Cours : Graphe, pratique

Table des matières

1 Représentation informatique d'un graphe

Définition 1.1: Taille d'un graphe

On appelle *taille* d'un graphe $G := (S, A)$ la quantité $|S| + |A|$.

Remarques

⇨ Cette quantité n'a aucune signification « mathématique » (on additionne des choux et des carottes…) mais est pertinente informatiquement : on dira par exemple qu'un algorithme qui s'exécute en temps $O(|S|+|A|)$ est *linéaire en la taille du graphe*.

⇨ La taille est au plus de l'ordre de $|S|^2$ (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 $[0 \dots n-1]$, 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, et éventuellement d'un dictionnaire associant à l'étiquette d'un nœud son numéro.

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 très compacte.

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

⇨ 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 $99,99999\%$ 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 100000 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.

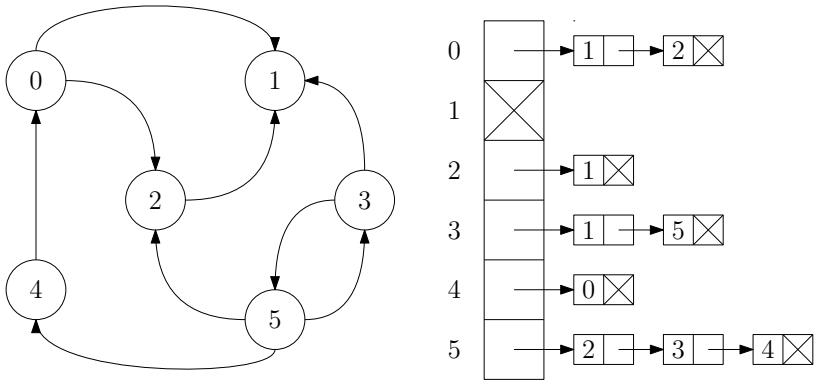
Un graphe G à n sommets sera donc stocké sous la forme d'un `int list array` de longueur n, et G sera non orienté si et seulement si $i \in t.(j) \Longrightarrow j \in t.(i)$ pour tous $i, j \in [0 \dots n]$.

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 $\Theta(|E| + |V|)$).



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

Remarques

⇨ 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.

⇨ On peut tout à fait remplacer les listes chaînées d'adjacence par des tableaux (éventuellement dynamiques si l'on souhaite pouvoir ajouter des arêtes). En C, où les listes chaînées ne sont pas très naturelles, c'est généralement ce que l'on fera.

Exercices 1

⇨ *Changement de représentation* :

1. Écrire une fonction prenant en entrée un graphe représenté sous forme d'une matrice d'adjacence et renvoyant une représentation du même graphe sous forme de tableau de listes d'adjacence. On fera en sorte que les listes d'adjacence soient triées par ordre croissant.
2. Écrire une fonction réalisant la transformation inverse.

```
listes_of_matrice : bool array array -> int list array
matrice_of_listes : int list array -> bool array array
```

```
let listes_of_matrice m =
  let n = Array.length m in
  let g = Array.make n [] in
  for i = 0 to n - 1 do
    for j = n - 1 downto 0 do
      if m.(i).(j) then
        g.(i) <- j :: g.(i)
    done
  done;
  g
```

```
let matrice_of_listes g =
  let n = Array.length g in
  let m = Array.make_matrix n n false in
  let rec ajoute x voisins =
    match voisins with
    | [] -> ()
    | y :: ys ->
      m.(x).(y) <- true;
      ajoute x ys in
  for i = 0 to n - 1 do
    ajoute i g.(i)
  done;
  m
```

□

⇨ *Utilisation d'ensembles et de dictionnaires* : On suppose que l'on dispose des structures d'ensemble et de dictionnaire qui fournissent les opérations suivantes :

- `set_new : unit -> 'a set`
- `set_add : 'a set -> 'a -> unit`
- `set_member : 'a set -> 'a -> bool`
- `set_remove : 'a set -> 'a -> unit`
- `set_iter : 'a set -> ('a -> unit) -> unit`
- `map_new : unit -> ('k, 'v) map`
- `map_set : ('k, 'v) map -> 'k -> 'v -> unit`
- `map_get : ('k, 'v) map -> 'k -> 'v option`
- `map_remove : ('k, 'v) map -> 'k -> unit`

On suppose de plus que `set_add`, `set_member`, `set_remove`, `map_set`, `map_get` et `map_remove` sont en temps constant, et que l'appel `set_iter s f` a un coût proportionnel à la somme des appels `f x` engendrés.

1. Proposer une structure de données permettant de représenter un graphe orienté de manière à fournir les opérations suivantes :
   - Test d'existence d'un arc entre deux sommets en temps constant.
   - Ajout et suppression d'un arc en temps constant
   - Ajout d'un sommet en temps constant.
   - Itération sur les successeurs d'un sommet en temps proportionnel à son degré sortant.
   - Itération sur tous les sommets du graphe en temps proportionnel à leur nombre.
2. Cette structure permet-elle de supprimer un sommet facilement ? Si ce n'est pas le cas, proposer une modification.

Solution. 1. Une solution est d'utiliser $n+1$ dictionnaires, où n est le nombres de sommets du graphe :
   - un dictionnaire pour chaque sommet, dont les clés sont les successeurs du sommet et les valeurs les étiquettes des arcs correspondants (ou juste `()`) ;
   - un dictionnaire pour le graphe, dont les clés sont les identifiants des sommets et les valeurs les dictionnaires du point précédent.

2. On peut facilement supprimer un sommet et tous les arcs qui en partent en temps constant : il suffit de l'enlever du dictionnaire principal. Le problème est que les arcs pointant vers ce sommet ne sont pas affectés, et que l'on aura une erreur plus tard si on essaie de les suivre. Le plus simple est d'ajouter un dictionnaire *predecesseurs* pour chaque sommet.

□

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

⇨ 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 (Les configurations ont un étiquetage naturel par les permutations de $[1 \dots 54]$.), de calculer les étiquettes de ses voisins. Ainsi, 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).

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 $O(n)$, de manière à disposer d'un test d'appartenance efficace ($O(1)$, ou à la limite $O(\log n)$).

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

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.

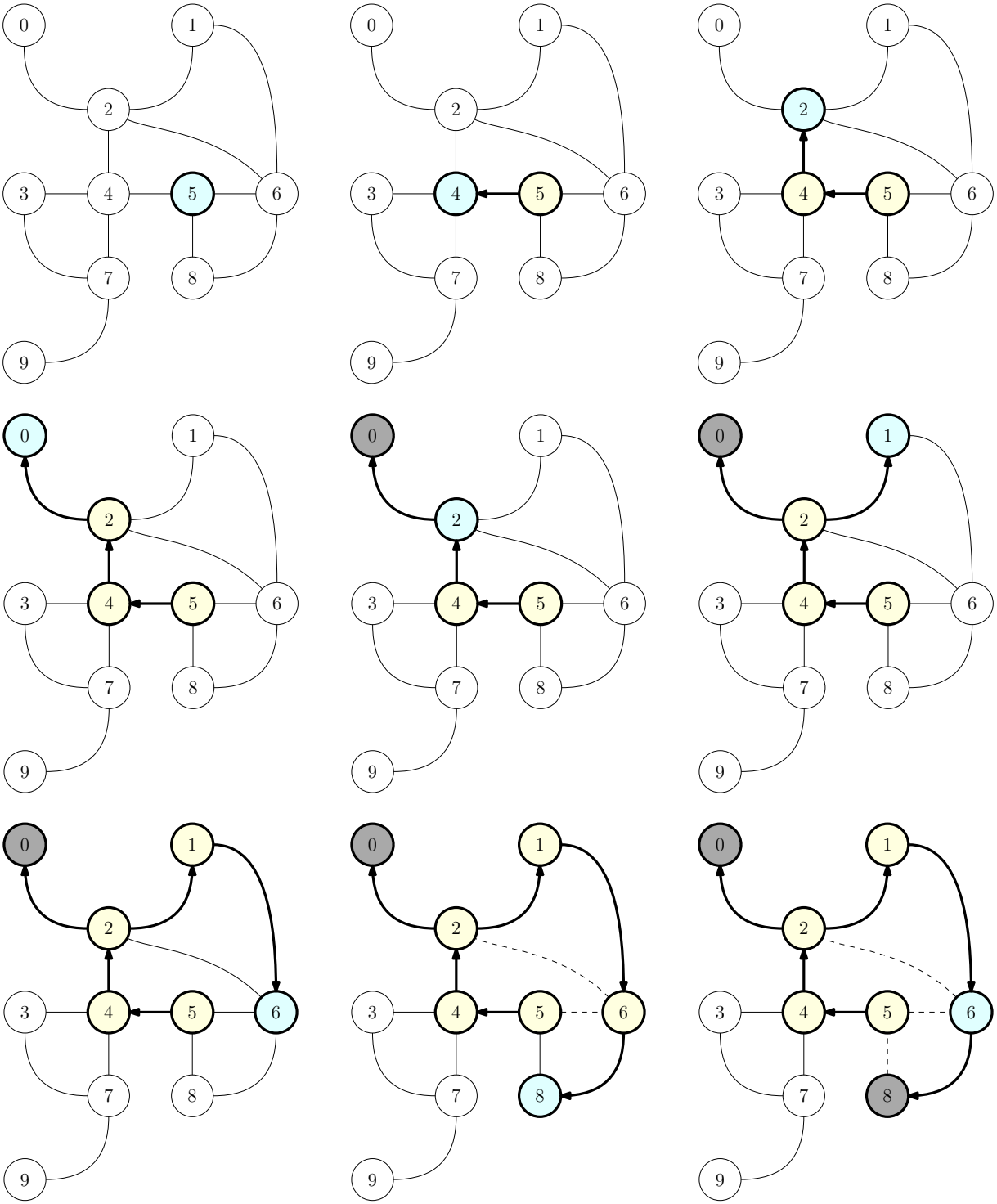
Algorithme 1 Parcours en profondeur (*Depth First Search*).

```
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:              fin pour
10:             post-traitement(x)
11:         fin si
12:     fin fonction
13:     Explorer(v)
14: fin fonction
```

