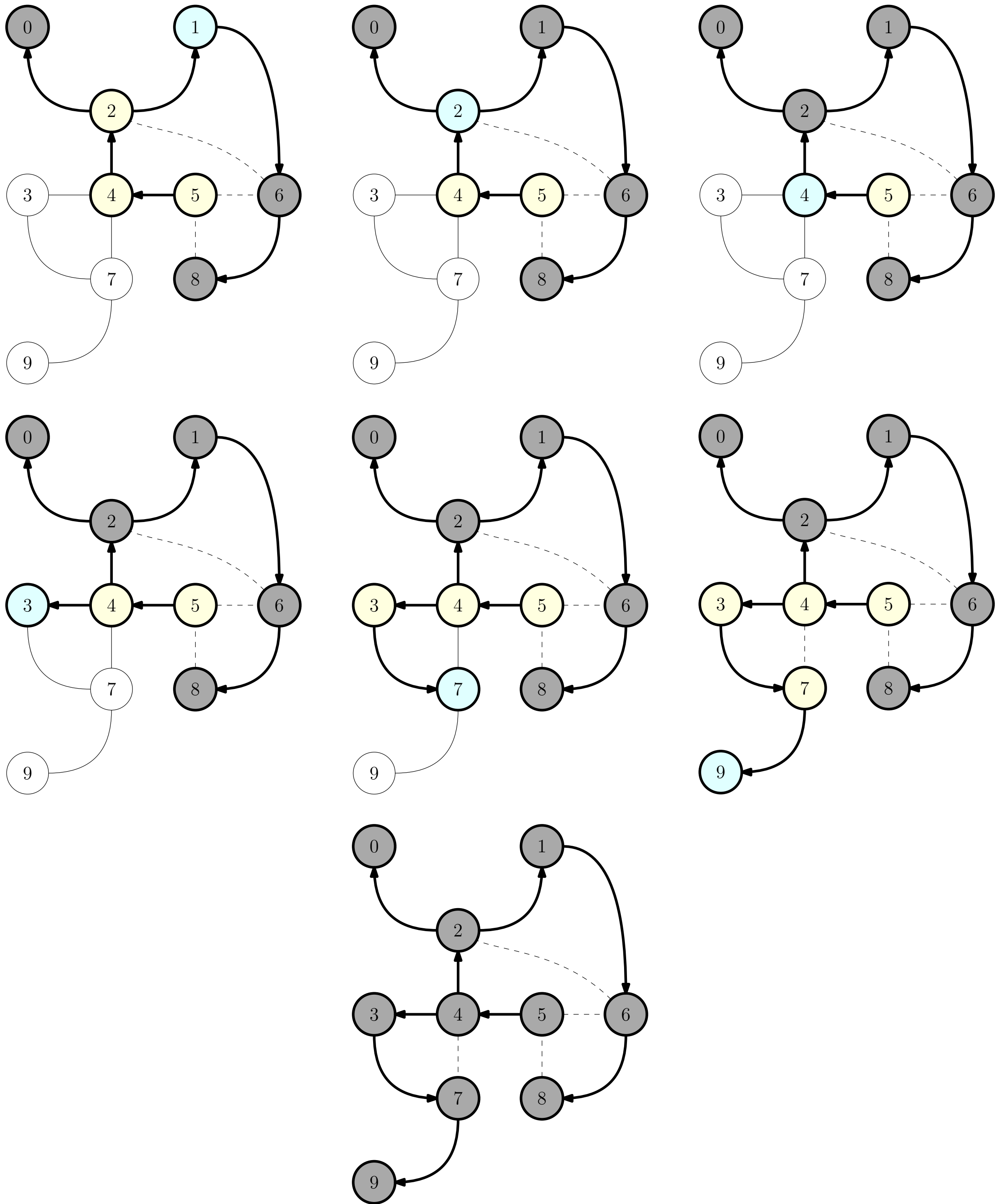


Figure 5.2 – Parcours en profondeur d'un graphe connexe non orienté.

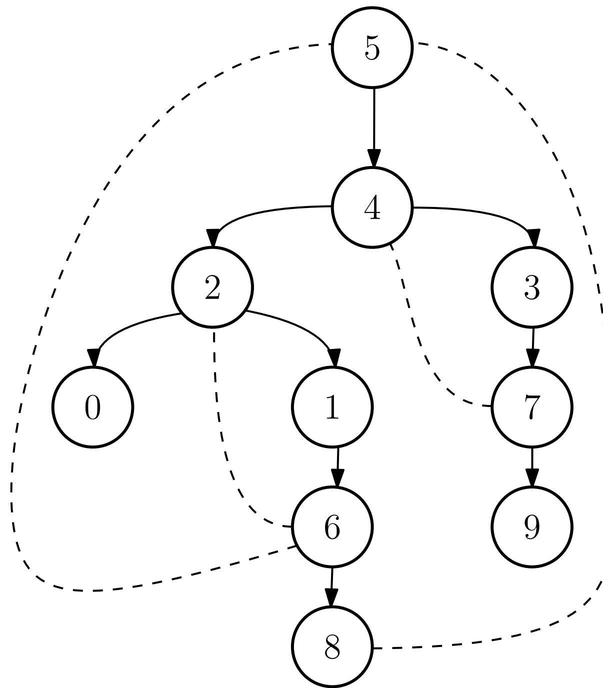


FIGURE 5.3 – Arbre associé au parcours en profondeur. Les arêtes en pointillé sont les arêtes du graphe initial qui ne sont pas conservées dans l'arbre.

Théorème 5.2 — Analyse du parcours en profondeur

On suppose dans cette analyse que l'ensemble `vus` est réalisé par un tableau de booléens et que l'on peut itérer sur les successeurs de x en temps proportionnel au nombre de successeurs

- La fonction DFS appelée sur un graphe fini G et un sommet v termine après avoir visité exactement les nœuds de G accessibles depuis v. Les fonctions *pré-traitement* et *post-traitement* sont appelées exactement une fois sur chacun de ces nœuds.
- La fonction DFS-COMPLET termine après avoir visité tous les nœuds de G. Les fonctions *pré-traitement* et *post-traitement* sont appelées exactement une fois par nœud.
- La complexité temporelle de DFS-COMPLET est $\Theta(|V| + |E|)$.
- La complexité spatiale de DFS-COMPLET est $\Theta(|V|)$ sur le tas et $\mathcal{O}(|V|)$ sur la pile.

