Chapitre 2

Parcours de graphes

Sommaire

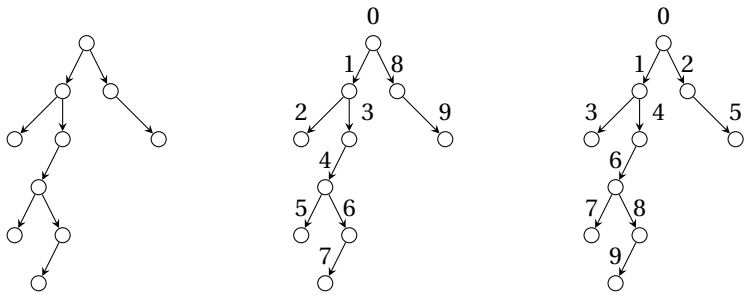
Une des tâches les plus basiques qu'on peut chercher à accomplir sur une structure de données est d'afficher son contenu, ce qui implique de pouvoir la parcourir. Les graphes se parcourent en général de deux manières différentes : en *profondeur* ou en *largeur*. Ce chapitre couvre ces algorithmes ainsi que des variantes qui seront utiles dans diverses applications. En guise d'échauffement, on commencera par réexaminer les parcours d'arbres.

2.1 Rappels : parcours d'arbres, piles et files

Les *arbres* sont des graphes sans cycle contenant un chemin unique entre toute paire de sommets. Pour faire le lien avec le cours d'algorithmique des arbres, focalisons-nous dans un premier temps sur les arbres binaires enracinés ; il est utile de les parcourir d'au moins deux façons :

1. en *profondeur*, c'est-à-dire qu'on explore d'abord la racine, puis tout le sous-arbre gauche, et enfin tout le sous-arbre droit récursivement ; ou
2. en *largeur*, c'est-à-dire qu'on explore d'abord la racine, puis le fils gauche, puis le fils droit, et qu'on répète ces actions récursivement sur les sous-arbres.

Exemple 9. Voici (*a*) un arbre binaire, et les numérotations des sommets que l'on peut obtenir en le parcourant (*b*) en profondeur ou (*c*) en largeur :



(*a*) (*b*) (*c*)

On aura besoin dans la suite de manipuler des *piles*, c'est-à-dire une structure de données de type LIFO [1], et des *files*, c'est-à-dire une structure de données de type FIFO [2]. Ces structures bien connues ont déjà été rencontrées dans des cours précédents ; on se contentera donc ici de rappeler les méthodes qu'on s'attend à trouver dans les classes qui les implémentent ces structures avec les complexités attendues.

Les méthodes suivantes sont supposées disponibles pour une pile et s'exécutent en $O(1)$:

— pile.empiler(x), qui rajoute x au sommet de la pile ;
— pile.dépiler(), qui extrait et renvoie le haut de la pile ;
— et les méthodes pile.est_vide() et pile.pas_vide(), dont les noms sont éloquents et qui renvoient VRAI ou FAUX.

Pour rendre le pseudocode plus concis, on supposera que la méthode empiler() peut également prendre une séquence de t éléments en paramètre — sa complexité sera alors en $O(t)$.

Les méthodes suivantes sont supposées disponibles pour une file et s'exécutent en $O(1)$:

— enfiler(x), qui rajoute x à la fin de la file ;
— défiler(), qui extrait et renvoie le début de la file ;
— et les méthodes est_vide() et pas_vide(), dont les noms sont éloquents et qui renvoient VRAI ou FAUX.

2.1.1 Parcours d'arbres en profondeur

Dans le parcours en profondeur, on identifie les (sous-)arbres et leur racine, et la fonction à écrire ne nécessite donc pas d'autre paramètre que l'arbre. L'algorithme 2 montre comment on procèderait de manière récursive pour parcourir un arbre binaire enraciné en profondeur.

Exercice 3. Écrivez une version itérative de l'algorithme 2.

1. Pour **L**ast **I**n, **F**irst **O**ut.
2. Pour **F**irst **I**n, **F**irst **O**ut.

Algorithme 2 : PARCOURSPROFONDEURARBRE(A)

Entrées : un arbre binaire enraciné A.
Résultat : l'affichage des sommets de A suivant un parcours en profondeur à partir de la racine.

```
1 si A.racine() ≠ NIL alors
2     afficher(A.racine());
3     PARCOURSPROFONDEURARBRE(A.sous_arbre_gauche());
4     PARCOURSPROFONDEURARBRE(A.sous_arbre_droit());
```

2.1.2 Parcours d'arbres en largeur

Le parcours d'un arbre en largeur s'effectue à l'aide d'une file. En partant de la racine de l'arbre, on insère tous les descendants du sommet actuel dans la file, et on répète l'opération sur chaque sommet que l'on extrait de la file jusqu'à ce qu'elle soit vide. L'algorithme 3 fonctionne pour un arbre enraciné de degré arbitraire et renvoie ses sommets dans l'ordre dans lequel ils ont été explorés au lieu de les afficher.