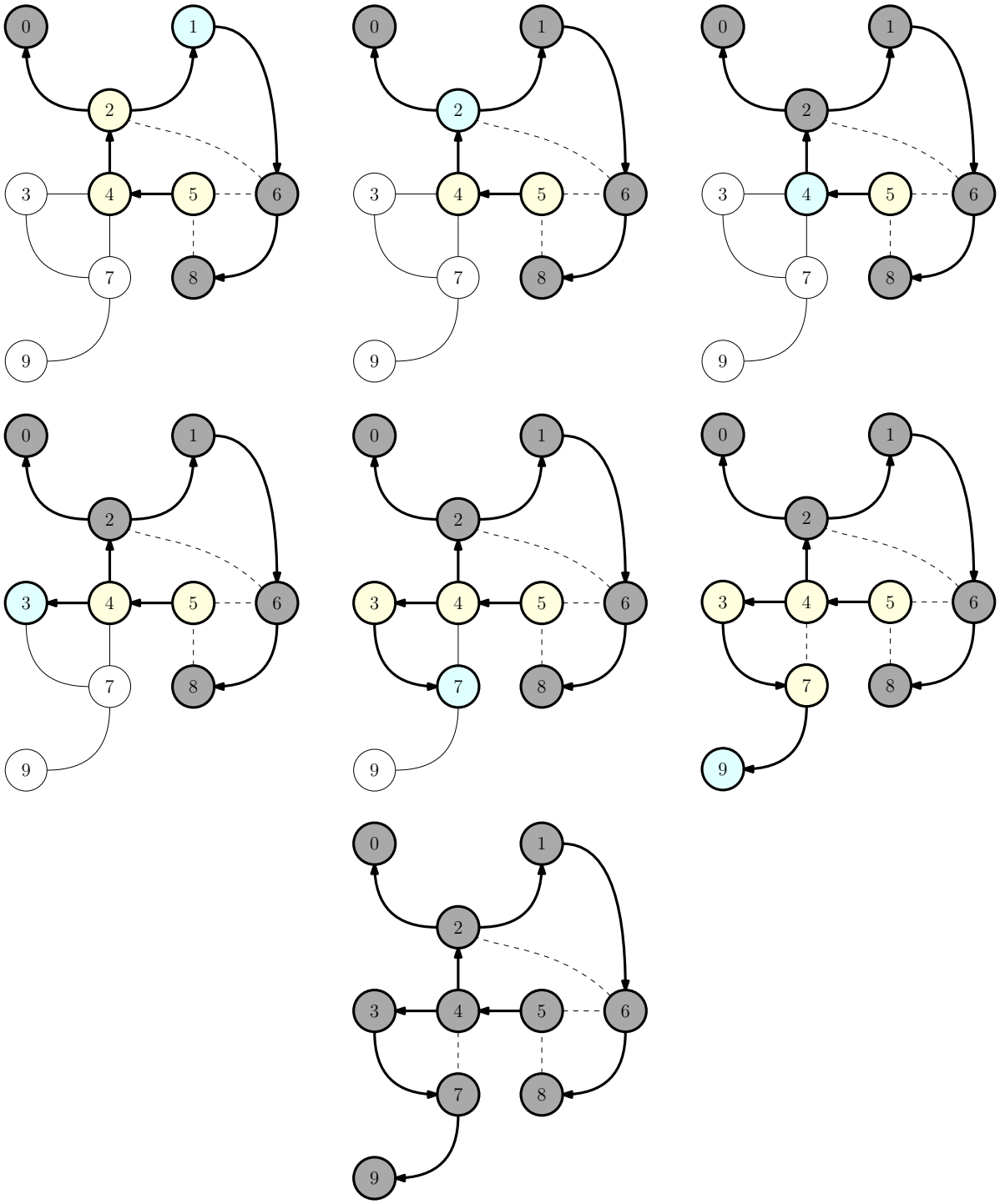
```
1:  fonction DFS-complet(G)
2:      vus ← ∅
3:      fonction Explorer(x)
4:          si x ∉ vus alors
5:              vus ← vus ∪ {x}
6:              pré-traitement(x)
7:              pour y ∈ successeurs(x) faire
8:                  Explorer(y)
9:              fin pour
10:             post-traitement(x)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

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.
- ⇨ 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` [1].



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



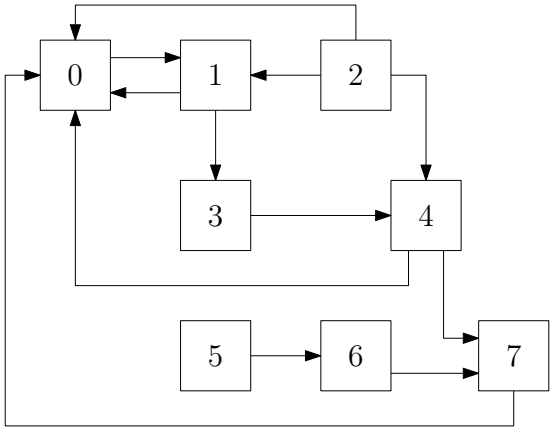
Parcours en profondeur d'un graphe connexe non orienté.

Exercice 2

⇨ 1. Écrire une fonction `affiche_dfs` qui effectue un parcours en profondeur complet du graphe passé en argument (donné sous forme de tableau de listes d'adjacence), dans l'ordre croissant des numéros de sommets. On affichera `Ouverture i` quand on commence à explorer le sommet i et `Fermeture i` quand on termine l'exploration.

```
affiche_dfs : int list array -> unit
```

2. Simuler l'exécution de cette fonction sur le graphe ci-dessous, en supposant que les listes d'adjacence sont triées par ordre croissant.