Remarques

- Pour le cas non orienté, l'ensemble des nœuds visités est exactement la composante connexe de u.
- L'utilisation d'un espace en $\mathcal{O}(|V|)$ sur la pile peut être problématique si $|V|$ est grand. Une version itérative (ou récursive terminale) efficace est présentée à l'exercice 5.7, et d'autres variantes moins satisfaisantes dans le TP associé à ce chapitre.

Démonstration

- L'ensemble `vus` et le test associé assure qu'un sommet sera traité au plus une fois.
- Si x est visité, alors la pile d'appels `explore v -> ... -> explore x` fournit un chemin (élémentaire) de v à x.
- Inversement, supposons qu'il existe un chemin $v = x_1, \ldots, x_n = x$ reliant v à x mais que x ne soit jamais visité. Soit alors i le premier indice tel que x_i ne soit pas visité. On a forcément $i > 1$ car v est visité, et donc x_{i-1} a été visité. C'est absurde : lors de cette visite, x_i (successeur de x_{i-1}) n'était pas dans `vus` et aurait donc dû être exploré.
- On suppose ici que `vus` est un tableau de $|V|$ booléens. Son initialisation (ligne 2) prend donc un temps $\Theta(|V|)$ et les tests d'appartenance se font en temps $\Theta(1)$. Quand on appelle DFS-COMPLET, la fonction

EXPLORE est appelée exactement une fois sur chaque sommet. Or un appel à EXPLORE(x) prend un temps $\Theta(1 + |succ(x)|)$ (sans tenir compte des appels récursifs). Au total, la complexité temporelle est donc :

$$\Theta(|V|) + \sum_{x \in V} \Theta(1 + |succ(x)|) = \Theta(|V|) + \Theta(|V|) + \Theta(|E|) = \Theta(|V| + |E|)$$

- Pour la complexité spatiale, il y a deux choses à considérer :
  - — le tableau vus prend un espace $\Theta(|V|)$ (sur le tas) ;
  - — l'espace sur la pile est proportionnel à la longueur maximale des chemins explorés. Ces chemins étant élémentaires, on peut la majorer par $|V|$: on obtient donc une consommation mémoire en $\mathcal{O}(|V|)$.

■

Exercice 5.3

Donner trois famille de graphes E_n, T_n, P_n à n sommets pour lesquels le facteur limitant dans l'exécution de DFS-COMPLET sera respectivement :

1. l'espace ;
2. le temps ;
3. l'espace sur la pile.

Théorème 5.3 – Arbre de parcours en profondeur d'un graphe non orienté

Soit $G = (V, E)$ un graphe non orienté et $u \in V$. À un parcours en profondeur de G à partir de u, on peut associer un graphe orienté $T = (V', E')$ défini par :

- V' est l'ensemble des sommets visités par le parcours (*i.e* V' est la composante connexe de u) ;
- $(x, y) \in E'$ ssi le sommet y a été exploré à partir du sommet x.

On a alors :

- T est un arbre enraciné en x ;
- si xy est une arête de G qui n'apparaît pas dans T, alors x est un ancêtre de y dans T (ou inversement).

T est appelé *arbre de parcours en profondeur à partir de* x.

Remarques

- Ce théorème est hors-programme, mais il est quand même bon de l'avoir en tête quand il faut analyser de manière un peu fine les propriétés d'un parcours en profondeur. Les exercices 5.6, 5.12 et 5.13 utilisent les idées présentées ici.
- Si l'on change l'ordre d'exploration des successeurs (qui n'est pas fixé par l'algorithme DFS), on change complètement l'arbre obtenu (hauteur, arité des nœuds internes, nombre de feuilles...). La propriété énoncée dans le théorème n'est bien sûr pas affectée.
- La situation est plus complexe pour un graphe orienté : se référer à l'exercice 5.9.

2.2 Parcours en largeur

Définition 5.4 – Distance dans un graphe non pondéré

Soit G un graphe, éventuellement orienté mais non pondéré. La distance $d(x, y)$ d'un sommet x à un sommet y est la longueur minimale, en nombre d'arêtes, d'un chemin reliant x à y.
Si un tel chemin n'existe pas, $d(x, y) = \infty$.

Remarques

- Comme vu au chapitre précédent, s'il existe un chemin de x à y, alors il existe un chemin élémentaire de x à y, et de plus un plus court chemin est nécessairement élémentaire. Par conséquent, la définition précise choisie pour la notion de chemin n'influe pas sur la définition de la distance.

- Si G est connexe et non orienté, alors d est bien une distance au sens mathématique usuel.
- Si G est non orienté et non connexe, d est essentiellement une distance, mais prend ses valeurs dans $\mathbb{R}_+ \cup \{\infty\}$.
- Si G est orienté, d n'est plus symétrique : ce n'est donc pas une distance. En revanche, l'inégalité triangulaire est toujours vérifiée (et on a bien sûr $d(x, y) \geqslant 0$ avec égalité ssi $x = y$).

Propriété 5.5

S'il existe un arc yy', alors pour tout sommet x on a $d(x, y') \leqslant d(x, y) + 1$.

Le *parcours en largeur* d'un graphe à partir d'un sommet v permet de visiter les sommets par distance croissante à v :

Algorithme 2 Parcours en largeur (*Breadth-First Search*) à l'aide d'une file.

```
1:  fonction BFS(G, x0)
2:      ouverts ← file_vide()
3:      PUSH(x0, ouverts)
4:      vus ← {x0}
5:      tant que ouverts n'est pas vide faire
6:          x ← POP(ouverts)                      ▷ On extrait l'élément de tête
7:          TRAITEMENT(x)
8:          pour y ∈ G.successeurs(x) faire
9:              si y ∉ vus alors
10:                 PUSH(y, ouverts)              ▷ On ajoute y en queue.
11:                 vus ← vus ∪{y}
```

Remarques

- Le fait que *ouverts* soit une *file* (structure FIFO) est crucial !
- On pourrait bien sûr définir une fonction BFS-COMPLET comme plus haut, mais elle serait assez peu utile : le parcours en largeur n'est en règle général intéressant qu'à partir d'un certain nœud distingué.
- L'appel à TRAITEMENT pourrait être fait au moment où l'on pousse le nœud sur la file sans impacter la propriété fondamentale (les nœuds sont traités par distance croissante à x_0).

