Chapitre 4

Graphes orientés

Sommaire

Dans un grand nombre de cas pratiques, les arêtes des graphes que l’on utilise ne peuvent être suivies que dans un sens ; c’est le cas entre autres :

— des réseaux sociaux asymétriques comme Twitter, où le fait qu’un utilisateur A suive un utilisateur B n’oblige pas B à suivre A ;
— des réseaux routiers dans lesquels tous les segments de route ne peuvent pas nécessairement être empruntés dans les deux sens ;
— ou encore des graphes de dépendances entre tâches impliquées dans la réalisation d’un projet (avant d’installer la plomberie et l’électricité, il faut que les fondations soient en place).

On utilise dans ces cas-là des graphes *orientés* pour modéliser nos données. Dans ce chapitre, on couvrira les algorithmes basiques liés à cette catégorie de graphes, qu’on supposera non pondérés pour simplifier la discussion. On verra qu’on peut réaliser pas mal de choses sur base d’un simple algorithme de parcours, qu’on devra cependant décliner en plusieurs versions. Notre choix se portera sur l’algorithme de parcours en profondeur, qu’on implémentera de manière récursive.

4.1 Concepts de base

Les graphes orientés présentent des différences conceptuelles mineures avec les graphes non-orientés. Chaque arête orientée, que l’on qualifie plutôt d’*arc*, ayant par définition un sens, il importe de distinguer la manière dont sont connectés deux sommets. On utilise sou-

vent la notation $G = (V, A)$ [1] plutôt que la notation $G = (V, E)$ dont on se servait dans le cas non orienté. Dans le contexte des graphes non orientés, on disait que les sommets reliés par l'arête $\{u, v\}$ étaient *adjacents* ou *voisins*; ici, l'orientation nous oblige à être plus précis, et on remplacera la notation $\{u, v\}$ par (u, v) pour signifier que l'ordre importe.

Définition 18. Si $G = (V, A)$ est un graphe orienté et $(u, v) \in A$, alors u est un *prédécesseur* de v, et v est un *successeur* de u. Le degré d'un sommet est la somme de son *degré entrant* (le nombre de prédécesseurs) et de son *degré sortant* (le nombre de successeurs). Les notations sont les suivantes :

$$N_G^-(u) = \{v \mid (v, u) \in A(G)\} \quad \text{et} \quad deg_G^-(u) = |N_G^-(u)|;$$
$$N_G^+(u) = \{v \mid (u, v) \in A(G)\} \quad \text{et} \quad deg_G^+(u) = |N_G^+(u)|.$$

Les définitions qu'on a utilisées jusqu'ici doivent également être adaptées pour prendre en compte l'orientation des arcs quand c'est nécessaire. Par exemple :

1. un *chemin* dans un graphe **orienté** $G = (V, A)$ est une séquence de sommets **distincts** $P = (u_0, u_1, u_2, \ldots, u_{p-1})$, où $(u_i, u_{i+1}) \in A$ pour $0 \leq i \leq p-2$;
2. un *cycle* dans un graphe **orienté** $G = (V, A)$ est une séquence de sommets $C = (u_0, u_1, u_2, \ldots, u_{p-1})$, où $(u_i, u_{i+1 \mod p}) \in A$ pour $0 \leq i \leq p-1$.

Par abus de langage, on utilisera les mêmes termes dans les deux cas de figure. Enfin, le terme de *descendant* remplacera la notion de sommet "accessible" vue dans le cas des graphes non orientés.

Définition 19. Un *descendant* d'un sommet u dans un graphe orienté G est un sommet $v \neq u$ tel qu'il existe un chemin de u vers v dans G.

4.2 Implémentation

Les implémentations des graphes non orientés qu'on a vues précédemment ne changent pas énormément dans le cas orienté, et deviennent même souvent plus simples. Par exemple, lorsqu'on ajoute un arc plutôt qu'une arête dans une matrice d'adjacence, on ne doit plus modifier qu'une case dans la matrice. Pour être complet, le Tableau 4.1 reprend les méthodes que l'on s'attend à retrouver dans une hypothétique classe `GrapheOrienté`.

4.3 Parcours

Dans le cas non-orienté, on avait la garantie que les algorithmes de parcours exploreraient tous les sommets d'un graphe non orienté connexe. On ne peut plus les appliquer tels quels dans le cas orienté, car en fonction du sommet que l'on sélectionne comme point de départ, on pourrait ne pas être capable d'explorer tout le graphe.

1. Là encore, on suit la terminologie anglo-saxonne : le "A" vient de ***a**rcs* (le même mot qu'en français dans ce cas précis).