```
let affiche_dfs g =
  let n = Array.length g in
  let vus = Array.make n false in
  let rec explore x =
    if not vus.(x) then begin
      vus.(x) <- true;
      Printf.printf "Ouverture %d\n" x;
      List.iter explore g.(x);
      Printf.printf "Fermeture %d\n" x
    end in
  for x = 0 to n - 1 do
    explore x
  done
```

□

Théorème 2.1: 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 $O(|V|)$ 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 $O(|V|)$ sur la pile peut être problématique si $|V|$ est grand. Une version itérative (ou récursive terminale) efficace est présentée à l'exercice **??**, 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 $v = x_1, \dots, x_n = x$ 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 $O(|V|)$.

□

Exercice 3

⇨ Donner trois familles 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.

Solution. 1. Si l'on considère un graphe entièrement déconnecté à n sommets, on aura besoin d'un espace proportionnel à n pour stocker l'ensemble des sommets déjà visités, et le parcours se fera en temps $O(n)$. On remplira rapidement la mémoire disponible, donc le facteur limitant est l'espace.

2. Si l'on prend un graphe complet à n sommets, l'espace nécessaire pour l'ensemble *vus* est en $O(n)$, tout comme la profondeur de récursion. En revanche, le temps est en $O(n^2)$ puisqu'il faut considérer toutes les arêtes : on sera limité par le temps.
3. Si l'on prend un graphe chemin à n sommets, le temps et l'espace sont en $O(n)$, tout comme l'espace sur la pile. En effet, pour un graphe $x_1 - x_2 - \cdots - x_n$ où l'on commence par explorer x_1, la profondeur de récursion sera de n quand on arrivera à x_n. Ici, le facteur limitant sera l'espace sur la pile d'appel.

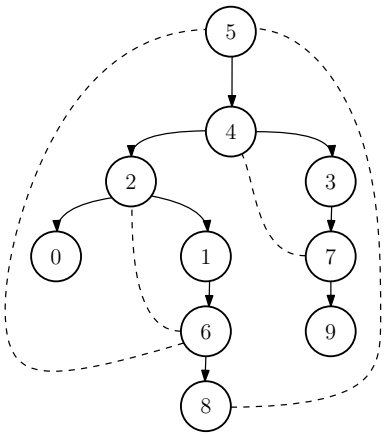
□

Théorème 2.2: 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 u ;
— 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* u.



Arbre associé au parcours en profondeur de la figure 2.1. Les arêtes en pointillé sont les arêtes du graphe initial qui ne sont pas conservées dans l'arbre.

Remarques

⇨ 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.

⇨ Si l'on fait un parcours en profondeur complet (fonction DFS-COMPLET), on obtiendra une forêt avec exactement un arbre par composante connexe.

⇨ La situation est plus complexe pour un graphe orienté : nous traiterons ce cas en exercice.

Exercice 4

⇨ Dessiner l'arbre de parcours en profondeur obtenu pour le graphe de la figure 2.1 si l'on explore à partir du sommet 3.

2.2 Parcours en largeur

Définition 2.3: Distance dans un graphe non pondéré

Soit G un graphe, éventuellement orienté mais non pondéré. La distance $\mathrm{d}(x, y)$ 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, $\mathrm{d}(x, y) := \infty$.

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 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 $x = y$).

Proposition 2.4

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

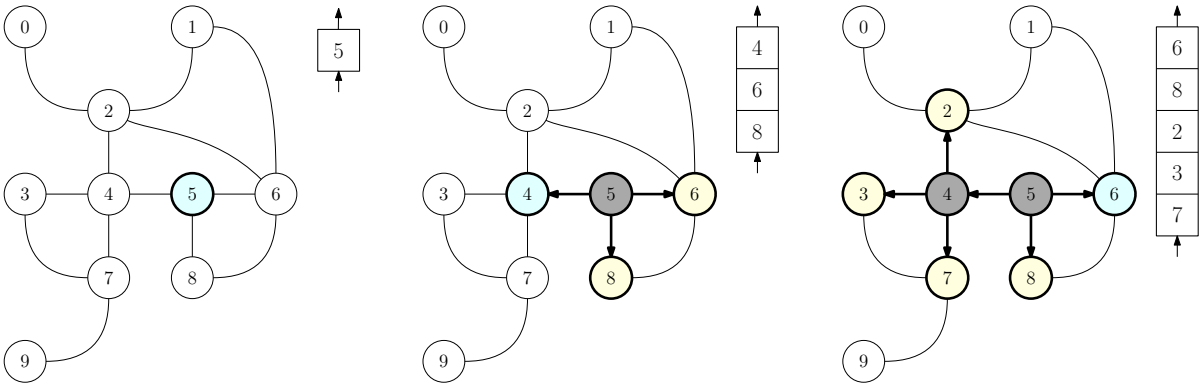
Le *parcours en largeur* d'un graphe à partir d'un sommet v permet de visiter les sommets par distance croissante à v :

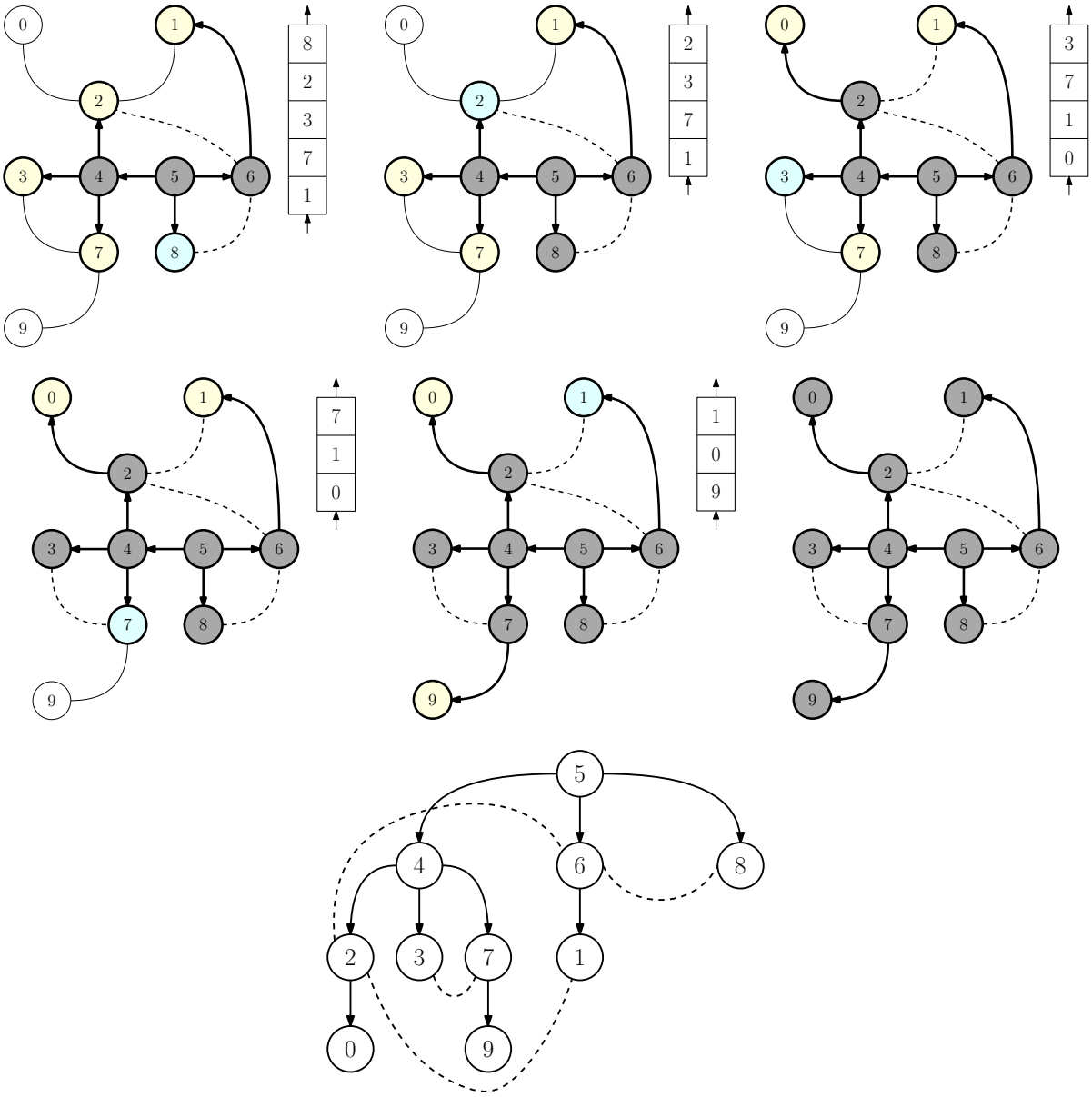
Algorithme 2 Parcours en largeur (*Breadth-First Search*) à l'aide d'une file.

```
1:  fonction BFS(G, x_0)
2:      ouverts ← file_vide()
3:      PUSH(x_0, ouverts)
4:      vus ← {x_0}
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}
12:             fin si
13:         fin pour
14:     fin tant que
15: fin fonction
```

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).





Parcours en largeur d'un graphe non orienté.

Théorème 2.5: 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 sont ajoutés à la file.

À un instant donné, un sommet sera dit :
- *ouvert* s'il appartient à *ouverts*[2] ;
- *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 :

(1) la file a la forme $x_{k+1}, \ldots, x_{k+p}$ avec $d_k \leqslant d_{k+1} \leqslant \ldots \leqslant d_{k+p} \leqslant d_k + 1$ (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* $i < k$) ;

2. Wow !

(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 **??**, 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.

□

Exercice 5

⇨ 1. Écrire une fonction `bfs` qui effectue un parcours en largeur d'un graphe à partir d'un sommet x_0 fourni en argument. Cette fonction affichera les sommets du graphe par distance croissante à x_0. On supposera que le graphe est donné sous forme d'un tableau de listes d'adjacence, et l'on pourra utiliser le module `Queue` pour réaliser la file.
   — `Queue.create : unit -> 'a Queue.t` crée une file vide.
   — `Queue.is_empty : 'a Queue.t` teste si une file est vide.
   — `Queue.push : 'a -> 'a Queue.t -> unit` ajoute un élément à la file.
   — `Queue.pop : 'a Queue.t -> 'a` extrait un élément de la file (qui doit être non vide).

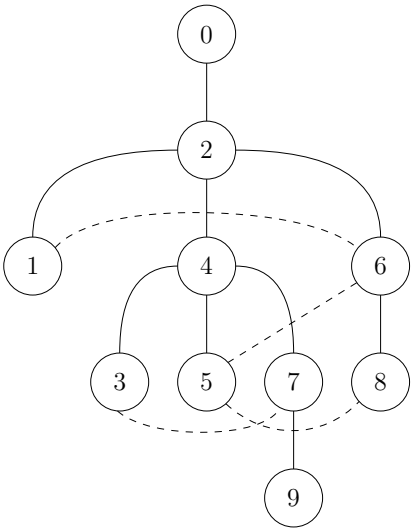
```
bfs : int list array -> int -> unit
```

2. Simuler l'exécution de cet algorithme sur le graphe de la figure 2.1 à partir du sommet 0, et dessiner l'arbre de parcours correspondant.
3. Si l'on ne souhaite pas utiliser le module `Queue`, comment peut-on réaliser de manière efficace une file impérative ? fonctionnelle ? On ne demande pas d'implémenter ces structures mais simplement de se remémorer les techniques que nous avons vues.

Solution. 1.