Algorithme 3 : PARCOURSLARGEURARBRE(A)

Entrées : un arbre enraciné A.
Sortie : la liste des sommets de l'arbre ordonné selon un parcours en largeur à partir de la racine.

```
1 a_traiter ← file();
2 résultat ← liste();
3 a_traiter.enfiler(A.racine());
4 tant que a_traiter.pas_vide() faire
5     sommet ← a_traiter.défiler();
6     résultat.ajouter_en_fin(sommet);
7     pour chaque descendant dans A.successeurs(sommet) faire
8         a_traiter.enfiler(descendant);
9 renvoyer résultat;
```

2.1.3 Complexité et correction

Il est assez simple de se convaincre que ces algorithmes se terminent et fournissent bien les sommets dans l'ordre où on s'attend à les obtenir. Les complexités de ces deux algorithmes sont les mêmes : on passe exactement une fois sur chacun des sommets, et on emprunte exactement une fois chaque arête pour s'y rendre ; cela nous donne donc du $O(|V|+|E|)$, qui se simplifie en $O(|V|)$ puisque tout arbre sur $|V|$ sommets contient exactement $|V|-1$ arêtes [3].

2.2 Parcours de graphes

Remarquons que ces arbres sont des graphes très particuliers :

3. La preuve de cette affirmation est laissée en exercice.

1. ils sont *enracinés* : le point de départ de l'exploration est donc unique ;
2. ils ne possèdent pas de cycle : il n'est donc possible d'aller de la racine à un sommet arbitraire v que par un seul chemin ;
3. et de même, lorsqu'on arrive sur un sommet en partant de la racine, il n'y a qu'un seul sens d'exploration possible ("vers le bas").

Il va donc falloir s'affranchir de ces hypothèses pour généraliser les algorithmes de parcours aux graphes. En effet :

1. les algorithmes que nous écrirons devront permettre d'explorer un graphe au départ de n'importe quel sommet, puisque nos graphes ne sont pas enracinés ;
2. comme plusieurs chemins différents peuvent exister entre deux sommets, il est possible dans un parcours de retomber sur un sommet que l'on a déjà exploré ; il nous faudra donc enregistrer les sommets déjà traités afin de pouvoir les ignorer, sans quoi l'exploration ne se terminera pas – on peut déjà constater ce problème sur le graphe K_2 (une simple arête).
3. enfin, comme chaque sommet peut servir de point de départ et possède un nombre arbitraire de voisins, il faudra aussi définir un ordre d'exploration sur ces voisins. On fera l'hypothèse suivante :

En cas d'ambigüité, les algorithmes de parcours examinent toujours les sommets dans l'ordre croissant suivant les identifiants.

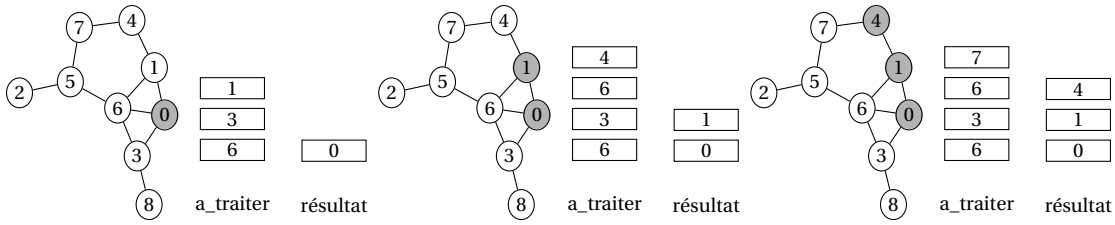
Nos algorithmes de parcours maintiendront un tableau de booléens initialisés à FAUX, qui indiqueront en position i si le sommet d'identifiant i a déjà été visité. On supposera à cet effet l'existence d'une fonction tableau(n, v) renvoyant un tableau de n cases toutes initialisées à la valeur v.

On supposera aussi, pour simplifier et s'éviter un appel à une fonction de tri, que la méthode G.voisins(sommet) nous renverra les voisins du sommet spécifié par identifiant croissant.

2.2.1 Parcours de graphes en profondeur

Le parcours d'un graphe en profondeur se réalise en partant d'un sommet arbitraire v à visiter, et en parcourant d'abord un de ses voisins u et tous ses "descendants" (c'est-à-dire tous les sommets accessibles à partir de u) avant de traiter le voisin suivant de v. Une fois ces descendants explorés, on applique le même traitement au prochain voisin de v non encore visité, et ainsi de suite jusqu'à ce que tous les sommets aient été couverts. On manipulera tout au long de notre parcours une liste résultat, enregistrant les identifiants des sommets au fur et à mesure qu'on les découvre, et une pile a_traiter, qui contiendra les prochains sommets à parcourir.

Exemple 10. Voici les trois premières étapes du parcours en profondeur sur un graphe au départ du sommet 0. Les sommets qu'on a visités sont marqués en gris (et se trouvent aussi dans le résultat) :



(fin exemple 10)

Le résultat est montré à l'algorithme 4 : la structure déjà_visités y est essentielle et nous permet de savoir quels sommets ont déjà été parcourus ; la structure a_traiter stocke les sommets dans l'ordre où il faut les traiter, et peut parfois contenir des sommets que l'on a déjà parcourus, ce que l'on vérifie à la ligne 7. Enfin, comme on décide de parcourir les sommets dans l'ordre croissant, on doit les empiler *dans l'ordre décroissant*, puisque la pile nous fournira les sommets qu'elle contient dans l'ordre inverse.

Algorithme 4 : PARCOURSPROFONDEURITÉRATIF(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.

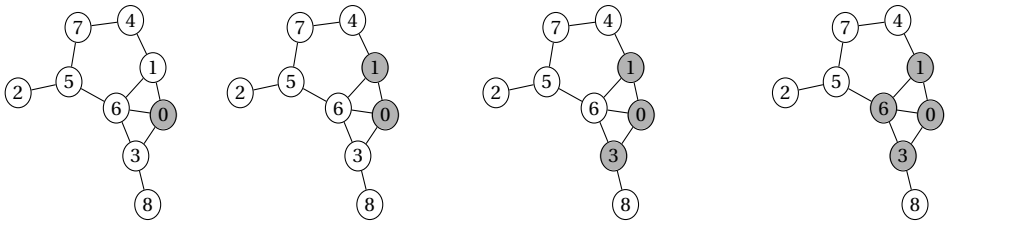
Sortie : les sommets de G accessibles depuis le départ dans l'ordre où le parcours en largeur les a découverts.