Nom	Description
`ajouter_arc(u, v)`	Ajoute un arc entre les sommets u et v, en créant au besoin ces sommets.
`ajouter_arcs(iterable)`	Ajoute tous les arcs de l'itérable donné au graphe.
`ajouter_sommet(sommet)`	Ajoute un sommet au graphe.
`ajouter_sommets(iterable)`	Ajoute tous les sommets de l'itérable donné au graphe.
`arcs()`	Renvoie l'ensemble des arcs du graphe, représentés par des tuples.
`boucles()`	Renvoie les boucles du graphe, c'est-à-dire les arcs reliant un sommet à lui-même.
`contient_arc(u, v)`	Renvoie `True` si l'arc (u, v) existe, `False` sinon.
`contient_sommet(u)`	Renvoie `True` si le sommet u existe, `False` sinon.
`degre(sommet)`	Renvoie le nombre de voisins du sommet ; s'il n'existe pas, provoque une erreur.
`degre_entrant(sommet)`	Renvoie le nombre de prédécesseurs du sommet ; s'il n'existe pas, provoque une erreur.
`degre_sortant(sommet)`	Renvoie le nombre de successeurs du sommet ; s'il n'existe pas, provoque une erreur.
`nombre_arcs()`	Renvoie le nombre d'arcs du graphe.
`nombre_boucles()`	Renvoie le nombre d'arcs de la forme (u, u).
`nombre_sommets()`	Renvoie le nombre de sommets du graphe.
`predecesseurs(sommet)`	Renvoie l'ensemble des prédécesseurs du sommet donné.
`retirer_arc(u, v)`	Retire l'arc (u, v) s'il existe ; provoque une erreur sinon.
`retirer_arcs(iterable)`	Retire tous les arcs de l'itérable donné du graphe s'ils existent ; provoque une erreur sinon..
`retirer_sommet(sommet)`	Efface le sommet du graphe, et retire tous les arcs qui lui sont incidents.
`retirer_sommets(iterable)`	Efface les sommets de l'itérable donné du graphe, et retire tous les arcs incidents à ces sommets.
`sommets()`	Renvoie l'ensemble des sommets du graphe.
`sous_graphe_induit(iterable)`	Renvoie le sous-graphe induit par l'itérable de sommets donné.
`successeurs(sommet)`	Renvoie l'ensemble des successeurs du sommet donné.
`voisins(sommet)`	Renvoie l'ensemble des voisins du sommet donné.

TABLEAU 4.1 – Les méthodes de la classe abstraite `GrapheOrienté`.

Exemple 27. Soit G le graphe orienté ⓪⟶①. Parcourir G à partir de 0 permet d'atteindre tous ses sommets, mais parcourir G à partir de 1 ne permet pas d'atteindre 0.

Dès lors, si l'on veut pouvoir explorer tout le graphe, il faut réitérer le processus à partir de chacun des sommets. L'algorithme 18 montre comment explorer un graphe orienté à partir d'un sommet donné en profondeur de manière récursive.

Algorithme 18 : PARCOURSPROFONDEURORIENTÉREC(G, départ, déjà_visités)

Entrées : un graphe orienté G, un sommet de départ, et un tableau booléen déja_visités de $|V|$ cases.

Sortie : les sommets de G accessibles à partir du sommet de départ dans l'ordre où le parcours en profondeur les a découverts.

```
1 résultat ← liste();
2 résultat.ajouter_en_fin(départ);
3 déjà_visités[départ] ← VRAI;
4 pour chaque v ∈ G.successeurs(départ) faire
5     si ¬ déjà_visités[v] alors
6         résultat.ajouter_en_fin(PARCOURSPROFONDEURORIENTÉREC(G, v,
            déjà_visités));
7 renvoyer résultat;
```

L'algorithme 19 montre quant à lui comment adapter le parcours en largeur itératif au cas orienté.

Algorithme 19 : PARCOURSLARGEURITÉRATIFORIENTÉ(G, départ, déjà_visités=NIL)

Entrées : un graphe non-orienté G et un sommet de départ ; éventuellement, un tableau déjà_visités de $|V|$ cases indiquant les sommets déjà traités.

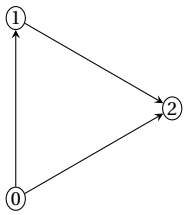
Sortie : la liste des sommets de G accessibles depuis le départ dans l'ordre où le parcours en largeur les a découverts.

```
1  résultat ← liste();
2  si déjà_visités = NIL alors déjà_visités ← tableau(G.nombre_sommets(), FAUX);
3  a_traiter ← file();
4  a_traiter.enfiler(départ);
5  tant que a_traiter.pas_vide() faire
6      sommet ← a_traiter.défiler();
7      si ¬ déjà_visités[sommet] alors
8          résultat.ajouter_en_fin(sommet);
9          déjà_visités[sommet] ← VRAI;
10         pour chaque successeur dans G.successeurs(sommet) faire
11             si ¬ déjà_visités[successeur] alors a_traiter.enfiler(successeur) ;
12 renvoyer résultat;
```

4.4 Détection de cycles (parcours tricolore)

Les cycles sont plus compliqués à détecter dans le cas des graphes orientés, car la présence de plusieurs chemins entre deux sommets n'implique plus nécessairement la présence d'un cycle.