```
let bfs g initial =
  let vus = Array.make (Array.length g) false in
  let file = Queue.create () in
  let ajoute x =
    if not vus.(x) then begin
        Printf.printf "%d\n" x;
        vus.(x) <- true;
        Queue.push x file
      end in
  ajoute initial;
  while not (Queue.is_empty file) do
    let x = Queue.pop file in
    List.iter ajoute g.(x);
  done
```

2. On affiche dans l'ordre 0, 2, 1, 4, 6, 3, 7, 5, 8, 9. L'arbre de parcours correspondant est :



3. Une file impérative peut être réalisée par un tableau circulaire (de taille fixée ou dynamique), par une liste doublement chaînée circulaire, par une liste simplement chaînée mutable pour laquelle on garde un pointeur vers chaque extrémité... Toutes ces variantes offrent des opérations élémentaires en temps constant (amorti dans le cas d'un tableau dynamique). Une file fonctionnelle peut être réalisée de manière efficace par un couple de listes fonctionnelles (temps constant amorti pour l'extraction, temps constant pour l'insertion).

□

Théorème 2.6: Complexité du parcours en largeur

Le parcours en larguer a une complexité spatiale en $\Theta(|S|)$. En supposant que les opérations élémentaires sur les files et les ensemblescse font en temps constant, sa complexité temporelle est en $O(|S| + |A|)$.

Démonstration. Si l'on réalise `vus` par un tableau de $|S|$ booléens, l'initialisation se fait en temps $\Theta(|S|)$. 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 $O(|S| + |A|)$, donc la complexité temporelle totale est en $O(|S| + |A|)$.
Pour l'espace, on consomme $\Theta(|S|)$ pour `vus` et $O(|S|)$ sur la file (puisque tous les sommets présents sur la file sont distincts). Au total, la complexité spatiale est donc en $O(|S|)$. □

Exercice 6

⇨ *Parcours en largeurs avec générations explicites* : On considère l'algorithme suivant :

Algorithme 3 Parcours en largeur avec générations explicites.

```
fonction BFS(G, x_0)
    vus ← {x_0}                                   ▷ Ensemble
    actuels ← [x_0]                               ▷ Liste
    tant que actuels ≠ ∅ faire
        nouveaux ← []
        pour x ∈ actuel faire
            Traiter x.
            pour y ∈ G.successeurs(x) faire
                si y ∉ vus alors
                    Ajouter y à nouveaux.
                    Ajouter y à vus.
                fin si
            fin pour
        fin pour
        actuels ← nouveaux
    fin tant que
fin fonction
```

1. Justifier que cet algorithme termine.
2. Énoncer un invariant permettant de montrer que cet algorithme effectue bien un parcours en largeur (qu'il

traite exactement les sommets accessibles depuis x_0, par distance croissante à x_0).

3. Rédiger la preuve de correction de l'algorithme.
4. Écrire en OCaml une fonction `tableau_distances` prenant en entrée un graphe sous forme d'un tableau de listes d'adjacences et un sommet x_0, et utilisant l'algorithme ci-dessus pour calculer un tableau `dist` tel que `dist.(i)` soit la distance entre x_0 et le sommet i. On mettra une valeur de -1 dans le tableau pour les sommets qui ne sont pas accessibles depuis x_0.

```
tableau_distances : int list array -> int -> int array
```

Solution. 1. Notons $f(p)$ le nombre de sommets du graphe qui ne sont pas dans vus après p passages dans la boucle externe. Ce nombre est décroissant au sens large puisqu'un sommet n'est jamais retiré de vus ; de plus, si $f(p) = f(p+1)$, alors aucun sommet ne sera ajouté à $nouveaux$ lors de l'itération p, et l'on aura donc $actuels = \varnothing$ au début de l'itération $p+1$. $f(p)$ est donc une quantité entière, positive, qui décroît strictement tant que l'on ne sort pas de la boucle externe : cela assure la terminaison de cette boucle, et donc de la fonction.

2. L'invariant à considérer est le suivant : « après p passages dans la boucle externe, $actuels$ contient exactement les sommets x tels que $d(x_0, x) = p$, et vus contient exactement les sommets x tels que $d(x_0, x) \leqslant p$ ».
3. L'invariant est clairement respecté au départ avec $p = 0$: le seul sommet à distance 0 est x_0, qui est le seul sommet dans actuels et le seul sommet dans vus.
   Si l'on suppose l'invariant respecté au début d'une itération, et que l'on note $actuels'$, vus' les valeurs en fin d'itération, on a :
   — $actuels' = \{x \in V \setminus vus \mid \exists y \in actuels,\ y \to x \in E\}$
   — $vus' = vus \cup actuels'$
   Les sommets de $actuels'$ sont clairement à distance inférieure ou égale à $p+1$ de x_0 (successeurs de sommets de $actuels$, situés à distance p d'après l'invariant). Ils ne sont pas à distance inférieure ou égale à p puisqu'ils ne sont pas dans vus (invariant). Donc ils sont tous à distance exactement $p+1$.
   De plus, un sommet à distance $p+1$ est nécessairement un successeur d'un sommet à distance p, et il n'est pas dans vus : ainsi, $actuels'$ contient exactement les sommets à distance $p+1$ du sommet initial.
   vus' est donc l'union de vus (ensemble des sommets à distance au plus p) et de $actuels'$ (ensemble des sommets à distance $p+1$) : c'est bien exactement l'ensemble des sommets à distance au plus $p+1$ du sommet initial.
   On explore donc les sommets par distance croissante au sommet initial : c'est bien un parcours en largeur.
4. On peut par exemple procéder ainsi :

```
let tableau_distances g x0 =
  let n = Array.length g in
  let d = Array.make n (-1) in
  let vus = Array.make n false in
  d.(x0) <- 0;
  vus.(x0) <- true;
  (* ajoute les x non vus de sommets à la liste nouveaux *)
  let rec ajoute sommets nouveaux =
    match sommets with
    | [] -> nouveaux
    | x :: xs when not vus.(x) ->
      vus.(x) <- true;
      ajoute xs (x :: nouveaux)
    | _ :: xs -> ajoute xs nouveaux in
  (* les sommets de actuels sont à distance p de x0 *)
  let rec loop actuels nouveaux p =
    match actuels, nouveaux with
    | [], [] -> ()
    | [], _ -> loop nouveaux [] (p + 1)
    | x :: xs, _ ->
      d.(x) <- p;
      let nouveaux' = ajoute g.(x) nouveaux in
      loop xs nouveaux' p in
  loop [x0] [] 0;
  d
```

□

3 Test d'acyclicité et tri topologique

3.1 Ordre d'exploration lors d'un parcours en profondeur

Lors d'un parcours en profondeur, on dit qu'un sommet est
— *non vu* si l'on n'a pas commencé à l'explorer.
— *ouvert* si son exploration a commencé, mais n'est pas encore terminée.
— *fermé* si son exploration est terminée.

Pour un sommet x, on définit $\text{pre}(x)$ et $\text{post}(x)$ comme les instants d'ouverture et de fermeture de x comme dans le code suivant :