Théorème 5.6 – Propriété fondamentale du parcours en largeur

Un appel à BFS(G, x_0), où x_0 est un sommet du graphe fini G, termine après avoir visité tous les sommets accessibles depuis x_0 une et une seule fois. Les visites de ces sommets se font par distance croissante à x_0.
Ainsi, si $d(x_0, x) < d(x_0, y) < \infty$, alors TRAITEMENT$(x)$ sera exécuté avant TRAITEMENT(y).

Démonstration
Quelques observations pour commencer :

- un sommet est ajouté au plus une fois à la file (G étant fini, cela garantit la terminaison), et tout sommet ajouté finit par être traité ;
- on adapte facilement la preuve faite pour le parcours en profondeur pour montrer que les sommets visités sont exactement ceux accessibles depuis x_0 ;
- l'ordre de traitement des sommets est le même que l'ordre dans lequel ils ajoutés à la file.

À un instant donné, un sommet sera dit :

- *ouvert* s'il appartient à *ouverts*[3] ;
- *fermé* s'il appartient à vus mais pas à *ouverts* ;
- *vierge* sinon (il n'y a que trois cas car *ouverts* $\subset$ vus).

On numérote les sommets dans l'ordre de leur ouverture, $x_0, \ldots, x_n$, et l'on note $d_k := d(x_0, x_k)$. L'invariant, valable à la fermeture de x_k, est le suivant :

3. Wow !

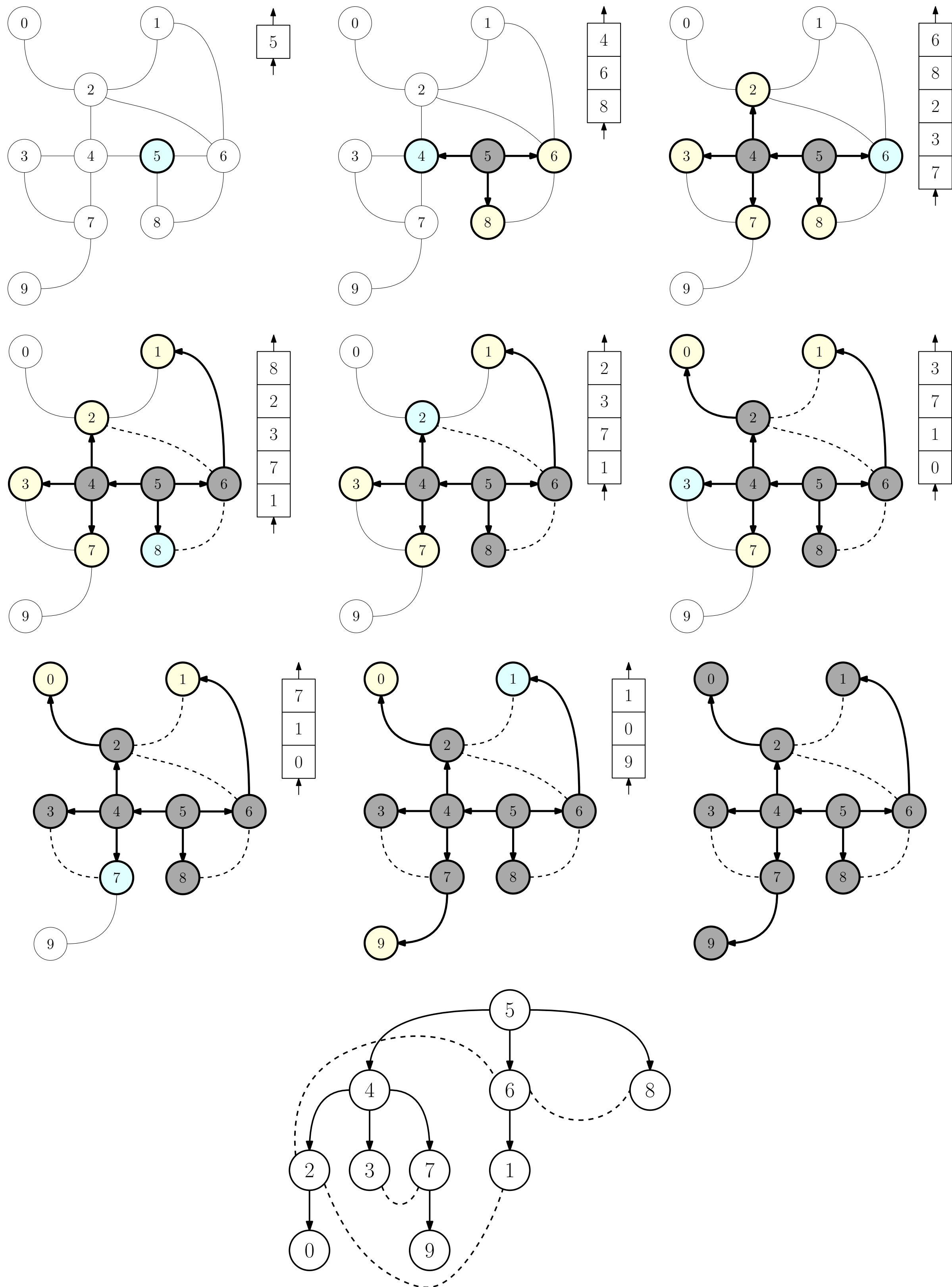


Figure 5.4 – Parcours en largeur d'un graphe non orienté.