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

2.2.2 Parcours de graphes en largeur

Une fois le parcours en profondeur d'arbre généralisé au cas des graphes non orientés, la généralisation aux graphes du parcours d'arbre en largeur ne présente pas de difficulté majeure. On manipule toujours une liste résultat dans laquelle on insère les sommets au fur et à mesure qu'on les découvre, mais cette fois, la structure a_traiter est une file, et l'on traite tous les voisins d'un sommet de départ avant de passer aux voisins non visités de ces voisins, et ainsi de suite.

Exemple 11. Voici les quatre premières étapes du parcours en largeur sur le graphe de l'exemple 10 au départ du sommet 0 :



(fin exemple 11)

L'algorithme 5 montre le résultat; les seules différences avec l'algorithme 4 concernent le marquage des sommets visités.

Algorithme 5 : PARCOURSLARGEURITÉRATIF(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 voisin dans G.voisins(sommet) faire
11             si ¬ déjà_visités[voisin] alors a_traiter.enfiler(voisin) ;
12 renvoyer résultat;
```

2.2.3 Correction et complexité

Les algorithmes présentés plus haut parcourent bien tous les sommets du graphe s'ils sont tous accessibles à partir du point de départ; autrement dit, si le graphe vérifie la propriété suivante.

Définition 9. Un graphe $G = (V, E)$ est *connexe* si pour toute paire de sommets $u, v \in V$, il existe un chemin dans G dont les extrémités sont u et v.

Informellement, et plus visuellement, on dirait que le graphe est "en un seul morceau".

La complexité des algorithmes de parcours dépendra directement de la représentation que l'on aura choisie [4] :

— si l'on utilise une matrice d'adjacence, remarquons que chaque sommet est marqué comme ayant été visité exactement une fois, et que pour chaque sommet, on parcourt

4. Rappel : voir le Tableau 1.2 page 18 pour la comparaison des complexités des méthodes dans les deux implémentations traitées ici.

tous ses voisins ; l'appel à la méthode G.voisins(u) nous coûtant $O(|V|)$, on se retrouve avec une complexité de $O(|V|^2)$.
— si l'on implémente le graphe à l'aide d'une liste d'adjacence, les appels à la méthode G.voisins(u) s'exécutent en $O(\deg(u))$. On peut obtenir une analyse plus fine de la complexité en se rendant compte que l'on marque chaque sommet comme ayant été visité exactement une fois, et que l'on parcourt chaque arête $\{u, v\}$ **deux fois** (une fois de u à v en examinant $N_G(u)$ et une fois de v à u en examinant $N_G(v)$). Comme la liste d'adjacence ne stocke pour chaque sommet que ceux qui lui sont adjacents, la complexité sera donc $O(|V| + |E|)$.

En réalité, une implémentation efficace des algorithmes de parcours demande un peu plus de soin pour éviter ce que l'on pourrait qualifier de problème d'empilements multiples : comme on l'a vu dans l'exemple 10 et l'exemple 11, même si chaque sommet ne se retrouve qu'une seule fois dans le résultat, il pourrait se retrouver plusieurs fois dans la structure a_traiter. C'est pourquoi on retrouve fréquemment dans la littérature des variantes "tricolores" de ces algorithmes de parcours, qui partitionnent $V(G)$ en trois ensembles selon le statut d'un sommet : au lieu de différencier les sommets "non traités" des sommets "traités", on distingue une troisième catégorie "en cours de traitement", et l'on n'empile donc que les sommets qui n'ont réellement pas encore été rencontrés. Nous en reparlerons lorsque nous aborderons les graphes orientés, car cette distinction y sera essentielle.

2.2.4 Parcours "génériques"

En comparant l'algorithme 4 et l'algorithme 5, on se rend compte qu'ils sont extrêmement similaires : la seule différence est la structure dans laquelle on insère les sommets à traiter. On pourrait donc écrire un pseudocode encore plus générique, comme celui qui est montré à l'algorithme 6. La structure S utilisée ici est supposée vide, et l'on ne fait plus aucune hypothèse sur son type ; on exige simplement qu'elle possède :

1. une méthode S.extraire() retirant et renvoyant l'élément suivant, et
2. une méthode S.insérer(élément) nous permettant d'y ajouter des éléments.

L'ordre dans lequel les éléments seront extraits et ajoutés à la structure sera imposé par son type ; ainsi, on retrouvera le parcours en largeur avec ParcoursGénérique(G, file(), départ) et le parcours en profondeur avec ParcoursGénérique(G, pile(), départ) — enfin, "presque" : la version générique ne se préoccupe pas de l'ordre lexicographique qu'on s'est imposé.

2.3 Arbres de parcours

Les parcours en profondeur et en largeur donnent naturellement naissance à des arbres associés, appelés *arbres de parcours*. Une définition formelle étant un peu fastidieuse à écrire, nous nous contenterons de mentionner que les sommets d'un tel arbre sont ceux du graphe qui sont accessibles au départ du sommet choisi pour explorer le graphe, et que ses arêtes sont définies par l'ordre que l'algorithme de parcours a produit. Bien entendu, la structure de chacun de ces arbres dépend, en plus du type de parcours choisi, du sommet qu'on a sélectionné comme point de départ.

Algorithme 6 : PARCOURSGÉNÉRIQUE(G, S, départ, déjà_visités=NIL)

Entrées : un graphe non-orienté G, une structure de données S supposée vide et un sommet de départ.

Sortie : les sommets de G accessibles depuis le départ dans l'ordre imposé par la structure S.