```
type statut =
  | Inconnu
  | Ouvert
  | Ferme

let dfs (g : int list array) : statut array * int array * int array =
  let n = Array.length g in
  let stat = Array.make n Inconnu in
  let pre = Array.make n (-1) in
  let post = Array.make n (-1) in
  let compteur = ref 0 in
  let rec explore (x : int) : unit =
    if stat.(x) = Inconnu then (
      stat.(x) <- Ouvert;
      pre.(x) <- !compteur;
      incr compteur;
      List.iter explore g.(x);
      stat.(x) <- Ferme;
      post.(x) <- !compteur;
      incr compteur)
  in
  for i = 0 to n - 1 do
    explore i
  done;
  (stat, pre, post)
```

Proposition 3.1: imbriquement-exploration

Si y est ouvert pendant l'exploration de x, c'est-à-dire si $\text{pre}(x) < \text{pre}(y) < \text{post}(x)$, alors :
— y est accessible depuis x.
— $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$.

De même, si $\text{pre}(x) < \text{post}(y) < \text{post}(x)$, on a y accessible depuis x et $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$.

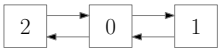
Exercice 7

⇨ Les propositions suivantes sont-elles correctes ? Donner une démonstration ou un contre-exemple.

1. Si y est accessible depuis x, alors l'ouverture de y aura lieu pendant l'exploration de x.
2. Si y est accessible depuis x et si $\text{pre}(x) < \text{pre}(y)$, alors $\text{pre}(y) < \text{post}(x)$.
3. S'il y a un arc xy et si $\text{pre}(x) < \text{pre}(y)$, alors $\text{pre}(y) < \text{post}(x)$.

Solution. 1. C'est faux : si l'on commence l'exploration du graphe $0 \leftrightarrow 1$ en 0, on a 0 accessible depuis 1 et pourtant 0 ne sera pas ouvert pendant l'exploration de 1 (il est déjà ouvert).

2. C'est également faux. On peut considérer le graphe suivant, en commençant l'exploration au sommet 0 :



Le parcours se déroule comme suit :
— ouverture 0 ;
— ouverture 1 ;
— fermeture 1 ;
— ouverture 2 ;
— fermeture 2 ;
— fermeture 0.

On a donc $pre(1) < pre(2)$ et 2 accessible depuis 1, et pourtant 2 n'est pas ouvert pendant l'exploration de 1.

3. Cette fois c'est vrai. Considérons l'état de y juste avant la fermeture de x :
   — y ne peut pas être vierge, puisqu'on a considéré l'arc xy pendant l'appel `explore x` et qu'on aurait alors ouvert y ;
   — on a donc $pre(x) < pre(y) < post(x)$, d'où $pre(x) < pre(y) < post(y) < post(x)$ d'après la proposition **??**.

□

3.2 Condition d'acyclicité

Proposition 3.2

Un graphe orienté est acyclique si et seulement si on ne tombe jamais sur un sommet ouvert lors de son parcours en profondeur.

Remarque

⇨ Autrement dit, le graphe est acyclique si et seulement si, lors du test ligne 10, le sommet considéré est toujours inconnu ou fermé, ou, ce qui revient au même, qu'un sommet x n'est jamais ouvert lorsqu'on appelle `explore x`.

Démonstration. Supposons que x soit ouvert au moment de l'appel `explore x`. Ce n'est pas possible si l'appel vient de la boucle `for` principal, donc `explore x` a forcément été appelé par `explore y` pour un certain y. On a donc :
— un arc yx puisque `explore y` a appelé `explore x` ;
— $pre(x) < post(y) < post(x)$ puisque x est ouvert au moment où l'on considère l'arc yx.
Or le deuxième point implique que y est accessible depuis x d'après la proposition refprop :imbrication-exploration : on a donc un chemin de x vers y et un arc yx, et donc un cycle.

Supposons maintenant que le graphe possède un cycle. Tous les sommets de ce cycle seront ouverts puis fermés pendant le parcours, on peut donc considérer le dernier sommet x du cycle à être fermé. Notons $y \to x \to \cdots \to y$ le cycle, et intéressons-nous à l'état de x au moment où l'on considère l'arc yx :
— x ne peut être fermé puisque y est encore ouvert et que x est le dernier sommet du cycle à être fermé ;
— x ne peut être vierge, puisque sinon on ouvrirait x à ce moment et l'on aurait $pre(y) < pre(x) < post(y)$ et donc $pre(y) < pre(x) < post(x) < post(y)$;
— x est donc ouvert, ce qui achève la démonstration du sens indirect (par contraposée).

□

Exercice 8

⇨ *Détection de cycle* : On travaille avec des graphes donnés sous forme d'un tableau de listes d'adjacence :

```
type sommet = int
type graphe = sommet list array
```

1. Écrire une fonction OCaml `est_dag` qui détermine si un graphe orienté est acyclique.

```
est_dag : graphe -> bool
```

2. Écrire une fonction `est_foret` qui détermine si un graphe non orienté est acyclique.

```
est_foret : graphe -> bool
```

Remarquons qu'il n'est pas possible d'écrire une fonction traitant correctement les deux cas.

3. **Optionnel.** Écrire une fonction `cycle` prenant en entrée un graphe orienté et renvoyant :
   — `None` si le graphe ne possède pas de cycle.
   — `Some u`, où u est une liste de sommets du graphe formant un cycle orienté, si le graphe possède un cycle.

```
cycle : graphe -> sommet list option
```

Solution. 1.

```
type statut = Inconnu | Ouvert | Ferme

exception Cycle

let est_dag (g : int list array) : bool =
  let n = Array.length g in
  let statuts = Array.make n Inconnu in
  let rec explore (x : int) : unit =
    match statuts.(x) with
    | Ouvert -> raise Cycle
    | Inconnu ->
      statuts.(x) <- Ouvert;
      List.iter explore g.(x);
      statuts.(x) <- Ferme
    | Ferme -> () in
  try
    for i = 0 to n - 1 do
      explore i
    done;
    true
  with
  | Cycle -> false
```

2. Pour un graphe non orienté, il ne faut pas considérer l'aller-retour le long d'une arête. On peut écrire :

```
let est_foret (g : int list) : bool =
  let n = Array.length g in
  let statuts = Array.make n Inconnu in
  let rec explore (depuis : int) (x : int) : unit =
    match statuts.(x) with
    | Ouvert -> raise Cycle
    | Inconnu ->
      statuts.(x) <- Ouvert;
      List.iter (fun y -> if y <> depuis then explore x y) g.(x);
      statuts.(x) <- Ferme
    | Ferme -> () in
  try
    for i = 0 to n - 1 do
      explore (-1) i
    done;
    true
  with
  | Cycle -> false
```

Une autre solution (plus simple, honnêtement) est de compter le nombre n de sommets, c de composantes connexes et p d'arêtes. Un graphe non orienté est acyclique si et seulement si $c = n - p$.

3. On garde le chemin actuel (qui correspond à la pile d'appel) en mémoire. Si l'on détecte un cycle, ce chemin est de la forme $x \leftarrow y \leftarrow \ldots \leftarrow x \leftarrow \ldots \leftarrow z$: on extrait la partie $x \leftarrow \ldots \leftarrow x$ puis on la remet à l'endroit.

```
exception Cycle of int list

let extraire_cycle chemin =
  let rec aux chemin sommet =
    match chemin with
    | x :: xs when x = sommet -> [x]
    | x :: xs -> x :: aux xs sommet
    | [] -> failwith "mauvais chemin" in
  match chemin with
  | x :: xs ->  x :: aux xs x
  | [] -> failwith "mauvais chemin"
```

```
let cycle g =
  let n = Array.length g in
  let statuts = Array.make n Inconnu in
  let rec explore chemin x =
    match statuts.(x) with
    | Ouvert -> raise (Cycle (x :: chemin))
    | Inconnu ->
      statuts.(x) <- Ouvert;
      List.iter (explore (x :: chemin)) g.(x);
      statuts.(x) <- Ferme
    | Ferme -> () in
  try
    for i = 0 to n - 1 do
      explore [] i
    done;
    None
  with
  | Cycle chemin ->
    let cycle = extraire_cycle chemin in
    Some (List.rev cycle)
```

□

3.3 Tri topologique

Le tri topologique d'un graphe orienté (donné sous forme d'un tableau de listes d'adjacence) peut être réalisé en temps linéaire en la taille du graphe par l'algorithme ci-dessous :

Algorithme 4 Tri topologique d'un graphe orienté acyclique

Entrées : G est un graphe orienté
Sorties : Une liste tri constituant un tri topologique des sommets de G, ou une erreur si G n'est pas un DAG.

fonction TriTopologique(G)
  $tat \leftarrow (Inconnu, \dots, Inconnu)$ ▷ Tableau de taille n
  $tri \leftarrow \varnothing$ ▷ Liste
  fonction Explore(v)
    si $tat[v] = ouvert$ **alors**
      Erreur : le graphe possède un cycle.
    sinon si $tat[v] = Inconnu$ **alors**
      $tat[v] \leftarrow ouvert$
      pour $v' \in successeurs(v)$ **faire**
        Explore(v')
      fin pour
      $tat[v] \leftarrow ferm$
      $tri \leftarrow v, tri$
    fin si
  fin fonction
  pour $v \in V$ **faire**
    Explore(v)
  fin pour
  renvoyer tri
fin fonction

Démonstration. La détection de cycle fonctionne comme à l'exercice **??**, on peut donc supposer que le graphe est acyclique et que l'algorithme renvoie une liste tri. Le fait que chaque sommet de G apparaisse exactement une fois dans tri découle directement de la correction de l'algorithme de parcours en profondeur. Il reste donc à prouver que, pour tout sommet x, si l'arc xy existe, alors x est avant y dans tri. C'est bien le cas, puisque :

- si $pre(x) < pre(y)$, alors on appellera Explore(y) pendant l'appel Explore(x), ce qui aura pour effet d'ajouter y à tri s'il n'y est pas déjà, puis on ajoutera x en tête de tri (et donc devant y) ;
- si $pre(y) < pre(x)$, alors $post(y) < pre(x)$ (sinon on aurait un cycle $y \to \cdots \to x \to y$), donc y est déjà dans tri au moment où l'on ajoute x en tête.

□

Exercice 9

⇨ *Tri topologique en OCaml* : Écrire une fonction `tri_topologique` renvoyant un tri topologique du graphe passé en argument, sous la forme d'une liste. On lèvera l'exception `Cycle` si un tel tri n'existe pas.