Exemple 28. Un exemple de graphe orienté acyclique, qui contiendrait un cycle si l'on oubliait l'orientation des arcs :

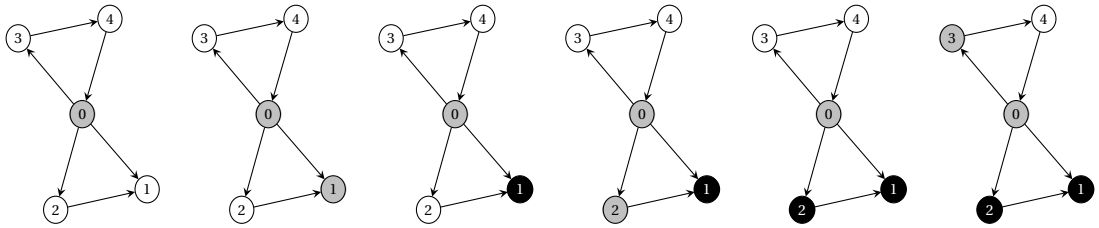


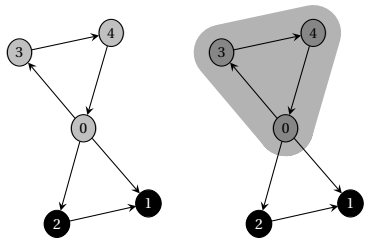
On peut détecter les cycles à l'aide d'un simple parcours en profondeur, mais il nous faut pouvoir faire la différence entre les sommets que l'on rencontre plusieurs fois à cause d'un cycle et ceux que l'on rencontre plusieurs fois à cause de la présence de plusieurs chemins dont l'union n'est pas un cycle. Dès lors, au lieu de partitionner les sommets en "visités" / "non visités", on utilisera un état intermédiaire supplémentaire "en cours de traitement". Pour être plus concis et plus accessible, on utilisera trois couleurs pour nos sommets :

1. "blanc" pour un sommet non encore visité ;
2. "gris" pour un sommet en cours de traitement, ce qui signifie que l'on est en train d'explorer ses descendants ;
3. "noir" pour un sommet que l'on a fini de traiter, ce qui veut dire qu'on a également exploré tous ses descendants.

Ainsi, passer sur un sommet blanc signifie qu'on le rencontre pour la première fois, et passer sur un sommet noir signifie qu'on a trouvé un autre chemin pour y arriver sans passer par ses descendants ; si l'on rencontre un sommet gris, cela signifie que l'on y est parvenu en passant par ses descendants, et donc qu'on a identifié un cycle.

Exemple 29. Voici les étapes de la détection de cycle suivant un parcours en profondeur tricolore au départ du sommet 0 :





(7) (8)

(fin exemple 29)

L'algorithme 20 montre une implémentation récursive de cet algorithme. On voit que ceci ressemble très fort à l'implémentation du parcours en profondeur récursif.

Algorithme 20 : CONTIENTCYCLEORIENTÉ(G, sommet, statuts)

Entrées : un graphe orienté G, un sommet de départ, et un tableau statuts de $|V|$ cases.

Sortie : VRAI si un cycle de G est accessible à partir du sommet de départ, FAUX sinon.

```
1 si statuts[sommet] = "gris" alors renvoyer VRAI;
2 si statuts[sommet] = "noir" alors renvoyer FAUX;
3 statuts[sommet] ← "gris";
4 pour chaque v ∈ G.successeurs(sommet) faire
5 |   si ContientCycleOrienté(G, v, statuts) alors renvoyer VRAI;
6 statuts[sommet] ← "noir";
7 renvoyer FAUX;
```

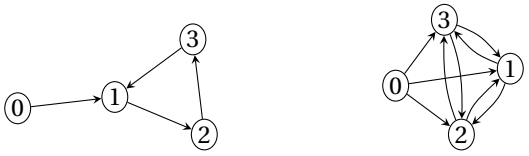
Attention : il est important de remarquer que ce n'est pas parce que l'algorithme 20 renvoie FAUX que le graphe est acyclique ! Simplement, cela signifie que l'on ne peut pas accéder à un cycle du graphe *au départ du sommet spécifié*. On peut constater cela sur le graphe de l'exemple 29, en tentant de démarrer l'exploration au départ des sommets 1 ou 2.

4.5 Fermeture transitive

Il est parfois utile d'enrichir la structure d'un graphe de manière à pouvoir répondre à certaines requêtes sur ce graphe plus rapidement.

Définition 20. La *fermeture transitive* d'un graphe orienté $G = (V, A)$ est le graphe orienté $G' = (V, A')$ avec $A' = \{(u, v) \mid v \in \text{ descendants}(u) \text{ dans } G\}$.

Exemple 30. Voici à droite la fermeture transitive du graphe montré à gauche :



Une fois ce graphe construit, il est possible de répondre rapidement à toutes les requêtes de la forme "existe-t-il un chemin de u à v dans G?" puisqu'il nous suffit de vérifier si l'arc

correspondant existe dans la fermeture transitive, ce que l'on peut faire en $O(1)$ si elle est stockée à l'aide d'une matrice d'adjacence.

Une idée simple permettant de calculer la fermeture transitive d'un graphe est de rajouter un arc connectant chaque sommet à tous ses descendants. Pour calculer ces descendants, on pourrait utiliser un parcours en profondeur ou en largeur ; en pratique, il semble plus efficace d'utiliser un parcours en largeur, car à mesure que l'on rajoute des arcs dans la fermeture transitive, de plus en plus de sommets deviennent accessibles plus rapidement et la recherche des descendants s'arrêtera donc plus tôt qu'avec un parcours en profondeur.