```
1  résultat ← liste();
2  si déjà_visités = NIL alors déjà_visités ← tableau(G.nombre_sommets(), FAUX);
3  S.insérer(départ);
4  tant que S.pas_vide() faire
5      sommet ← S.extraire();
6      si ¬ déjà_visités[sommet] alors
7          résultat.ajouter_en_fin(sommet);
8          déjà_visités[sommet] ← VRAI;
9          pour chaque voisin dans G.voisins(sommet) faire
10             si ¬ déjà_visités[voisin] alors S.insérer(voisin) ;
11 renvoyer résultat;
```

Exemple 12. Voici les arbres de parcours en profondeur (gauche) et en largeur (droite) obtenus au départ du sommet 0 en suivant les parcours de l'exemple 10 et de l'exemple 11 jusqu'au bout :

Si le graphe est connexe, ces arbres couvrent tous les sommets du graphe et on les qualifie donc de *couvrants*. Disposer d'un tel arbre permet d'atteindre tous les sommets du graphe au départ de n'importe quel sommet; les autres arêtes du graphe deviendront donc superflues si tout ce qui nous intéresse est de pouvoir relier deux sommets. Mais ces arbres couvrants ne nous donnent pas toujours un chemin de longueur minimale entre deux sommets : il se pourrait en effet que deux sommets soient très éloignés dans un arbre couvrant alors qu'ils sont adjacents dans le graphe d'origine. Ces arbres se généralisent naturellement en *forêts*, c'est-à-dire en ensembles d'arbres, dans le cas où le graphe exploré n'est pas connexe.

2.4 Applications

Comme les parcours en largeur et en profondeur ont la même complexité, ce n'est pas ce critère qu'on utilisera en général pour préférer un algorithme à l'autre : ce sont les applications et les utilisations de ces parcours qui guideront notre choix, en particulier dans les cas où il n'est pas nécessaire d'explorer le graphe en entier. Un exemple assez simple est celui d'un GPS qui explorerait le graphe du réseau routier pour déceler les stations-service les plus proches de nous : dans ce cas-là, une exploration en largeur nous donnerait le résultat

voulu. Nous allons voir ci-dessous que de nombreux problèmes peuvent être résolus à l'aide d'algorithmes qui sont des variations assez simples sur le thème des parcours vus plus haut.

2.4.1 Connexité et composantes connexes

Une manière simple de vérifier si un graphe est connexe est de le parcourir à partir d'un sommet arbitraire, et de vérifier à la fin du parcours si tous les sommets ont été visités ; c'est ce que fait l'algorithme 7. Ici, l'ordre dans lequel les sommets sont visités importe peu : on choisit donc le parcours qu'on préfère, et l'ordre dans lequel on explore les voisins peut aussi être arbitraire.

Algorithme 7 : EstConnexe(G)

Entrées : un graphe non orienté G.
Sortie : VRAI si G est connexe, FAUX sinon.

```
1 si G.nombre_sommets() = 0 alors renvoyer VRAI; // on suppose le graphe vide
    connexe
2 départ ← sommet arbitraire de G;
3 renvoyer taille(ParcoursLargeurItératif(G, départ)) = G.nombre_sommets();
```

Si le graphe n'est pas connexe, alors il est en plusieurs "morceaux", définis comme suit :

Définition 10. Une *composante connexe* d'un graphe G est un sous-graphe connexe H de G qui est maximal, c'est-à-dire qu'il n'existe pas de sommet de G à la fois accessible à partir d'un élément de $V(H)$ et hors de $V(H)$.

L'aspect "maximal" de cette définition est important car il définit de manière unique les composantes connexes ; sans cela, on pourrait par exemple dire que chaque sommet est une composante connexe du graphe. Là encore, on peut identifier les composantes connexes d'un graphe en répétant ce que l'on fait dans l'algorithme 7. L'algorithme 8 montre le résultat : tant qu'il existe un sommet non visité dans le graphe, on parcourt le graphe à partir de ce sommet, et la composante connexe du graphe contenant ce sommet est l'ensemble de tous les sommets visités par ce parcours.

Algorithme 8 : ComposantesConnexes(G)

Entrées : un graphe non orienté G.
Sortie : les composantes connexes de G, identifiées par la liste de leurs sommets.