```
exception Cycle
tri_topologique : graphe -> sommet list
```

```
type statut = Inconnu | Ouvert | Ferme
exception Cycle

let tri_topologique (g : int list array) : Some int list =
  let n = Array.length g in
  let marques = Array.make n Inconnu in
  let pile = ref [] in
  let rec explore (i : int) : unit =
    match marques.(i) with
    | Ouvert -> raise Cycle
    | Ferme -> ()
    | Inconnu ->
       marques.(i) <- Ouvert;
       List.iter explore g.(i) ;
       marques.(i) <- Ferme;
       pile := i :: !pile in
  try
    for i = 0 to n - 1 do
      explore i
    done;
    Some !pile
  with
  | Cycle -> None
```

□

4 Plus court chemin dans un graphe pondéré

4.1 Distance, plus court chemin

Définition 4.1: Graphe pondéré

Un *graphe pondéré* est un triplet $G = (S, A, \rho)$, où S est l'ensemble des sommets, A l'ensemble des arcs (ou des arêtes) et ρ est une application de A dans $\mathbb{R}$ qui à un arc associe son *poids*. On étend usuellement ρ en posant $\rho(xy) \coloneqq +\infty$ si $xy \notin A$.

Remarques

⇨ ρ sera souvent à valeurs dans $\mathbb{R}_+$, mais pas systématiquement.

⇨ Pour un graphe non orienté, on aura systématiquement $\rho(xy) = \rho(yx)$, mais dans cette partie nous verrons essentiellement un graphe non orienté comme un cas particulier de graphe orienté.

Définition 4.2: Poids d'un chemin

Si $c \coloneqq x_0, \dots, x_k$ est un chemin, on définit le *poids* de c par

$$\rho(c) \coloneqq \sum_{i=0}^{k-1} \rho(x_i x_{i+1}).$$

Autrement dit, le poids d'un chemin est la somme des poids des arcs qui le composent.

Remarque

⇨ Il vaut mieux éviter de parler de la longueur d'un chemin dans un graphe pondéré : il y a une ambiguïté entre le nombre d'arcs et le poids total.

Définition 4.3: Distance dans un graphe pondéré

Si G est un graphe pondéré ne possédant pas de cycle de poids strictement négatif, on définit la *distance* de x à y par

$$\mathrm{d}(x,y) \coloneqq \inf\{\rho(c) \mid c \text{ chemin de } x \text{ à } y\}$$

Un *plus court chemin* de x à y est un chemin c de x à y vérifiant $\rho(c) = \mathrm{d}(x,y)$.

Remarques

⇨ Cette définition a bien un sens, puisque la condition sur l'absence de cycle de poids négatif permet de se limiter aux chemins élémentaires (à partir d'un chemin quelconque de x à y de poids p, on peut toujours obtenir un chemin élémentaire de x à y de poids $p' \leqslant p$ en supprimant les cycles). Les chemins élémentaires étant en nombre fini (ils possèdent au plus $n-1$ arêtes), on a en fait $\mathrm{d}(x,y) = \min\{\rho(c) \mid c \text{ chemin élémentaire de } x \text{ à } y\}$.

⇨ On peut ici avoir une distance négative, puisqu'on n'exige pas que les arcs soient à poids positif.

4.2 Algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall permet de calculer $\mathrm{d}(x,y)$ pour tous les couples (x,y) d'un graphe pondéré G sans cycle de poids négatif. Il s'agit d'un algorithme de programmation dynamique, qui utilise les deux observations suivantes pour faire apparaître une sous-structure optimale exploitable :

— Si $c : x = v_0 \to v_1 \to \cdots \to v_k = y$ est un plus court chemin de x à y, alors on peut supposer que les v_i sont distincts.

— Sous les mêmes hypothèses, $v_0 \to \cdots \to v_i$ et $v_i \to \cdots \to v_k$ sont des plus courts chemins, respectivement de v_0 a v_i et de v_i à v_k.

Proposition 4.4

On suppose donnée une numérotation $x_0, \dots, x_{n-1}$ des sommets du graphe. Pour $0 \leqslant k \leqslant n$, on définit $\mathrm{d}_k(x_i,x_j)$ comme le poids minimum d'un chemin de x_i à x_j dont tous les sommets (sauf éventuellement les extrémités) appartiennent à $\{x_0,\dots,x_{k-1}\}$. On pose $\mathrm{d}_k(x_i,x_j) = +\infty$ si un tel chemin n'existe pas. On a alors

$$\begin{cases} \mathrm{d}_0(x_i,x_j) = \rho(x_ix_j) \\ \mathrm{d}_{k+1}(x_i,x_j) = \min(\mathrm{d}_k(x_i,x_j), \mathrm{d}_k(x_i,x_k) + \mathrm{d}_k(x_k,x_j)) \quad \text{si } 0 \leqslant k < n \end{cases}$$

De plus, $\mathrm{d}_n(x_i,x_j) = \mathrm{d}(x_i,x_j)$ pour tout couple (x_i,x_j) de sommets.

Démonstration. Notons $E_k = \{x_0,\dots,x_{k-1}\}$ (avec donc $E_0 = \varnothing$) et disons qu'un chemin est *dans* E_k si tous ses sommets, sauf éventuellement ses extrémités, appartiennent à E_k.

— $d_0(x_i,x_j) = \rho(x_i,x_j)$ est immédiat, puisque le seul chemin éventuel dans E_0 de x_i à x_j est celui réduit à l'arc x_ix_j.

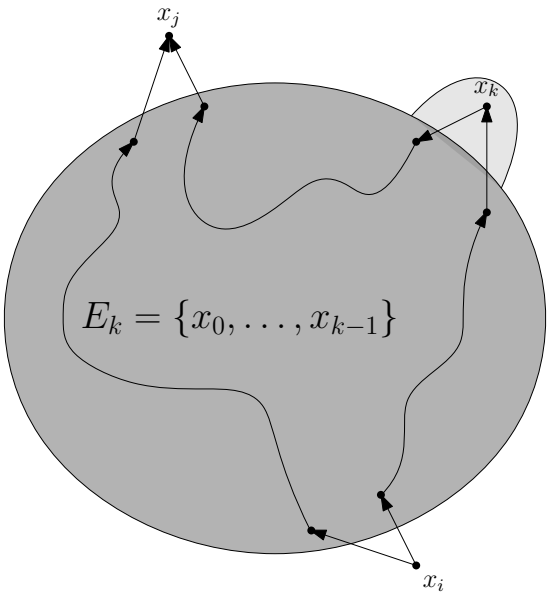
— Pour l'autre cas, un chemin dans E_{k+1} de x_i à x_j :

  — soit ne passe par x_k, auquel cas c'est un chemin dans E_k, et son poids minimum est $d_k(x_i,x_j)$;

  — soit passe par x_k. Dans ce cas, on peut supposer (à cause de l'absence de cycle de poids négatif, comme dit plus haut) qu'il n'y passe qu'une fois. Il est donc de la forme $x_i \underset{c}{\rightsquigarrow} x_k \underset{c'}{\rightsquigarrow} x_j$ avec c et c' dans E_k. Son poids minimum est donc $d_k(x_i,x_k) + d_k(x_k,x_j)$.

La dernière remarque découle directement de la définition de $d_n(x_i,x_j)$. □

La situation est illustrée par la figure ci-dessous (où l'on a représenté x_i et x_j à l'extérieur de E_k, ce qui n'est pas nécessairement le cas).



Les deux types de chemins à considérer pour le calcul de $d_{k+1}(x_i, x_j)$.

À partir de cette relation de récurrence, on obtient immédiatement un algorithme de programmation dynamique : il suffit de créer $n+1$ matrices de dimension (n, n) $D_0, \ldots, D_n$, d'initialiser D_0 à partir de la matrice d'adjacence pondérée du graphe, de calculer successivement $D_1, \ldots, D_n$ et de renvoyer D_n. On peut tout de suite remarquer que le calcul de D_{k+1} ne fait intervenir que D_k et qu'on peut donc se contenter de stocker deux matrices.

Algorithme 5 Floyd-Warshall, première version

Entrées : La matrice d'adjacence A d'un graphe pondéré à n sommets. $A[i, j] = +\infty$ si l'arc (x_i, x_j) n'existe pas.
Sorties : Une matrice carrée D de taille n tel que $D[i, j] = d(x_i, x_j)$.