(1) la file a la forme $x_{k+1}, \ldots, x_{k+p}$ avec $d_k \leqslant d_{k+1} \leqslant \cdots \leqslant d_{k+p} \leqslant d_k + 1$ (un nombre éventuellement nul de sommets à distance d_k suivis d'un nombre éventuellement nul de sommets à distance $d_k + 1$) ;

(2) si $d_i < d_k$, alors x_i est fermé (*i.e* $i < k$) ;

(3) si $d_i = d_k$, alors x_i n'est pas vierge.

On passe à la preuve :

Initialisation Quand on ferme x_0, la file est vide ; de plus, aucun i ne vérifie $d_i < d_0$ et seul $i = 0$ vérifie $d_i = d_0$, donc l'invariant est vérifié.

Invariance On suppose l'invariant vérifié à l'étape $k < n$, on ferme x_k et l'on ajoute ses successeurs vierges $y_1, \ldots, y_l$ sur la file. Comme les y_i sont vierges, on a d'après l'invariant $d(x_0, y_i) > d_k$. Comme de plus $d(x_0, y_i) \leqslant d_k + 1$ d'après la proposition 5.5, on a en fait $d(x_0, y_i) = d_k + 1$: la forme de la file est préservée.
On distingue maintenant deux cas :

- si $d_{k+1} = d_k$, il n'y a rien de plus à prouver ;
- si $d_{k+1} = d_k + 1$, alors tous les sommets à distance d_k sont fermés, puisqu'il ne peut en rester sur la file et que l'invariant garantit qu'aucun n'est vierge. Cela montre le point (2) de l'invariant. De plus, comme tous ces sommets sont fermés, aucun de leurs successeurs ne peut être vierge ; comme tout sommet à distance $d_k + 1$ est successeur d'un sommet à distance d_k, cela montre le point (3) de l'invariant.

Conclusion L'invariant est donc vérifié pour tout k, et son point (2) garantit donc que si $d_i < d_k$, alors $i < k$: c'est la conclusion cherchée.

■

Théorème 5.7 — Complexité du parcours en largeur

Le parcours en larguer a une complexité spatiale en $\Theta(|V|)$. En supposant que les opérations élémentaires sur les files et les ensembles se font en temps constant, sa complexité temporelle est en $\mathcal{O}(|E| + |V|)$.

Démonstration
Si l'on réalise `vus` par un tableau de $|V|$ booléens, l'initialisation se fait en temps $\Theta(|V|)$. Ensuite, le traitement de chaque nœud prend un temps proportionnel à son nombre de successeurs (plus une constante). Ainsi, le temps total est proportionnel à la taille (nombre d'arêtes plus nombre de sommets) du sous-graphe accessible depuis le sommet initial. Cette taille est bien en $\mathcal{O}(|E|+|V|)$, donc la complexité temporelle totale est en $\mathcal{O}(|E| + |V|)$.
Pour l'espace, on consomme $\Theta(|V|)$ pour `vus` et $\mathcal{O}(|V|)$ sur la file (puisque tous les sommets présents sur la file sont distincts). Au total, la complexité spatiale est donc en $\mathcal{O}(|V|)$. ■

Exercices Supplémentaires

Dans tous les exercices qui suivent, on supposera que l'on dispose d'un type `graphe` défini ainsi :

```
(* Uniquement pour clarifier les signatures *)
type sommet = int

type graphe =
 {nb_sommets : int;
  voisins : sommet -> sommet list;
  adjacents : sommet -> sommet -> bool}
```

On supposera de plus que la fonction `voisins` s'exécute en temps constant, ce qui est le cas si le graphe est stocké sous forme d'un tableau de listes d'adjacence :

```
(* of_listes : sommet list array -> graphe *)
let of_listes t =
  let n = Array.length t in
  let v i = t.(i) in
  let adj i j = List.mem j (voisins i) in
  {nb_sommets = n; adjacents = adj; voisins = v}

(* Si g est défini par let g = of_listes t pour un certain t, alors g.voisins
   est bien en O(1) (mais pas g.adjacents). *)
```

Exercice 5.4 — Graphes 2-coloriables

Une k-*coloration* d'un graphe non orienté $G = (V, E)$ est une application $\varphi : V \to [0 \dots k-1]$ telle que $xy \in E \Rightarrow \varphi(x) \neq \varphi(y)$. Un graphe est dit k-*coloriable* s'il admet une k-coloration. Déterminer si un graphe est k-coloriable est difficile (problème NP-complet) dès que $k \geqslant 3$. En revanche, le problème est très simple pour $k = 2$.

1. Montrer que si $G = (V, E)$ est connexe et $x_0 \in V$, alors il existe au plus une 2-coloration φ de G tel que $\varphi(x_0) = 0$.
2. Écrire une fonction `deux_coloration : graphe -> int array option` qui renvoie `Some t`, où `t` code une 2-coloration du graphe passé en argument, s'il en existe une, `None` sinon. On exige une complexité *linéaire* en la taille $|E| + |V|$ du graphe.
   Indication : on pourra initialiser `t` à -1 et utiliser la question précédente.

Exercice 5.5 — Test de forte connexité

On considère un graphe orienté $G = (V, E)$, et l'on souhaite déterminer de manière efficace s'il est fortement connexe.

1. Proposer un algorithme naïf pour répondre à la question, et déterminer sa complexité.
2. On note $G^{\leftarrow}(V, E^{\leftarrow})$, le graphe obtenu en changeant le sens de toutes les arêtes de G ; autrement dit, on pose $E^{\leftarrow} := \{(v, u) | (u, v) \in E\}$. Proposer un algorithme répondant au problème posé à l'aide d'un parcours de G et d'un parcours de $G^{\leftarrow}$. On justifiera soigneusement sa correction.
3. Déterminer la complexité de l'algorithme.
4. Écrire la fonction `est_fortement_connexe : graphe -> bool`.

Remarque
Le problème de la détermination des composantes fortement connexes est plus délicat mais peut également se résoudre en temps linéaire en la taille du graphe. Voir par exemple l'exercice 5.11.