```
1 résultat ← liste();
2 déjà_visités ← tableau(G.nombre_sommets(), FAUX);
3 pour chaque sommet dans G.sommets() faire
4     si ¬ déjà_visités[sommet] alors
5         résultat.ajouter_en_fin(ParcoursLargeurItératif(G, sommet,
            déjà_visités))
6 renvoyer résultat;
```

2.4.2 Reconnaissance de graphes bipartis

Définition 11. Un graphe est *biparti* si l'on peut partitionner l'ensemble V de ses sommets en deux parties V_1 et V_2, de telle sorte que deux sommets appartenant à la même partie ne sont jamais adjacents.

Bien souvent, des problèmes algorithmiquement difficiles [5] sur les graphes deviennent plus simples à résoudre sur des graphes bipartis, et il est donc important de pouvoir les reconnaître.

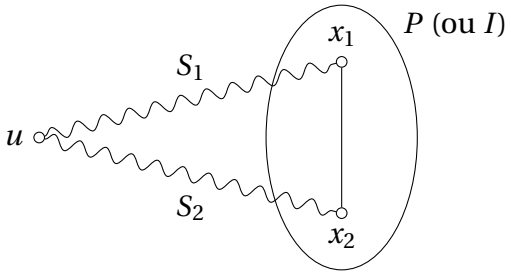
Comment procéder ? Une approche naïve consisterait à examiner les $2^{|V|-1}-1$ bipartitions possibles jusqu'à ce qu'on en trouve une satisfaisante ou qu'on puisse conclure qu'il n'en existe pas. Comme cela se produit souvent en algorithmique, cette approche naïve est correcte mais pas optimale (loin s'en faut dans ce cas), et l'on peut construire un algorithme beaucoup plus efficace en exploitant des propriétés structurelles des objets que l'on étudie. Le résultat suivant nous donne une caractérisation alternative des graphes bipartis qui nous permettra d'arriver à nos fins.

Théorème 2.4.1. Un graphe est biparti si et seulement s'il ne contient pas de cycle de longueur impaire.

Démonstration. On procède en deux temps :

- $\Rightarrow$: soit G un graphe biparti avec les classes V_1 et V_2 : alors tout cycle de ce graphe peut se parcourir en partant de V_1 et doit revenir au sommet de départ en visitant alternativement V_2 et V_1. Comme pour revenir dans V_1, on est obligé de faire des aller-retours qui sont par définition de longueur 2, chacun des cycles que l'on parcourt ainsi est pair.
- $\Leftarrow$: montrons que si G n'est pas biparti, alors il contient un cycle impair. On suppose par simplicité que G est connexe (si ce n'est pas le cas, il nous suffira d'appliquer le même raisonnement à chacune de ses composantes). Prenons n'importe quel sommet v, et construisons une partition $V = I \cup P$, où :
  - I est l'ensemble des sommets à distance **i**mpaire de v (la distance entre deux sommets étant la longueur du plus court chemin les reliant) ;
  - P est l'ensemble des sommets à distance **p**aire de v.

  Par construction, on a $v \in P$ et $I \cap P = \emptyset$. Puisque G n'est pas biparti, il existe deux sommets x_1 et x_2 appartenant à la même classe (soit P, soit I) qui sont adjacents, sinon (P, I) serait une bipartition satisfaisant la définition d'un graphe biparti ; comme G est connexe, il existe un chemin S_1 de x_1 à v et un chemin S_2 de v à x_2, et ces deux chemins sont de même parité par définition de x_1 et de x_2 :



5. C'est-à-dire NP-difficiles.

Dès lors, la somme des longueurs de ces deux chemins est paire, et le cycle induit par $E(S_1) \cup E(S_2) \cup \{\{x_1, x_2\}\}$ est donc de longueur impaire.

□

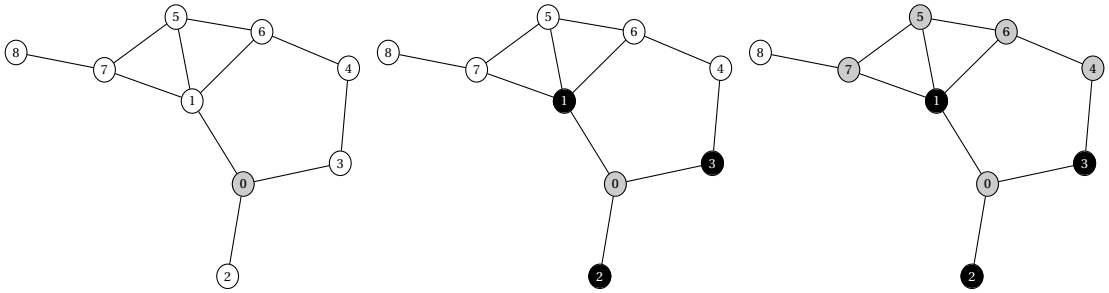
Un algorithme simple qui exploite le Théorème 2.4.1 consiste à parcourir le graphe en largeur au départ d'un sommet arbitraire, et à colorier les sommets ainsi explorés en alternant deux couleurs — par exemple : le sommet de départ est colorié en gris, ses voisins en noir, leurs voisins non encore visités en gris, et ainsi de suite. Lors du parcours, il se peut que l'on rencontre des sommets déjà visités :

— si l'on tombe sur un voisin déjà visité (et donc déjà colorié) auquel on tente d'affecter une couleur correspondant à celle qu'il a déjà, alors cela signifie qu'on a réussi à atteindre ce voisin par un autre chemin de la même parité que celui qu'on est en train de construire ; dès lors, si l'union de ces deux chemins forme un cycle, il est forcément pair et ce n'est donc pas un problème.

— par contre, si l'on essaie d'affecter à un voisin déjà visité une couleur différente de sa couleur, cela signifie qu'on a réussi à l'atteindre par un chemin de parité différente du chemin qu'on est en train de construire ; et l'union de ces deux chemins forme donc un cycle de longueur impaire.

Ainsi, si l'on s'arrête à cause d'un conflit de couleur, on peut renvoyer FAUX puisque le graphe ne peut pas être biparti. Sinon, on renvoie VRAI.

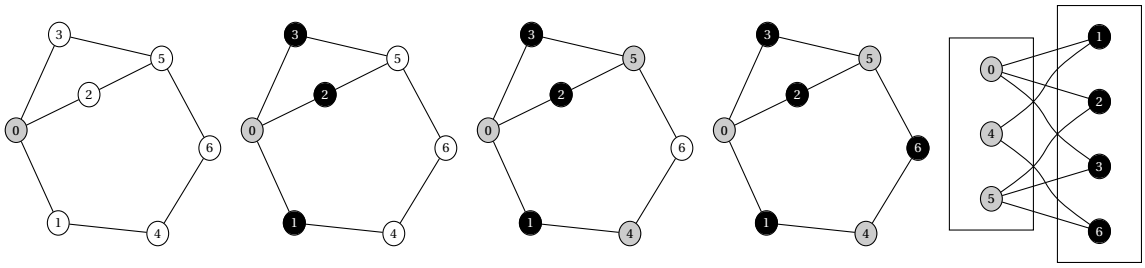
Exemple 13. Dans l'exemple suivant, on détecte que le graphe fourni en entrée n'est pas biparti, puisque le coloriage partiel nous forcerait après l'étape (3) à affecter des couleurs différentes à un même sommet.



(1) (2) (3)

On peut donc aussi en conclure que si deux sommets adjacents ont la même couleur, alors le graphe ne pourra pas être biparti.

Exemple 14. Le graphe ci-dessous est bien biparti : la dernière étape (5) en montre une présentation.



(1) (2) (3) (4) (5)

(fin exemple 14)

L'algorithme 9 implémente cette approche et vérifie si un graphe donné est biparti. On suppose que le graphe donné est connexe, mais il n'est pas difficile d'adapter l'algorithme au cas où le graphe n'est pas connexe. Il n'est pas difficile de modifier l'algorithme pour renvoyer à la fin du parcours les deux classes de couleurs que l'on a identifiées, qui forment forcément une bipartition valide au sens de la définition 11, et qui nous permettent ainsi de construire une représentation bipartie du graphe.

Exercice 4. Pourquoi l'algorithme 9 peut-il donner des résultats incorrects sur un graphe non connexe ?

Algorithme 9 : ESTBIPARTI(G)

Entrées : un graphe connexe G.
Sortie : VRAI si G est biparti, FAUX sinon.