L'algorithme 21 implémente cette approche, en se basant sur l'algorithme 19 du parcours en largeur. On pourrait évidemment accélérer le calcul des descendants en se débarrassant de l'étape de tri, qui est superflue.

Algorithme 21 : FERMETURETRANSITIVE(G)

Entrées : un graphe orienté connexe G.
Sortie : la fermeture transitive de G.

```
1 F ← GrapheOrienté(G.sommets());
2 pour chaque u ∈ G.sommets() faire
3 |   pour chaque v ∈ PARCOURSLARGEURITÉRATIFORIENTÉ(F, u) faire
4 |   |   si u ≠ v alors F.ajouter_arc(u, v) ;
5 renvoyer F;
```

La complexité de cet algorithme se calcule simplement : on lance $|V|$ parcours, lesquels requièrent chacun $O(|V| + |A|)$ opérations, et l'on a donc un algorithme en $O(|V|^2 + |A||V|)$ opérations.

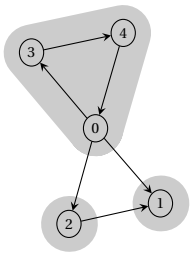
4.6 Composantes fortement connexes

Rappelons que les composantes connexes d'un graphe non orienté sont les ensembles maximaux de sommets reliés deux à deux par un chemin. Un concept similaire existe dans le cas des graphes orientés, dans lequel on prend naturellement en compte l'orientation des arcs :

Définition 21. Une *composante fortement connexe* H d'un graphe orienté G est un ensemble maximal de sommets dont toute paire est reliée par un chemin orienté. H est dit *faiblement connexe* s'il s'agit d'une composante connexe de la version non-orientée de G, mais pas d'une composante fortement connexe de G.

À cause de l'orientation, il n'est donc plus correct de dire qu'un graphe orienté fortement connexe est forcément "en un seul morceau".

Exemple 31. Voici les composantes fortement connexes du graphe de l'exemple 29 :



Il existe bien un chemin de 2 vers 1, mais pas de 1 vers 2 ; c'est pourquoi ces deux sommets n'appartiennent pas à la même composante fortement connexe.

(fin exemple 31)

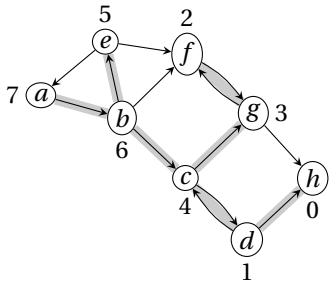
Dès lors, comment faire pour identifier les composantes fortement connexes d'un graphe orienté ? Une approche naïve pourrait consister à calculer la fermeture transitive du graphe, et à regrouper les paires de sommets du résultat qui sont reliés par un arc dans les deux sens. Cette approche fonctionne mais est inefficace, car elle nécessite $|V|$ parcours ; au lieu de cela, il est possible de s'en tirer avec seulement deux parcours, comme le fait l'algorithme de Kosaraju-Sharir [5] que l'on va présenter ci-dessous.

4.6.1 Parcours en profondeur "daté"

Deux sommets u et v appartiennent à la même composante fortement connexe si et seulement si il existe un chemin de u vers v **et** un chemin de v vers u. Pour déterminer quels sommets on peut atteindre à partir de u, il va bien falloir explorer le graphe en entier au moins une fois. Mais en récupérant des informations supplémentaires lors de ce premier parcours, on va réussir à éviter de devoir relancer un parcours à partir de chaque autre sommet pour vérifier si un chemin vers u existe également à partir de chacun de ses descendants.

La modification que l'on apporte au parcours en profondeur consiste à "dater" les sommets en fonction du moment où leur exploration se termine ; on considère l'exploration d'un sommet terminée quand tous ses descendants ont été visités.

Exemple 32. Voici comment le parcours en profondeur datera les sommets d'un graphe tiré de Cormen et al. [1], en supposant comme d'habitude que l'on lance les explorations dans l'ordre lexicographique. L'arbre de parcours correspondant est également montré.



On modifie légèrement l'algorithme 20 pour qu'il renvoie les dates de fin de visite de chaque sommet du graphe plutôt que les sommets ; on se passe des couleurs car on n'en a pas besoin dans ce cas-ci. Le résultat est montré à l'algorithme 22 et à l'algorithme 23.

Les dates obtenues nous donnent des informations sur l'accessibilité ; en effet, comme le parcours se fait en profondeur, le traitement des descendants de chaque sommet se termine

Algorithme 22 : PROFONDEURDATES(G)

Entrées : un graphe orienté G.
Sortie : les dates de fin de parcours en profondeur de chaque sommet du graphe.

```
1 dates ← tableau(G.nombre_sommets(), −1);
2 instant ← 0;
3 pour chaque v ∈ G.sommets() faire
4 |   si dates[v] = −1 alors
5 |   |   PARCOURSPROFONDEURORIENTÉDATESREC(G, v, dates, instant);
6 renvoyer dates;
```

Algorithme 23 : PARCOURSPROFONDEURORIENTÉDATESREC(G, départ, dates, instant)