```
1:  fonction FloydWarshall(A)
2:      Ancien ← copie(A)
3:      pour k = 0 à n − 1 faire
4:          Nouveau ← copie(Ancien)                          ▷ Ancien = D_k
5:          pour i = 0 à n − 1 faire
6:              pour j = 0 à n − 1 faire
7:                  Nouveau[i, j] ← min(Ancien[i, j], Ancien[i, k] + Ancien[k, j])
8:              fin pour
9:          fin pour
10:         Ancien ← Nouveau                                 ▷ Ancien = D_{k+1}
11:     fin pour
12:     renvoyer Ancien
13: fin fonction
```

On peut en fait n'utiliser qu'une seule matrice. En effet, juste avant de l'exécution de la ligne 7, on a :

- $Nouveau[i, j] = Ancien[i, j] = d_k(x_i, x_j)$ (puisque la case n'a pas encore été mise à jour) ;
- $Nouveau[i, k]$ qui est égal soit à $d_k(x_i, x_k)$, soit à $d_{k+1}(x_i, x_k)$ (suivant si la case (i, k) a déjà été traitée ou non). Mais de toute façon, un chemin minimal de x_i à x_k dans E_{k+1} est en fait dans E_k (ou en tout cas peut choisi dans E_k) : en effet, il n'y a pas de cycle de poids négatif, donc $\rho(\underbrace{x_i \ldots x_k}_{c \in E_k} \ldots x_k) \geqslant \rho(c)$. On a donc en fait $Nouveau[i, k] = d_k(x_i, x_k)$ dans tous les cas.
- De même, $Nouveau[k, j] = d_k(x_k, x_j)$.
- Finalement, la ligne peut être remplacée par $Nouveau[i, j] \leftarrow \min(Nouveau[i, j], Nouveau[i, k] + Nouveau[k, j])$ sans rien changer.

On en déduit l'algorithme final :

Algorithme 6 Floyd-Warshall, version finale

Entrées : La matrice d'adjacence A d'un graphe pondéré à n sommets. $A[i,j] = +\infty$ si l'arc (x_i, x_j) n'existe pas.
Sorties : Une matrice carrée D de taille n tel que $D[i,j] = d(x_i, x_j)$.

```
fonction FloydWarshall(A)
    D ← copie(A)
    pour k = 0 à n − 1 faire
        pour i = 0 à n − 1 faire
            pour j = 0 à n − 1 faire
                D[i,j] ← min(D[i,j], D[i,k] + D[k,j])
            fin pour
        fin pour
    fin pour
    renvoyer D
fin fonction
```

Proposition 4.5: Complexité de l'algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall permet d'obtenir la matrice de distance D d'un graphe $G := (S, A)$ en temps $O(|S|^3)$ et en espace $O(|S|^2)$ (espace qui correspond uniquement à la taille du résultat).

Exercice 10

⇨ *Reconstruction de plus courts chemins* : Si l'on ne dispose que de la matrice D, il n'y a pas de moyen efficace de reconstruire un plus court chemin entre deux sommets : il faut donc stocker un peu plus d'information.

1. Écrire une fonction `floyd_warshall` prenant en entrée une matrice d'adjacence pondérée (avec des `infinity` pour coder l'absence d'arc) et renvoie :
   — une matrice `d` comme plus haut ;
   — une matrice `prochain`, de même dimension que `d`, telle que :
     — `prochain.(i).(j)` vaut `None` si et seulement si `j` n'est pas accessible depuis `i` ;
     — si `prochain.(i).(j)` vaut `Some k`, alors l'un des plus courts chemins de `i` à `j` commence par l'arc $i \to k$ (ou alors $i = j$, auquel cas la valeur de `prochain.(i).(j)` n'a pas d'importance).

```
floyd_warshall : float array array
                 -> (float array array * int option array array)
```

2. Écrire une fonction `reconstruit` prenant en entrée une matrice `prochain` définie comme ci-dessus et deux indices i et j de sommets, et renvoyant un plus court chemin de i vers j sous forme d'une liste d'indices de sommets. On lèvera une exception si j n'est pas accessible depuis i.

```
reconstruit : int option array array -> int -> int -> int list
```

4.3 Algorithme de Dijkstra

L'algorithme de Dijkstra résout un problème similaire à celui de l'algorithme de Floyd-Warshall, avec cependant trois différences :
— On *se restreint au cas où les poids sont positifs.*
— Il y a un sommet source s distingué, et l'on souhaite calculer $d(s, x)$ pour tous les sommets x.
— L'algorithme est bien adapté au cas des graphes creux, et prend donc son argument sous forme d'un tableau de listes d'adjacence.

Essentiellement, l'idée est d'adapter le parcours en largeur au cas d'un graphe pondéré. Dans un parcours en largeur, le prochain sommet à explorer est systématiquement celui qui a été découvert en premier : on utilise donc une file. En réalité, ce qui nous intéresse dans ce sommet c'est qu'il s'agit du (ou d'un des) sommet le plus proche du sommet initial parmi ceux qui n'ont pas encore été explorés. Nous allons garder cet ordre d'exploration (par distance croissante au sommet initial) dans l'algorithme de Dijkstra, mais il faudra pour ce faire remplacer la file par une *file de priorité.*

On suppose ici que l'on dispose d'une structure de file de priorité fournissant les opérations suivantes :
— FilePrioVide() qui renvoie une nouvelle file de priorité ;
— EstVide(f) qui teste si la file f est vide ;
— ExtraireMin(f) qui renvoie un couple (c, p) de f pour lequel la priorité p est minimale, et l'enlève de la file ;
— Insérer(f, c, p) qui insère la clé c avec la priorité p dans la file f.
— DiminuerPriorité(f, c, p') qui a le comportement suivant :

Préconditions :
- la clé c est présente (une et une seule fois) dans la file f, avec une priorité p ;
- $p' \leqslant p$.

Effet : après l'appel, la clé c est toujours présente une et une seule fois dans la file, mais sa priorité est désormais p'.

Algorithme 7 Dijkstra

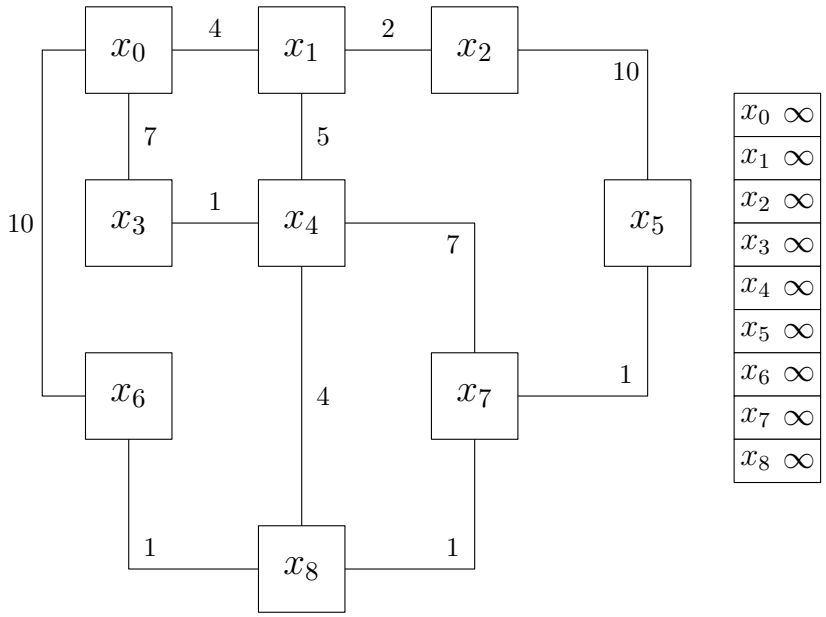
Entrées : un graphe pondéré à poids positifs G à n sommets et un sommet s
Sorties : un tableau $dist$ tel que $dist[k] = d(s, k)$ pour $0 \leqslant k < n$