```
1  si G.nombre_sommets() = 0 alors renvoyer VRAI;
2  couleurs ← tableau(G.nombre_sommets(), −1);
3  couleur_actuelle ← VRAI;
4  départ ← sommet arbitraire de G;
5  a_traiter ← file();
6  a_traiter.enfiler(départ);
7  couleurs[départ] ← couleur_actuelle;
8  tant que a_traiter.pas_vide() faire
9      sommet ← a_traiter.défiler();
10     couleur_actuelle ← ¬ couleurs[sommet];
11     pour chaque voisin dans G.voisins(sommet) faire
12         si couleurs[voisin] = −1 alors
13             couleurs[voisin] ← couleur_actuelle;
14             a_traiter.enfiler(voisin);
15         sinon si couleurs[voisin] ≠ couleur_actuelle alors renvoyer FAUX;
16 renvoyer VRAI;
```

Notons que la notion de graphe biparti se généralise naturellement aux graphes k-*partis*, dont les sommets se partitionnent en k classes *indépendantes* (c'est-à-dire que deux sommets de la même classe ne sont jamais adjacents). Malheureusement, les idées utilisées pour reconnaître les graphes bipartis ne se généralisent pas aux graphes k-partis : reconnaître les graphes tripartis (ou 3-partis) est un problème NP-complet [1].

2.4.3 Détection de cycles

Le Théorème 2.4.1 et l'algorithme 9 nous permettent de déterminer si un graphe donné contient un cycle impair. Mais comment savoir si un graphe contient un cycle de parité ou de longueur quelconque ?

Déterminer si un graphe contient un cycle est assez simple si la réponse demandée est "oui" ou "non" (donc si on ne demande pas de fournir un cycle explicitement) : un arbre étant un graphe connexe $G = (V, E)$ tel que $|E| = |V| - 1$, on peut s'en sortir avec l'approche suivante :

1. si G est connexe, alors il n'est acyclique que si c'est un arbre, et on renvoie donc la valeur du test G.nombre_aretes() $\geq$ G.nombre_sommets() ;
2. sinon, il nous suffit d'appliquer le même raisonnement à chacune de ses composantes connexes.

Attention : la condition sur le nombre d'arêtes est suffisante mais pas nécessaire ; si le graphe contient "trop" d'arêtes, il contient un cycle, mais s'il n'en contient pas trop, il se pourrait qu'il contienne quand même un cycle. Par exemple, le graphe constitué d'un triangle et d'un sommet isolé possède moins d'arêtes que son nombre de sommets, mais il contient pourtant un cycle. Il est donc important de savoir si le graphe en entrée est connexe, d'où notre distinction ci-dessus. Le raisonnement présenté nous donne l'algorithme 10.

Algorithme 10 : CONTIENTCYCLE(G)

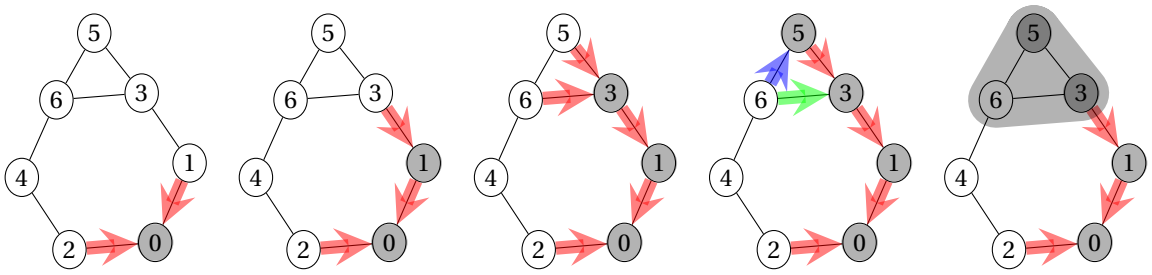
Entrées : un graphe non orienté G.
Sortie : VRAI si G contient un cycle, FAUX sinon.