Entrées : un graphe orienté G, un sommet de départ, un tableau de dates, et un instant.
Résultat : dates contient les dates de fin de visite des sommets de G accessibles à partir du sommet de départ dans l'ordre où le parcours en profondeur les a découverts.

```
1 dates[départ] ← 0;          // marquer le début de l'exploration;
2 pour chaque v ∈ G.successeurs(départ) faire
3 |   si dates[v] = −1 alors
4 |   |   PARCOURSPROFONDEURORIENTÉDATESREC(G, v, dates, instant);
5 dates[départ] ← instant;        // marquer la fin de l'exploration;
6 instant ← instant + 1;
```

avant le traitement de ce sommet, et la numérotation attribuée garantit donc la propriété suivante :

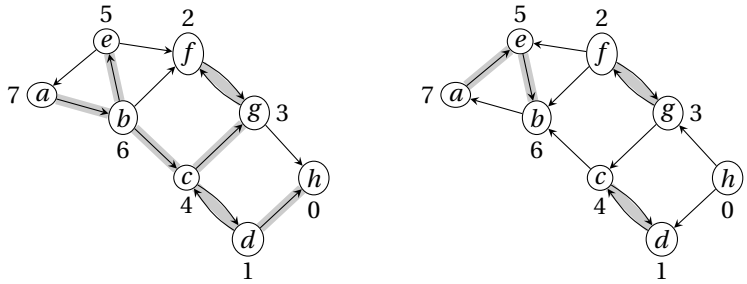
$$\forall u \in V(G) : \quad \text{date\_fin}(u) = \max_{v \in \text{ descendants}(u)} \text{date\_fin}(v) + 1.$$

Autrement dit : si v est un descendant de u, alors $\text{date\_fin}(v) < \text{date\_fin}(u)$. La réciproque est fausse, comme on peut déjà le constater sur un graphe contenant deux sommets mais pas d'arc.

4.6.2 Parcours du graphe renversé

La deuxième étape de l'algorithme de Kosaraju-Sharir consiste à parcourir le graphe **renversé**, c'est-à-dire une copie du graphe de départ $G = (V, A)$ dans lequel on suit les arcs à l'envers. Autrement dit, on parcourt le graphe $G' = (V, A')$ où $A' = \{(v, u) \mid (u, v) \in A\}$. Pour choisir les points de départ de l'exploration de ce graphe, on utilise les dates récupérées lors du premier parcours : les explorations seront lancées par date décroissante de fin de parcours.

Exemple 33. Reprenons le résultat de l'exemple 32 (à gauche). On obtient facilement le renversement de ce graphe, ainsi qu'une forêt de parcours pour ce graphe renversé en parcourant ses sommets par date de fin décroissante (à droite) :



D'où sort ce renversement de graphe, et quel rapport son parcours dans l'ordre préconisé peut-il avoir avec ce qui nous intéresse ? Les éléments suivants nous apportent des intuitions (qu'il nous faudra cependant prouver plus loin) :

1. un chemin existe du sommet u vers le sommet v dans le graphe original G si et seulement si un chemin existe de v vers u dans le graphe renversé G'. Ainsi, les composantes fortement connexes de G sont les mêmes que celles de G'.
2. comme on a déjà identifié que si v était accessible à partir de u, alors $\text{date\_fin}(u) > \text{date\_fin}(v)$, l'exploration du sommet x de date maximale nous donnera tous les sommets accessibles à partir de x dans G', ce qui par la relation ci-dessus correspond à l'ensemble de tous les sommets à partir desquels x est accessible dans G.

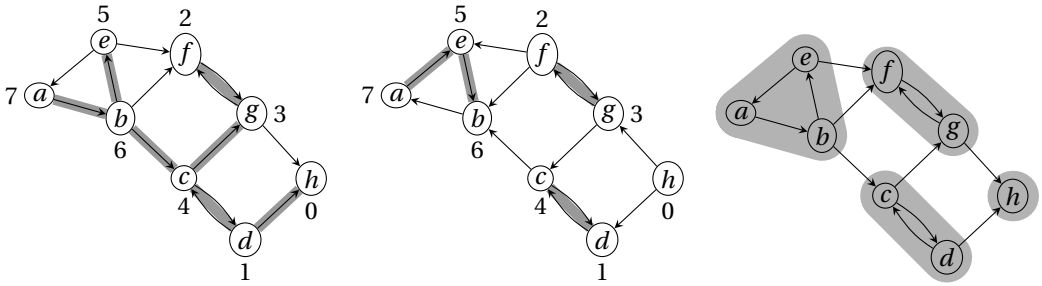
4.6.3 L'algorithme de Kosaraju-Sharir

Maintenant qu'on a exposé les ingrédients de l'algorithme, on peut résumer son approche comme suit :

1. on explore le graphe G en profondeur, en notant les dates de fin de parcours des sommets ;

2. on explore le graphe renversé en profondeur, en traitant les sommets par date décroissante de fin de parcours ;
3. les ensembles de sommets identifiés par le second parcours sont les composantes fortement connexes de G.