```
fonction DIJKSTRA(G, s)
    dist ← (∞, ..., ∞)                                  ▷ Taille n
    dist[s] = 0
    ouverts ← FILEPRIOVIDE()                            ▷ File de priorité min
    INSÉRER(ouverts, s, 0)
    tant que ouverts ≠ ∅ faire
        (j, d_j) ← EXTRAIREMIN(ouverts)
        pour k successeur de j faire
            d ← d_j + ρ(j → k)
            si d < dist[k] alors
                si dist[k] < ∞ alors
                    DIMINUERPRIORITÉ(ouverts, k, d)
                sinon
                    INSÉRER(ouverts, k, d)
                fin si
                dist[k] ← d
            fin si
        fin pour
    fin tant que
    renvoyer dist
fin fonction
```

Exercices 11

⇨ *Terminaison de l'algorithme de Dijkstra* Montrer que l'algorithme ci-dessus termine.

⇨ Simuler à la main l'exécution de l'algorithme de Dijkstra sur le graphe suivant, en prenant $s = x_0$:



Proposition 4.6: Correction de l'algorithme de Dijkstra

Si $G = (V, E, \rho)$ est un graphe pondéré tel que $\rho(e) \geqslant 0$ pour tout $e \in E$, alors l'appel DIJKSTRA(G, s) renvoie un tableau $dist$ tel que $dist[k] = \mathrm{d}(s, x_k)$ pour tout $k \in [0 \dots n-1]$.

Démonstration. On peut déjà remarquer que, si un sommet x apparaît dans $ouverts$, alors $dist[x] < \infty$, et que dans ce cas il y apparaît avec la priorité $dist[x]$ (immédiat à la lecture du code). On dira que :

— un sommet x est *vierge* si $dist[x] = \infty$;
— un sommet est *ouvert* s'il est présent dans la file *ouverts* ;
— un sommet x est *fermé* si $dist[x] < \infty$ et il n'est pas ouvert ;
— un chemin est fermé si tous les sommets qui le composent, sauf éventuellement le dernier, sont fermés.

Pour chaque sommet j, on définit :
— $d(j) = d(s, j)$ la longueur d'un plus court chemin de s à j (ou ∞ si un tel chemin n'existe pas) ;
— $d_f(j)$ la longueur d'un plus court chemin fermé de s à j.

On considère à présent l'invariant de boucle suivant (valable à chaque début de boucle, pour tout sommet x) :
— $dist[x] = d_f(x)$;
— si x est fermé, alors $d_f(x) = d(x)$.

Initialisation Au départ, aucun sommet n'est fermé donc $d_f(s) = d(s) = 0$ (chemin vide) et $d_f(j) = \infty$ si $j \neq s$. C'est bien cohérent avec l'état initial de *dist*.

Invariance On suppose l'invariant respecté au début d'une itération, et soit j le sommet choisi (qui est donc ouvert pour l'instant, et que l'on ferme). Pour tout sommet x, on note $d'_f(x)$ et $dist'[x]$ les valeurs en fin d'itération.
— Pour le sommet j, on a $dist'[j] = dist[j] = d_f(j)$ (la première égalité par lecture du code, la seconde d'après l'invariant). Il faut donc prouver $d'_f(j) = d_f(j) = d(j)$. La première égalité est immédiate : le seul sommet supplémentaire autorisé dans $d'_f(j)$ est j lui-même, or on peut se restreindre aux chemins élémentaires puisque les poids sont positifs. De plus, on a $d_f(j) \geqslant d(j)$ par définition : il suffit donc de prouver $d_f(j) \leqslant d(j)$. Considérons un chemin c de s à j de poids total l.
  — Si les sommets intermédiaires de c sont tous fermés, alors $l \geqslant d_f(j)$ par définition.
  — Sinon, soit x le premier sommet ouvert ou vierge traversé par ce chemin : $\underbrace{init \to \cdots \to x}_{c'} \underbrace{\to \cdots \to j}_{c''}$. On a $l = \rho(c') + \rho(c'') \geqslant \rho(c')$ (les poids sont positifs, **hypothèse cruciale**), donc $l \geqslant d_f(x)$ puisque c' est un chemin fermé de s à x, puis $l \geqslant d_f(j)$ puisqu'on a extrait le minimum de *ouverts*.

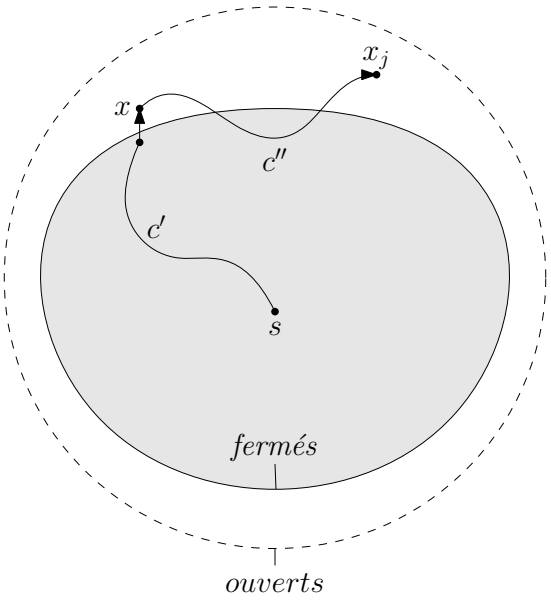


FIGURE 1 – Pour le sommet extrait, il existe un plus court chemin qui est fermé.

On a donc bien $d_f(j) = d(j)$.
— Pour les sommets $i \neq j$, les nouveaux chemins à considérer pour $d'_f(i)$ sont les chemins fermés passant par j. Si ce chemin se termine par $x \to i$ avec $x \neq j$ (et x fermé, nécessairement), alors on peut le remplacer par un chemin au moins aussi léger ne passant pas par j (puisque $d(x) = d_f(x)$). Il suffit donc de considérer les chemins de la forme $s \to x_1 \to \cdots \to x_k \to j \to i$ avec les x_i fermés en début d'itération, ce qui est exactement ce que fait le code : un tel chemin est de longueur minimale $d_f(j) + \rho(j \to i)$.

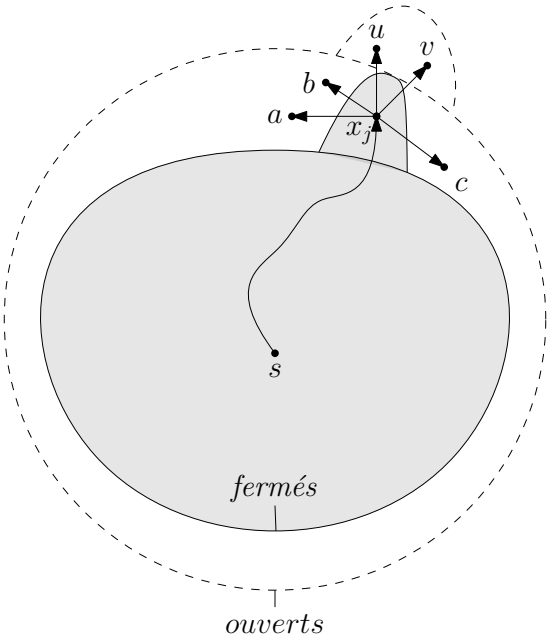


FIGURE 2 – Nouveaux chemins à considérer pour d'_f.

Conclusion À la fin de l'exécution, tous les sommets sont fermés donc le tableau *dist* contient bien les longueurs des plus courts chemins.

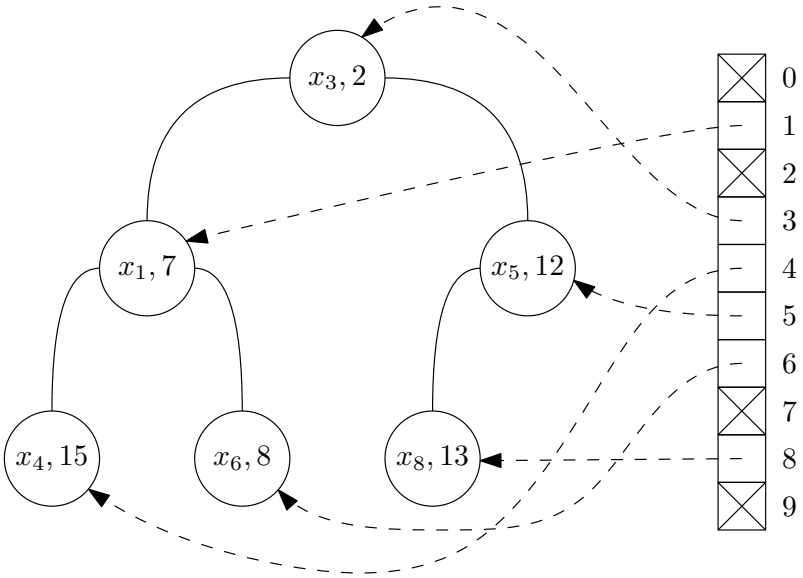
□

Proposition 4.7

Pour un graphe à n sommets et p arcs, la fonction DIJKSTRA effectue au plus :
- n opérations INSÉRÉR et n opérations EXTRAIREMIN.
- p opérations DIMINUERPRIORITÉ.

Toutes ces opérations se font sur une file de taille majorée par n.

La question qui se pose à présent est celle de la réalisation d'une structure de file de priorité permettant une opération DIMINUERPRIORITÉ efficace. On rappelle que la réalisation la plus classique d'une file de priorité est celle qui utilise un tas binaire stocké dans un tableau *donnes*. Ici, il suffit de rajouter un deuxième tableau *position* tel que $position[i]$ indique l'indice auquel se trouve le couple (i, d_i) associé au sommet i dans le tableau *donnes*.



`donnees`

$x_3, 2$	$x_1, 7$	$x_5, 12$	$x_4, 15$	$x_6, 8$	$x_8, 13$				

-1	1	-1	0	3	2	4	-1	5	-1

`position`

Tas binaire avec tableau indiquant la position des clés.

Pour exécuter une opération DiminuerPriorité sur le sommet x_i, on fait une percolation vers le haut de la case $position[i]$ du tableau *donnes*. Pendant cette percolation, on effectuera un certain nombre d'échanges dans le tableau *donnes* (entre père et fils) : à chaque fois, il faudra faire le même échange dans le tableau *position* pour maintenir la correspondance.

Exercice 12

⇨ Donner l'état des tableaux *donnes* et *position* de la figure **??** après chacune des opérations suivantes (on suppose qu'on les fait successivement) :
- Insérer($ouverts, x_0, 10$) ;
- DiminuerPriorité($ouverts, x_6, 1$) ;
- ExtraireMinimum($ouverts$).

Proposition 4.8

Avec la réalisation décrite ci-dessus pour la structure de file de priorité, la complexité temporelle de l'algorithme de Dijkstra est en $O\big((n+p)\log n\big)$.