Exercice 5.6 — Théorème de Robbins et détection des ponts

Un graphe non orienté est dit *fortement orientable* s'il existe une orientation de ses arêtes qui le rende fortement connexe. D'autre part, un graphe non orienté connexe est dit 2-arête connexe, ou *sans pont*, s'il n'existe pas d'arête dont la suppression déconnecterait le graphe (une telle arête est appelée *pont*).

1. Montrer qu'une arête est un pont si et seulement si elle ne fait partie d'aucun cycle élémentaire.
2. On considère un graphe connexe G et T un arbre de parcours en profondeur pour G à partir d'un sommet (quelconque) r. On considère l'orientation suivante de G :
   - les arêtes de T sont orientées de la racine vers les feuilles ;
   - les autres arêtes, qui relient forcément un nœud et l'un de ses ancêtres dans T (*cf.* théorème **??**), sont orientées des feuilles vers la racine.

   Montrer que si G est sans pont, cette orientation rend G fortement connexe.
3. En déduire le *théorème de Robbins (1939)* : un graphe est fortement orientable si et seulement si il est sans pont.

   Remarque

   Autrement dit, il est possible de mettre toutes les rues d'une ville à sens unique (en gardant la possibilité d'aller de n'importe quel point A à n'importe quel point B) si et seulement si on ne peut séparer la ville en deux parties qui ne sont reliées que par une seule rue.
4. On définit :

```
(* Soit un graphe, soit une arête *)
type t = G of graphe | A of int * int
```

   Écrire une fonction `orientation_forte : graphe -> t` qui prend en entrée un graphe supposé non orienté, et renvoie :
   - `G g'`, où `g'` est une orientation forte de `g` s'il en existe une ;
   - `A (i, j)`, où ij est un pont dans `g`, sinon.

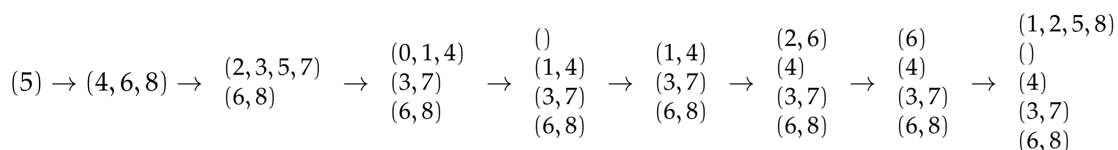
   On demande une complexité linéaire en la taille du graphe (c'est-à-dire en $\mathcal{O}(|V| + |E|)$).

Exercice 5.7 — Parcours en profondeur itératif efficace

On a vu au TD **??** exercice **??** que la manière la plus simple d'écrire un « vrai » parcours en profondeur (vérifiant le théorème **??**) itératif avait une complexité spatiale en $\mathcal{O}(|E|)$. Écrire une fonction purement itérative (ou récursive terminale) réalisant un parcours en profondeur, en utilisant l'idée suivante :

- la pile ne contient pas des sommets, mais des *listes de sommets* ;
- un élément de la pile correspond précisément à une *stack frame* du parcours en profondeur récursif : la liste contient les appels récursifs qu'il reste à faire.

En parcourant à partir du sommet 5 dans le graphe de la figure 5.2, les premières étapes d'évolution de la pile doivent être :



On justifiera bien que, si le graphe est stocké sous forme de listes d'adjacence, la complexité spatiale est en $\mathcal{O}(|V|)$ (un schéma mémoire peut être utile).

Exercice 5.8 – Rendu de monnaie

On *système monétaire* S est une suite finie $1 = s_1 < \cdots < s_k$ d'entiers représentant des dénominations de pièces ou billets (en centimes d'euros, par exemple). Le *problème du rendu de monnaie* est d'obtenir un montant m donné en utilisant le moins de pièces possibles. Par exemple, si $S = (1, 2, 5)$, alors la décomposition optimale de 12 est $5 + 5 + 2$, qui utilise 3 pièces.

1. Proposer un algorithme glouton pour ce problème. Exhiber un S et un m pour lesquels cet algorithme ne renvoie pas une solution optimale.
2. Montrer que ce problème se ramène à la recherche d'un plus court chemin dans un certain graphe. Proposer
3. Programmer une fonction exploitant cette idée. On pourra utiliser le module `Hashtbl` de la bibliothèque standard de Caml.
4. Comparer cette fonction avec ce qu'on aurait obtenu si l'on avait utilisé une solution récursive mémoïsée.

Problèmes

1 Forêt de parcours en profondeur pour un graphe orienté et applications

On étend l'algorithme de parcours en profondeur de manière à obtenir :

- pour chaque nœud u, le temps de début de traitement **pre**(u) et de fin de traitement **post**(u) (ces temps sont numérotés séquentiellement à partir de 1) ;
- pour chaque arc, une classification en tant qu'arc *arbre*, *avant*, *arrière* ou *latéral*.

Algorithme 3 Parcours en profondeur avec classification des arcs

Entrées : Un graphe orienté $G = (V, E)$
Sorties : Deux tableaux **pre** et **post** indexés par V, une partition des arcs de G

```
actifs ← ∅
pre ← [0, ..., 0]
post ← [0, ..., 0]
t ← 0
fonction EXPLORER(u)
    t ← t + 1
    pre[u] ← t
    actifs ← actifs ∪ {u}
    pour v ∈ successeurs(u) faire
        si pre[v] = 0 alors
            (u, v) est un arc arbre
            EXPLORER(v)
        sinon si pre(v) > pre(u) alors
            (u, v) est un arc avant
        sinon si v ∉ actifs alors
            (u, v) est un arc latéral
        sinon
            (u, v) est un arc arrière
    t ← t + 1
    post[u] ← t
    actifs ← actifs \ {u}
pour v ∈ V faire                                  ▷ Boucle principale
    EXPLORER(v)
```