```
1 pour chaque C dans COMPOSANTESCONNEXES(G) faire
2 |   H ← G.sous_graphe_induit(C);
3 |   si H.nombre_aretes() ≥ H.nombre_sommets() alors renvoyer VRAI;
4 renvoyer FAUX;
```

Les choses se compliquent quand on veut renvoyer explicitement un cycle du graphe. Intuitivement, un parcours du graphe (en largeur ou en profondeur) devrait pouvoir nous aider : on explore le graphe à partir d'un sommet arbitraire, et si l'on découvre deux chemins différents menant à un même sommet, c'est qu'il existe un cycle dans le graphe.

Pour ce faire, on ne peut pas simplement se contenter de marquer les sommets déjà visités : en effet, si l'on considérait que rencontrer un sommet déjà visité implique l'existence d'un cycle, alors une simple arête serait déjà identifiée comme un cycle. Au lieu de cela, on va enregistrer pour chaque sommet visité qui est son parent dans le parcours effectué (initialement, tous les parents sont à NIL) : si l'on rencontre en cours d'exploration un sommet dont un voisin non visité a déjà un parent, c'est qu'on a réussi à l'atteindre par un autre chemin, et donc qu'on a un cycle. Il faut encore montrer que l'on peut reconstruire ce cycle, ce que l'on fait dans la Proposition 2.4.1.

Exemple 15. Voici un cycle détecté à l'aide d'une exploration en profondeur au départ du sommet 0. Les sommets dont l'exploration est finie sont en gris ; à la dernière étape, on rencontre le sommet 6 parmi les voisins de 5, et on se rend compte que 6 a déjà comme parent 3. Dès lors, on conclut que l'union du chemin de 3 à 6 avec l'arête {3, 6} forme un cycle.



(1) (2) (3) (4) (5)

(fin exemple 15)

L'algorithme 11 implémente cette approche, en se basant sur un parcours en profondeur itératif.

Algorithme 11 : DÉTECTERCYCLEPROFONDEUR(G, départ)

Entrées : un graphe connexe non-orienté G et un sommet de départ.
Sortie : un cycle de G, ou NIL si G est acyclique.