Exemple 34. Reprenons le graphe de l'exemple 32 pour illustrer le déroulement de l'algorithme de Kosaraju-Sharir. En premier lieu, (a) on calcule les dates de fin de traitement lors d'une exploration en profondeur à partir du sommet a ; l'arbre de parcours correspondant est illustré en gris pour aider à la compréhension, mais n'est pas utilisé dans l'algorithme. Ensuite, (b) on explore le graphe renversé par ordre décroissant des dates de fin d'exploration : les points de départ sont donc a, puis c, puis g, et enfin h. Dans la dernière étape (c), on a identifié les composantes fortement connexes.



(a) (b) (c)

L'algorithme 24 montre l'implémentation de cette approche, dans laquelle on suppose l'existence d'une fonction renversant le graphe.

Algorithme 24 : KOSARAJUSHARIR(G)

Entrées : un graphe orienté G.
Sortie : les composantes fortement connexes de G.

```
1 CFC ← liste();
2 dates ← PROFONDEURDATES(G);
  /* parcours du graphe renversé par date de fin décroissante      */
3 G' ← renverser_arcs(G);
4 déjà_visités ← tableau(G.nombre_sommets(), FAUX);
5 pour chaque v ∈ renverser(trier_sommets_par_date(G'.sommets(), dates)) faire
6     si ¬ déjà_visités[v] alors
7         CFC.ajouter_en_fin(PARCOURSPROFONDEURORIENTÉREC(G', v,
            déjà_visités));
8 renvoyer CFC;
```

Correction

Les trois étapes de l'algorithme 24 sont conceptuellement simples, mais la raison pour laquelle elles fonctionnent n'est pas nécessairement évidente. Tentons de les justifier intuitivement avant de prouver formellement que l'algorithme est correct.

Lors du premier parcours, on identifie les sommets que l'on peut atteindre à partir des racines de nos explorations. Le parcours effectué nous donne pour chaque sommet u l'en-

semble $D(u)$ de ses descendants, qui contient forcément la composante connexe de u mais ne coïncide pas forcément avec elle puisque l'existence d'un chemin de u vers $v \in D(u)$ n'implique pas celle d'un chemin de v vers u.

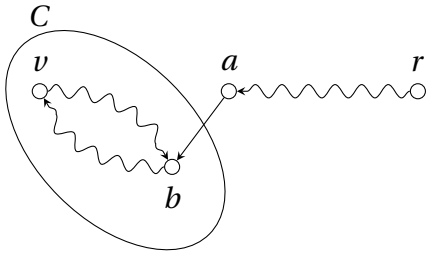
C'est pourquoi, lors du second parcours, on veut détecter s'il existe un chemin *vers u* à partir de chacun de ses descendants ; en parcourant le graphe renversé au départ d'un sommet x, on obtient l'ensemble $D(x)$ de ses descendants dans G', qui correspond à l'ensemble des sommets *à partir desquels il existe un chemin vers x dans le graphe original G* !

Toutefois, ce second parcours ne peut pas se faire dans un ordre arbitraire : si l'on atteint dans G', au départ de x, un sommet y avec $\text{date\_fin}(x) < \text{date\_fin}(y)$, alors on n'a obtenu aucune information nouvelle par rapport au premier parcours. C'est pourquoi on traite dans cette deuxième phase les sommets par ordre décroissant de fin de parcours : l'exploration des descendants s'est forcément terminée avant celle de leurs ancêtres, et on veut maintenant savoir quels sommets parmi ceux de date inférieure au sommet actuel nous permettent de l'atteindre dans le graphe d'origine. Les sommets identifiés dans la deuxième phase sont donc tous ceux qui sont à la fois accessibles à partir de u (cf. phase 1) et à partir desquels il existe un chemin vers u, ce qui correspond bien à une composante fortement connexe.

Le résultat suivant prouve que la stratégie appliquée dans la deuxième phase nous donne le résultat voulu.

Lemme 4.6.1. [2] Soit v le sommet de G dont la date de fin d'exploration est maximal. Alors la composante fortement connexe C de G contenant v est une source, c'est-à-dire qu'il n'existe pas d'arc (a, b) avec $a \notin C$ et $b \in C$.

Démonstration. Par contradiction, supposons qu'il existe un arc (a, b) avec $a \notin C$ et $b \in C$. Soit r le sommet à partir duquel l'exploration en profondeur mène au sommet a ; ce sommet r ne peut appartenir à C, sinon a appartiendrait aussi à C, et en particulier, on a donc $r \neq v$.



Les chemins $v \rightsquigarrow b$ et $b \rightsquigarrow v$ existent par définition de C. La fin du traitement de r précède celle de v, puisque la date de fin de traitement de v est maximale. Mais lors du début de traitement de r, il existe par définition de r un chemin de r vers a, et vers b via l'arc (a, b), et vers v puisque b appartient à C. Dès lors, la date de fin de traitement de v précède celle de r, ce qui est impossible puisque la date de fin de traitement de v est maximale. □

On peut maintenant prouver que l'algorithme de Kosaraju-Sharir est correct.

Proposition 4.6.1. L'algorithme 24 renvoie bien les composantes fortement connexes du graphe G sur lequel il agit.