Figure 5.5 – Le graphe G_0

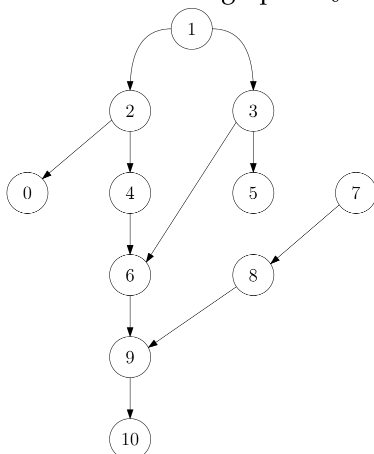
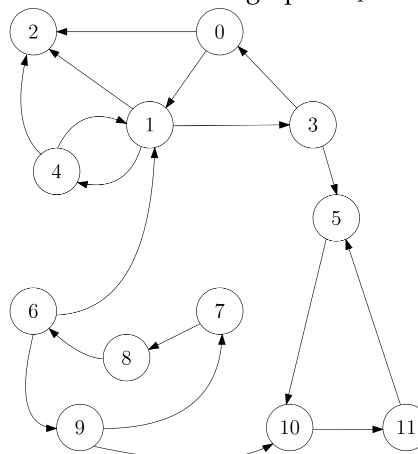


Figure 5.6 – Le graphe G_1.



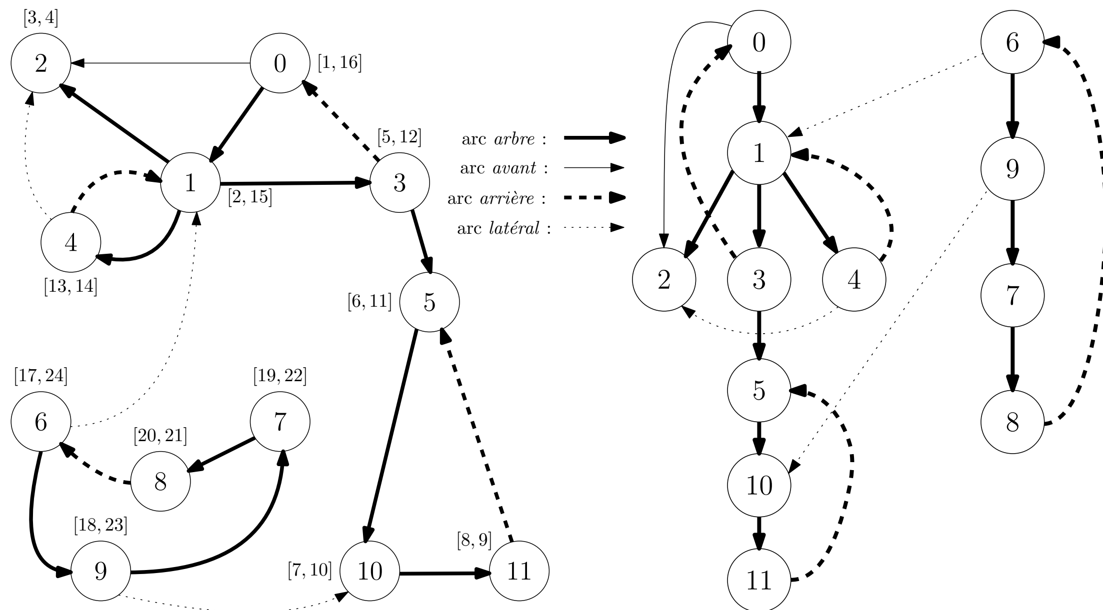


Figure 5.7 – Exemple de classification des arcs de G_1, avec la forêt correspondante. Pour chaque nœud v, on a indiqué l'intervalle $[\mathbf{pre}(v), \mathbf{post}(v)]$.

Exercice 5.9

1. Les temps **pre** et **post** et la classification des arcs obtenus à la figure 5.7 correspondent à un traitement des nœuds dans l'ordre croissant des étiquettes, autant dans la boucle principale que pour chaque parcours d'une liste de successeurs. Simuler l'exécution de l'algorithme et faire la figure correspondante si le traitement se fait dans l'ordre décroissant des étiquettes.
2. Justifier que les arcs *arbre* définissent une forêt F (dont l'ensemble des sommets est V).
3. Pour le graphe G_1, exhiber un ordre d'énumération des sommets dans la boucle principale donnant une forêt réduite à un arbre et un ordre donnant une forêt constituée de cinq arbres.
4. Classer $\mathbf{pre}(u), \mathbf{pre}(v), \mathbf{post}(u), \mathbf{post}(v)$ par ordre croissant suivant le type de l'arc (u, v) et justifier que, dans la forêt F :
   - si (u, v) est un arc *avant*, alors v est un descendant de u ;
   - si c'est un arc *arrière*, alors u est un descendant de v ;
   - si c'est un arc *latéral*, alors les (sous-)arbres enracinés en u et en v sont disjoints.

   Quels sont les ordres impossibles pour les temps de traitement (respectant $\mathbf{pre}(u) < \mathbf{post}(u)$ et $\mathbf{pre}(v) < \mathbf{post}(v)$) ?
5. Si G est un graphe non orienté (c'est-à-dire si $(u, v) \in E \Rightarrow (v, u) \in E$, quels types d'arcs restent possibles ? Que peut-on dire de la forêt obtenue ?

Exercice 5.10 – Test d'acyclicité, tri topologique

On rappelle (chapitre **??**, section **??**) qu'un *tri topologique* d'un graphe orienté $G = (V, E)$ est un ordre total $x_1 \prec \cdots \prec x_n$ sur V tel que $(x, y) \in E \Rightarrow x \prec y$. On rappelle également que G admet un tri topologique si et seulement si G est acyclique.