```
1  déjà_visités ← tableau(G.nombre_sommets(), FAUX);
2  parents ← tableau(G.nombre_sommets(), NIL);
3  a_traiter ← pile();
4  a_traiter.empiler(départ);
5  tant que a_traiter.pas_vide() faire
6      sommet ← a_traiter.dépiler();
7      si ¬ déjà_visités[sommet] alors
8          déjà_visités[sommet] ← VRAI;
9          pour chaque voisin dans renverser(G.voisins(sommet)) faire
10             si ¬ déjà_visités[voisin] alors
11                 si parents[voisin] = NIL alors // sommet jamais vu ⇒ à traiter
12                     a_traiter.empiler(voisin);
13                     parents[voisin] ← sommet;
14                 sinon // sommet déjà accessible autrement
15                     cycle ← Graphe();
16                     ancien_parent ← parents[voisin];
17                     cycle.ajouter_arête(sommet, voisin);
18                     cycle.ajouter_arête(voisin, ancien_parent);
19                     tant que sommet ≠ ancien_parent faire // remonter l'autre
                        chemin
20                         cycle.ajouter_arête(sommet, parents[sommet]);
21                         sommet ← parents[sommet];
22                     renvoyer cycle;
23 renvoyer NIL;
```

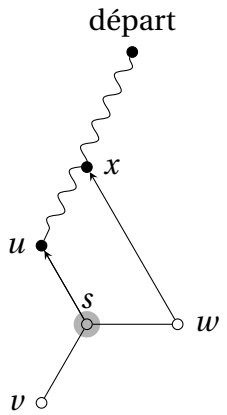
Montrons maintenant que l'approche proposée est correcte.

Proposition 2.4.1. L'algorithme 11 est correct.

Démonstration. La terminaison est claire et découle directement de celle du parcours en profondeur (algorithme 4). Pour ce qui est de la correction, le graphe connexe donné en

entrée n'est acyclique que s'il s'agit d'un arbre, et l'on ne se trouvera donc jamais dans le cas traité à la ligne 14.

Dans le cas contraire, supposons que l'on vient d'explorer un sommet u et que l'on examine maintenant un sommet s, qui est l'un des descendants non encore explorés de u. Si s possède un voisin w qui a déjà un parent x, alors ce parent a déjà été visité puisque c'est à ce moment-là qu'on a découvert que w avait un parent (ligne 13). Pour que la reconstruction du cycle fonctionne, il nous faut donc montrer que suivre de parent en parent le chemin suivi jusqu'à présent vers s nous permettra de remonter vers x. Autrement dit, il faut montrer que x est un ancêtre de u, ou éventuellement que $x = u$, et donc que la situation est celle présentée ci-dessous :

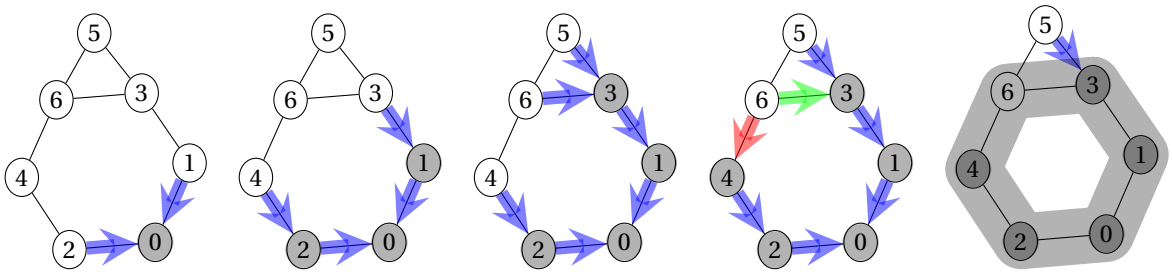


Si $x \neq u$, on doit montrer que x est un ancêtre de u. Pour ce faire, procédons par contradiction : si x n'est pas un ancêtre de u, il existe du moins un sommet a dans le graphe qui est un ancêtre commun de u et de x. L'exploration des descendants de u étant en cours à partir de a, on a donc forcément exploré x, ainsi que tous ses descendants y compris w, avant u ; dès lors, le parcours en profondeur aurait également dû visiter w avant de passer à l'exploration de u, ce qui contredit notre hypothèse selon laquelle w n'a pas encore été visité. □

On pourrait également utiliser un parcours en largeur plutôt qu'en profondeur pour détecter les cycles, mais la reconstruction du cycle détecté devient plus compliquée. En effet, dans le parcours en profondeur, l'arrivée sur un sommet ayant déjà un parent nous donne presque directement le cycle : il suffit d'enregistrer l'arête reliant le sommet à son ancien parent, et de remonter le cycle vers ce parent en suivant l'autre chemin déjà reconstruit par le parcours en profondeur et en ajoutant les arêtes ainsi rencontrées au cycle qu'on renverra. Dans le cas du parcours en largeur, on dispose de deux moitiés de cycle qu'il faut combiner en remontant les deux chemins en parallèle jusqu'à ce qu'on tombe sur un ancêtre commun (0 dans cet exemple). Le principe reste le même mais il faut savoir jusqu'où on doit remonter dans les deux chemins.

Exemple 16. Voici les étapes effectuées par un algorithme de détection de cycle basé sur un parcours en *largeur* pour détecter un cycle dans le graphe de l'exemple 15, au départ du même sommet 0 (avec une "accélération" pour le début de l'exploration). Les sommets dont l'exploration est finie sont en gris ; à la dernière étape, l'algorithme rencontre le sommet 6 parmi les voisins de 4, et se rend compte que 6 a déjà comme parent 3. Dès lors, il en conclut que l'union du chemin déjà découvert qui mène à 6 en passant

par 3 et du chemin menant à 6 en passant par 4 forme un cycle ; pour le reconstruire, il faut revenir sur nos pas en suivant les parents dans les deux chemins jusqu'à ce qu'on retombe sur l'origine commune.



(1) (2) (3) (4) (5)

(fin exemple 16)

Exercice 5. Modifiez l'algorithme 11 pour vous débarrasser de la structure déja_visités.

2.5 Clusters en dot

Comme on l'a vu dans les exemples précédents, il est parfois utile dans une représentation visuelle d'un graphe de regrouper certains sommets partageant la même propriété — par exemple celle d'appartenir à un sous-graphe particulier, qu'il s'agisse d'une composante connexe du graphe ou d'une classe d'un graphe biparti. Le format `dot` permet justement de spécifier des *clusters*, c'est-à-dire des sous-graphes d'un graphe donné qu'il faut regrouper dans le dessin pour une raison ou une autre. Pour ce faire, dans le graphe donné, on spécifie les sous-graphes de la façon suivante :

```
graph G {
    # ...
    subgraph cluster_moncluster {
        # définition des sommets et / ou des arêtes du cluster
        # ...
    }
}
```

Attention : il est important d'utiliser le préfixe `cluster_` après `subgraph` pour nommer le sous-graphe. De plus, seuls les programmes `dot` et `fdp` prennent en compte les clusters ; cela fonctionne également sur `http://webgraphviz.com/`.

2.6 Pour aller plus loin

Parcours. Si l'on ne sait rien de particulier sur les graphes à parcourir, et que seul l'affichage de leur contenu ou une tâche similaire nous intéresse, les deux stratégies de parcours proposées semblent équivalentes. Les cas d'applications du parcours en largeur semblent en revanche beaucoup plus limités que ceux du parcours en profondeur, qui nous permet de résoudre un grand nombre de problèmes, en modifiant l'algorithme de base de manière plus ou moins subtile selon les cas mais sans le rendre beaucoup plus gourmand en ressources. On en verra plusieurs exemples dans le chapitre 4 sur les graphes orientés ; outre

ces exemples et ceux déjà couverts dans ce chapitre, mentionnons que le parcours en profondeur permet également de détecter les “ponts” et les “points d’articulation” d’un graphe (les arêtes ou les sommets dont la suppression augmente le nombre de composantes connexes du graphe).

Connexité. La notion de connexité se généralise pour mesurer la “fragilité” d’un graphe : on dit qu’un graphe est k-connexe par les sommets (ou par les arêtes) s’il faut supprimer k sommets (ou arêtes) pour le déconnecter. Par exemple, un graphe ne contenant aucune arête est qualifié de 0-connexe, un arbre est 1-connexe, et un cycle est 2-connexe. Des algorithmes efficaces basés sur les parcours existent également pour calculer la k-connexité d’un graphe.

Bibliographie

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