Démonstration. Le Lemme 4.6.1 garantit que lors de l'exploration de G' à partir du sommet traité en dernier dans le premier parcours, on reconstitue bien la composante fortement

connexe contenant ce sommet “sans déborder” puisqu'il n'existe pas d'arc sortant de cette composante. La composante fortement connexe de G contenant ce sommet est donc identifiée correctement, et relancer l'exploration sur le prochain sommet non encore visité de date de fin de visite maximale est équivalent à relancer l'algorithme sur G' dont on aurait retiré la première composante fortement connexe identifiée. Par applications répétées du Lemme 4.6.1, l'algorithme est donc correct. □

Complexité

En implémentant le graphe à l'aide d'une liste d'adjacence, les deux parcours du graphe et le calcul du graphe renversé coûtent $O(|V|+|A|)$, mais on trie également les sommets dans la version présentée ici ; la complexité est donc de $O(|V|\log|V|+|A|)$. Remarquons qu'il n'est pas indispensable de disposer explicitement des dates de fin de visite : elles servent simplement à expliquer l'algorithme plus clairement. On aurait pu se contenter, lors du premier parcours, d'empiler les sommets à la fin de leur traitement, et de les dépiler lors du second parcours tout en vérifiant si chaque sommet dépilé a déjà été traité. Ainsi, on évite l'étape de tri, et on obtient donc une complexité en $O(|V|+|A|)$ comme pour un simple parcours.

Exercice 9. Donnez un algorithme améliorant l'algorithme 24 comme expliqué ci-dessus.

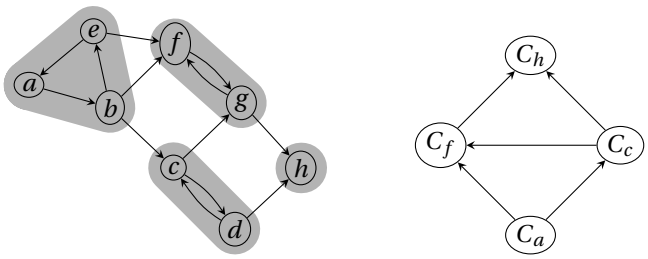
Mentionnons que Tarjan [6] a proposé un algorithme de même complexité mais plus efficace, dans la mesure où il n'effectue qu'un seul parcours.

4.6.4 Le graphe des composantes fortement connexes

Quel que soit l'algorithme utilisé, les composantes fortement connexes identifiées seront les mêmes et nous permettent d'obtenir une représentation plus compacte du graphe de départ :

Définition 22. Soit G un graphe orienté. Le *graphe des composantes fortement connexes* de G est le graphe orienté H défini par :
- $V(H)$ est l'ensemble $\{C_1, C_2, \ldots, C_p\}$ des composantes fortement connexes de G ;
- $A(H) = \{(C_i, C_j) \mid \exists u \in C_i, v \in C_j : (u,v) \in A(G)\}$.

Exemple 35. Reprenons (à gauche) le graphe dont on a identifié les composantes fortement connexes, et identifions les composantes connexes par le plus petit sommet qu'elles contiennent. Le graphe des composantes fortement connexes correspondant est montré à droite :



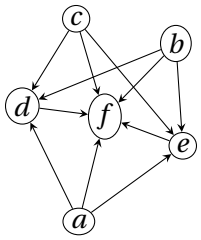
4.7 Graphes orientés acycliques

Dans le cadre des graphes non-orientés, certains problèmes sont (algorithmiquement) faciles à résoudre sur des arbres, mais deviennent difficiles dès que des cycles apparaissent. Les graphes orientés présentent un degré de finesse supplémentaire : certains problèmes restent faciles à résoudre tant que le graphe ne contient pas de **cycle orienté**, ce qui n'en fait pas pour autant un arbre puisqu'il peut très bien contenir plusieurs chemins entre deux paires de sommets. On qualifie ces graphes de *graphes orientés acycliques*, ou plus traditionnellement de DAGs (pour ***d**irected **a**cyclic **g**raphs*).

Dans plusieurs applications, les sommets du graphe orienté que l'on manipule peuvent être vus comme des tâches à réaliser, et un arc (u, v) relie une tâche v dont la réalisation nécessite que la tâche u ait été menée à bien. C'est le cas par exemple du graphe de dépendances des diverses sources d'un projet informatique, ou de celui du planning des travaux de construction (on ne peut pas par exemple bâtir les murs avant que les fondations aient été finalisées), ou encore de l'étude de divers cours qui nécessitent la maîtrise de certains prérequis. Pour ces applications, il est impératif de savoir dans quel ordre exécuter ces tâches.

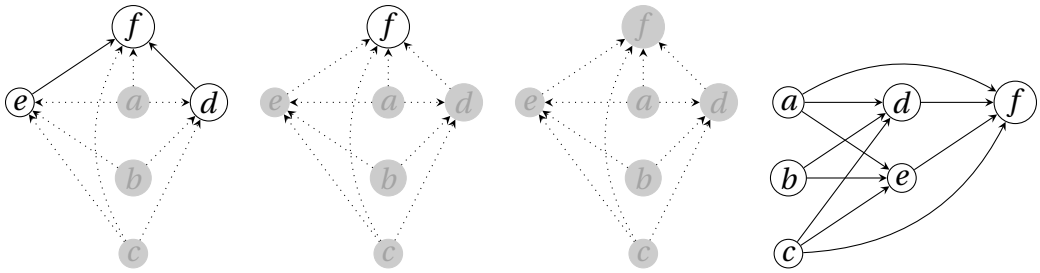
Définition 23. Un *ordre topologique* pour un graphe orienté acyclique G est un ordonnancement L de ses sommets dans lequel tout sommet apparaît après ses prédécesseurs.