1. Appliquer l'algorithme 3 au graphe G_0, en traitant les sommets par étiquettes croissantes.
2. Montrer qu'un graphe est acyclique si et seulement si son parcours en profondeur ne produit pas d'arc *arrière*.
3. En déduire un algorithme calculant un tri topologique d'un graphe orienté, ou signalant l'impossibilité d'un tel tri.
4. Écrire une fonction Caml `tri_topologique : graphe -> sommet list option` renvoyant

un tri topologique `Some [x_1; ... ; x_n]` si l'argument est un DAG, `None` sinon. On exige une complexité linéaire (en la taille $|E|+|V|$ du graphe), et l'on précise que la fonction ne devrait pas faire plus d'une vingtaine de lignes.
Indication : il est inutile de calculer **pre** et **post**, il suffit de définir

```
type marque = Vierge | Ouvert | Ferme
```

Exercice 5.11 – Algorithme de Kosaraju pour les composantes fortement connexes

On considère un graphe orienté $G = (V, E)$ et l'on note $C_1, \dots, C_k$ ses composantes fortement connexes. Le but de cet exercice est d'obtenir une fonction Caml `cfc : graphe -> sommet list list` calculant une partition des sommets du graphe de départ en composantes fortement connexes en temps linéaire (en $|E|+|V|$), et de prouver sa correction. On rappelle une définition et deux résultats (très simples) vus au chapitre **??**, section **??**.

- Le graphe $G_{cfc} = (V_{cfc}, E_{cfc})$ des composantes fortement connexes de G est défini par :
  — $V_{cfc} = \{C_1, \dots, C_k\}$ l'ensemble des composantes fortement connexes de G ;
  — l'arc (C_i, C_j) est dans E_{cfc} si et seulement si on peut trouver $x \in C_i$ et $y \in C_j$ tel que $(x, y) \in E$.
- Le graphe G_{cfc} est un DAG.
- Tout DAG[a] possède au moins une racine (degré entrant nul) et une feuille (degré sortant nul).

On notera parfois $C(x)$ la composante fortement connexe d'un sommet x.

1. Déterminer les composantes fortement connexes du graphe G_1 de la figure 5.6 et représenter le graphe G_{1cfc}.
2. **Principe de l'algorithme** Soit C_i une feuille de G_{cfc}. Montrer qu'un parcours en profondeur de G à partir d'un sommet $x \in C_i$ explore exactement C_i.

   On en déduit immédiatement la correction de l'algorithme suivant :

   Composantes $\leftarrow \emptyset$
   tant que G est non vide **faire**
       Trouver un sommet x tel que $C(x)$ soit une feuille de G_{cfc}.
       $C_x \leftarrow \emptyset$
       pour v rencontré lors d'un appel DFS(G, x) **faire**
           $C_x \leftarrow C_x \cup \{v\}$
           Supprimer v dans G
       Composantes $\leftarrow$ **Composantes** $\cup \{C_x\}$
   renvoyer Composantes

   Il reste deux problèmes à résoudre :
   - comment trouver un sommet x tel que $C(x)$ soit une feuille (en temps linéaire) ?
   - comment obtenir une complexité linéaire pour la totalité de l'algorithme ?
3. **Lemme principal** On considère un parcours en profondeur de G et les nombres **post** associés, et, pour chaque composante connexe C, on note $\textbf{post}(C) = \max_{v \in C}(\textbf{post}(v))$ l'instant de fin de traitement de C.
   Montrer que si $(C, C') \in E_{cfc}$, alors $\textbf{post}(C) > \textbf{post}(C')$.
4. En déduire une méthode pour trouver un sommet x tel que $C(x)$ soit une *racine* en temps linéaire.
5. Nous voulions une feuille et nous avons obtenu une racine : justifier que $(G_{cfc})^{\leftarrow} = (G^{\leftarrow})_{cfc}$ et expliquer comment résoudre ce problème.
6. Justifier alors la correction de l'algorithme suivant, et sa complexité linéaire :

Algorithme 4 Algorithme de Kosaraju

Entrées : Un graphe orienté G
Sorties : La liste $C_1, \dots, C_k$ de ses composantes fortement connexes
1: Calculer $G^{\leftarrow}$
2: Effectuer un DFS de $G^{\leftarrow}$, récupérer la liste $x_1, \dots, x_n$ des sommets par **post** décroissant
3: `vus` $\leftarrow \emptyset$
4: $L \leftarrow \emptyset$
5: **pour** x de $x_1, \dots, x_n$ dans l'ordre **faire**
6: **si** $x \notin$ `vus` **alors**
7: $C \leftarrow \emptyset$
8: Effectuer un DFS de G à partir de x
9: Ajouter les sommets rencontrés à `vus` et à C
10: $L \leftarrow L \cup \{C\}$
11: **renvoyer** L

7. Implémenter l'algorithme en Caml (la fonction réalisant la ligne 1 a déjà été vue à l'exercice **??** du TD **??**, celle réalisant la ligne 2 devrait prendre une douzaine de lignes, celle correspondant à la boucle une petite vingtaine).

a. Non vide, pour les adeptes de la tétrapiloctomie.

2 Algorithme de Schmidt pour la recherche de points d'articulation

Dans cette partie, les termes *cycles* et *chemins* désigneront systématiquement des cycles et chemins **élémentaires**.

Exercice 5.12 – Décomposition en chaînes

On considère un graphe non orienté connexe G et un arbre de parcours en profondeur T associé, à partir d'un certain sommet r. On oriente maintenant les arêtes de G de la manière suivante :

- les arêtes appartenant à T sont orientées des feuilles vers la racine r et deviennent des *arcs arbre* ;
- les autres arêtes (dont le théorème **??** assure qu'ils relient un sommet et l'un de ses descendants) sont orientées de la racine vers les feuilles et deviennent des *arcs arrière*.

On numérote les sommets $v_1, \dots, v_n$ dans l'ordre de parcours préfixe de l'arbre (ce qui revient à les numéroter par leur instant de début de traitement dans le parcours en profondeur).