Exemple 36. Voici un DAG pour lequel l'ordre (a, b, c, d, e, f) est un ordre topologique :



Un algorithme simple et intuitif dû à Kahn [4] nous permet de produire un ordre topologique selon le principe suivant : tout DAG possède au moins une *source*, c'est-à-dire un sommet de degré entrant nul. On peut donc placer ces sources en premier dans notre ordre, et supprimer les arcs qui en sortent, ce qui fait chuter de 1 le degré entrant de leurs successeurs. On réitère le processus en plaçant à la fin de notre résultat les nouvelles sources ainsi créées, en supprimant les arcs du graphe jusqu'à ce que tout ait été traité.

Exemple 37. Voici le déroulement de l'algorithme de Kahn sur le graphe de l'exemple 36. À chaque étape, l'identification des sources nous conduit à simuler la suppression des arêtes qui en sont issues (en pointillés). Le résultat final nous permet de dessiner le graphe de manière à ce que tous les prédécesseurs de chaque sommet apparaissent avant lui, en lisant le graphe de gauche à droite.



(fin exemple 37)

L'algorithme 25 montre une implémentation de ce tri, où l'on stocke des degrés que l'on fait diminuer plutôt que de modifier le graphe.

Algorithme 25 : KAHN(G)

Entrées : un graphe orienté acyclique G.
Sortie : les sommets de G ordonnés topologiquement.

```
   /* stocker les degrés entrants et les sources                    */
1  résultat ← liste();
2  sources ← pile();
3  degrés_entrants ← tableau(G.nombre_sommets(), 0);
4  pour chaque v ∈ G.sommets() faire
5  |  degrés_entrants[v] ← G.degré_entrant(v);
6  |  si degrés_entrants[v] = 0 alors sources.empiler(v);
   /* dépiler les sources, les ajouter au résultat, et empiler les
      nouvelles sources                                             */
7  tant que sources.pas_vide() faire
8  |  u ← sources.dépiler();
9  |  résultat.ajouter_en_fin(u);
10 |  pour chaque v ∈ G.successeurs(u) faire
11 |  |  degrés_entrants[v] ← degrés_entrants[v] −1;
12 |  |  si degrés_entrants[v] = 0 alors sources.empiler(v);
13 renvoyer résultat;
```

Exercice 10. L'algorithme 25 suppose que le graphe fourni en entrée est acyclique. Comment peut-on le modifier ou l'utiliser pour déterminer si un graphe donné contient un cycle (sans reconstruire ce cycle explicitement) ?

Complexité

La complexité de l'algorithme de Kahn dépend de l'implémentation du graphe :

1. si l'on choisit une matrice d'adjacence, on passe $|V|$ fois dans une boucle où l'on fait appel à la méthode G.successeurs(u) qui est en $O(|V|)$ pour les mêmes raisons que la méthode voisins() des graphes non orientés. Dès lors, la complexité de l'algorithme est $O(|V|^2)$.
2. si l'on choisit une liste d'adjacence, on passe $|V|$ fois dans une boucle où l'on fait appel à la méthode G.successeurs(u) qui est en $O(deg^+(u))$ pour les mêmes raisons que

la méthode voisins() des graphes non orientés. Dès lors, la complexité de l'algorithme est $O(|V|+|A|)$.

4.8 Graphes orientés en graphviz

Peu de différences avec le cas non-orienté : on écrit `digraph`[2] au lieu de `graph`, et là où on renseignait l'arête $\{u, v\}$ par la notation `u -- v`, on renseignera l'arc (u, v) par la notation `u -> v`. On ne peut pas mélanger les arêtes et les arcs dans un même graphe, car Graphviz ne gère pas de situation intermédiaire entre les graphes orientés et ceux qui ne le sont pas, mais on peut contourner cette limitation dans le dessin de deux manières :

— `u -> v [dir=none]` ; supprime les flèches, et l'on visualise donc l'arc (u, v) comme une arête ;
— `u -> v [dir=both]` ; affiche un lien contenant des flèches aux deux extrémités, ce qui donne un dessin plus compact que l'alternative `u -> v; v -> u;`.

Bibliographie

[1] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, ET C. STEIN, *Introduction to Algorithms*, MIT Press, 3ème édition, 2009.

[2] S. EVEN, *Graph Algorithms*, Cambridge University Press, 2ème édition, 2012.

[3] M. R. GAREY ET D. S. JOHNSON, *Computers and Intractability : A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.

[4] A. B. KAHN, *Topological sorting of large networks*, Communications of the ACM, 5 (1962), pages 558–562.

[5] M. SHARIR, *A strong-connectivity algorithm and its applications in data flow analysis*, Computers & Mathematics with Applications, 7 (1981), pages 67–72.

[6] R. E. TARJAN, *Depth-first search and linear graph algorithms*, SIAM Journal on Computing, 1 (1972), pages 146–160.

2. Pour ***di****rected* ***graph***.