1. Justifier que chaque arc arrière e appartient à un unique cycle que l'on notera $\sigma(e)$.

2. On définit alors la *décomposition en chaînes* de G comme suit :

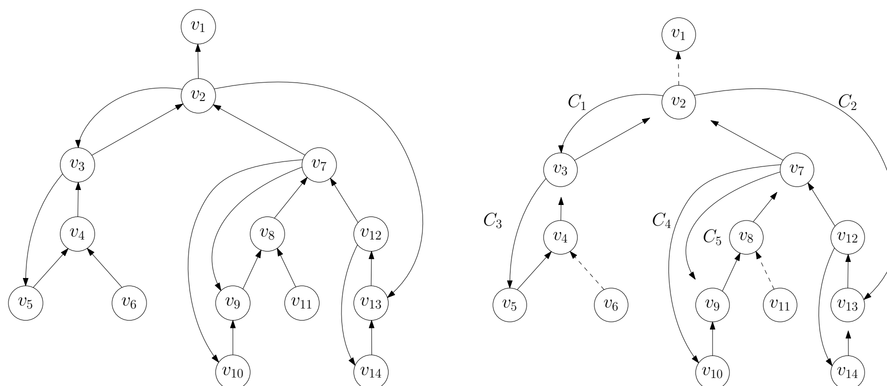


FIGURE 5.8 – Un arbre de parcours en profondeur et la décomposition en chaînes associée. Les chaînes sont $C_1 = v_2v_3v_2$, $C_2 = v_2v_{13}v_{12}v_7v_2$, $C_3 = v_3v_5v_4v_3$, $C_4 = v_7v_{10}v_9v_8v_7$ et $C_5 = v_7v_9$.

Algorithme 5 Décomposition en chaînes

Entrées : G un graphe connexe non orienté
Sorties : Une liste de chemins et de cycles (orientés) $C_1, \dots, C_k$

1: Calculer T, orienter G et obtenir $v_1, \dots, v_n$ comme ci-dessus
2: **Dec** $\leftarrow \emptyset$
3: **vus** $\leftarrow \emptyset$
4: **pour** $u \in v_1, \dots, v_n$ dans l'ordre **faire**
5:     **pour** arc arrière e commençant en u **faire**
6:         $C \leftarrow \emptyset$
7:         **vus** $\leftarrow$ **vus** $\cup \{u\}$
8:         **pour** $e' = (u, u')$ arc de C(e), en commençant par e **faire**
9:             $C \leftarrow C, e'$
10:             **si** $u' \in$ **vus alors**
11:                 **break**
12:             **sinon**
13:                 **vus** $\leftarrow$ **vus** $\cup \{u'\}$
14:         **Dec** $\leftarrow$ **Dec**, C
15: **renvoyer Dec** $= C_1, \dots, C_k$

Remarques

- La notation $A \leftarrow A, x$ signifie que l'on ajoute x à la fin de la liste A.
- L'ordre de traitement des différents arcs arrière partant d'un sommet donné n'est pas fixé. Des choix différents donneront des décompositions en chaînes différentes (réfléchir par exemple à ce qui se passe si l'on traite C_5 avant C_4 dans l'exemple ci-dessus).
- Une chaîne sera soit un cycle soit un chemin, suivant que le processus a ou non continué jusqu'à revenir au sommet u initial.

Déterminer une décomposition en chaînes du graphe G suivant (en partant de l'arbre de parcours en profondeur donné) :

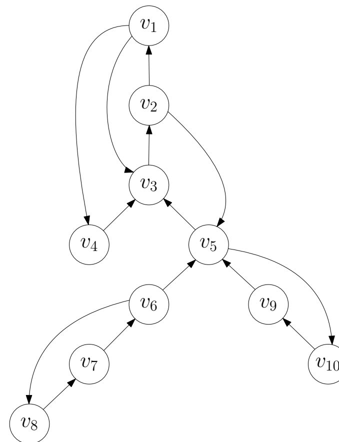


FIGURE 5.9 – Le graphe G orienté suivant un parcours en profondeur.

3. Justifier que, si C_1 existe, alors c'est un cycle. Que peut-on dire de G si C_1 n'existe pas ?

4. Écrire une fonction `decomposition_chaines : graphe -> sommet list list` réalisant l'algorithme. On demande une complexité linéaire en la taille du graphe.

Exercice 5.13 – Application à la 2-arête et 2-sommet connexité

On considère un graphe non orienté et connexe G, ayant au moins 3 sommets pour éviter des cas particuliers sans intérêt.

- Un *pont* de G est une arête dont la suppression déconnecte G. G est dit *2-arête connexe* (ou simplement *sans pont*) s'il n'a pas de pont.
- Un *point d'articulation* de G est un sommet dont la suppression déconnecte G. G est dit *2-sommet connexe*, ou simplement *biconnexe*, s'il n'a pas de point d'articulation.

On note T l'arbre de parcours en profondeur de G (c'est-à-dire le sous-graphe ne contenant que les arcs arbre), et $T(x)$ le sous-arbre de T enraciné en un sommet x. On note également $C = C_1, \dots, C_k$ une décomposition en chaîne correspondant à T, telle que renvoyée par l'algorithme 5.

1. Montrer qu'une arête e est un pont si et seulement si elle n'appartient à aucune chaîne.

2. On note $\delta(G)$ le degré minimum de G, et l'on souhaite montrer le résultat suivant : si $\delta(G) \geqslant 2$, alors un sommet v est un point d'articulation si et seulement si il est incident à un pont ou est le premier sommet d'une chaîne cyclique autre que C_1.

- **a.** Supposons que v soit le premier sommet d'une chaîne cyclique C autre que C_1.
  - **(i)** Montrer que si v est la racine r de T, alors v est un point d'articulation.
  - **(ii)** On suppose que $v \neq r$, et l'on note (w, v) le dernier arc de C. Montrer que la suppression de v déconnecte $T(w)$ de la racine r.
- **b.** Supposons que v soit un point d'articulation non incident à un pont, et soient X, Y deux composantes connexes de $G \setminus \{v\}$. On note X', Y' les sous-graphes de G induits par $X \cup \{v\}$ et $Y \cup \{v\}$.
  - **(i)** Montrer que X' et Y' contiennent chacun un cycle passant par v.
  - **(ii)** Justifier qu'un cycle de G est soit inclus dans X', soit inclus dans $G \setminus X$.
  - **(iii)** On suppose que C_1 n'est pas inclus dans X'. Montrer que X' contient une chaîne cyclique ayant v comme premier sommet.
- **c.** Conclure.

3. En déduire un algorithme renvoyant la liste des ponts et la liste des points d'articulation de G et justifier qu'il peut être implémenté de manière à avoir une complexité linéaire en la taille de G.

4. Proposer une telle implémentation en Caml.