

Graphes orientés et applications

OPTION INFORMATIQUE - TP n° 3.4 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ✎ expliquer l'intérêt pratique du tri topologique
- ✎ coder l'algorithme de tri topologique d'un graphe orienté
- ✎ détecter les cycles dans un graphe orienté
- ✎ trouver les composantes connexes d'un graphe
- ✎ faire le lien entre le problème 2-SAT et les graphes orientés

A Composantes fortement connexes d'un graphe orienté et 2-SAT

■ **Définition 1 — Composante fortement connexe d'un graphe orienté** $G = (V, E)$. Une composante fortement connexe d'un graphe orienté G est un sous-ensemble S de ses sommets, maximal au sens de l'inclusion, tel que pour tout couple de sommets $(s, t) \in S$ il existe un chemin de s à t dans G .

En notant \rightarrow^* la relation d'accessibilité du graphe, C est une composante fortement connexe de $G = (S, A)$ si et seulement si :

$$\forall (s, t) \in C, s \rightarrow^* t \text{ et } t \rightarrow^* s. \quad (1)$$

Le calcul des composantes connexes d'un graphe est par exemple utilisé pour résoudre le problème 2-SAT. Dans le cadre de ce problème, on dispose d'une formule logique sous la forme conjonctive normale et chaque clause comporte deux variables. Par exemple :

$$F_1 : (a \vee b) \wedge (b \vee \neg c) \wedge (\neg a \vee c) \quad (2)$$

On observe que l'assignation $a = b = c = 1$ est un modèle de F . F est donc satisfaisable. Comment automatiser cette vérification ?

L'idée est de construire un graphe à partir de la formule F . Supposons qu'elle soit constituée de m clauses et n variables (v_1, v_2, \dots, v_n) . On élabore alors un graphe $G = (V, E)$ à $2n$ sommets et $2m$ arêtes. Les sommets représentent les n variables v_i ainsi que leur négation $\neg v_i$. Les arêtes sont construites de la manière suivante : on transforme chaque clause de F de la forme $v_i \vee v_j$ en deux implications $\neg v_i \Rightarrow v_j$ ou $\neg v_j \Rightarrow v_i$. Cette transformation utilise le fait que la formule $a \Rightarrow b$ est équivalent à $\neg a \vee b$.

Théorème 1 F n'est pas satisfaisable si et seulement s'il existe une composante fortement connexe contenant une variable v_i et sa négation $\neg v_i$.

Démonstration. (\Leftarrow) S'il existe une composante fortement connexe contenant a et $\neg a$, alors cela signifie $F : (a \Rightarrow \neg a) \wedge (\neg a \Rightarrow a)$. Or cette formule n'est pas satisfaisable. En effet, si a est vrai alors $(a \Rightarrow \neg a)$ est faux, car du vrai on ne peut pas conclure le faux d'après la définition sémantique de l'implication. De

même, si a est faux alors $(\neg a \implies a)$ est faux, pour la même raison. Dans tous les cas, la formule est fausse. F n'est pas satisfaisable.

(\implies) Par contraposée. Supposons qu'il n'existe pas de composante fortement connexe contenant a et $\neg a$. Cela peut se traduire en la formule $\neg F : \neg(a \implies \neg a) \vee \neg(\neg a \implies a)$. Or, cette formule F est toujours satisfaisable. En effet, $\neg F$ s'écrit

$$\neg(\neg a \vee \neg a) \vee \neg(a \vee a) = a \vee \neg a \quad (3)$$

ce qui est toujours vérifié. Par contraposée, F est donc satisfaisable s'il existe une composante fortement connexe. ■

A1. En construisant le graphe de la formule suivante, statuer sur sa satisfaisabilité.

$$F_2 : (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b) \wedge (a \vee \neg c) \quad (4)$$

A2. On considère le graphe orienté équivalent à la formule F_2 . Choisir un algorithme déjà vu en cours pour calculer les composantes connexes de ce graphe et statuer sur la satisfaisabilité de F_2 . On pourra prendre la convention suivante pour numéroter les sommets :

```
(*
a -> 0
b -> 1
c -> 2
not a -> 3
not b -> 4
not c -> 5
*)
```

A3. En déduire une fonction `check_sat2` qui teste la satisfaisabilité de la formule F_2 .

A4. Ajouter une clause pour rendre la formule F_2 non satisfaisable et la tester sur l'algorithme.

A5. Quelle est la complexité de votre algorithme? Y-a-t-il un avantage à l'utiliser par rapport à l'algorithme de Quine?

B Ordre dans un graphe orienté acyclique

■ **Définition 2 — Graphe orienté.** Un graphe $G = (V, E)$ est orienté si ses arêtes sont orientées selon une direction. Les arêtes sont alors désignées par le mot arc.

Les graphes orientés peuvent représenter des contextes d'**ordonnement de tâches**, dans un projet industriel ou pour l'exécution d'un calcul par un ordinateur parallèle par exemple. Si deux sommets v et u sont des tâches à exécuter et si (v, u) est un arc, ceci peut être interprété comme : il faut réaliser la tâche v avant la u , probablement car la tâche u utilise le résultat de v .

(R) La relation d'accessibilité \rightarrow^* d'un graphe est la relation qui atteste de l'existence d'un chemin d'un sommet u à un sommet v dans un graphe G . C'est un préordre, c'est-à-dire une relation réflexive et transitive. En effet, elle n'est pas symétrique, car il se peut que $u \rightarrow^* v$ et $v \rightarrow^* u$ sans que $u = v$. Dans un graphe orienté acyclique, la relation d'accessibilité \rightarrow^* peut devenir une relation d'ordre \leq telle que $u \rightarrow^* v$ implique $u \leq v$.

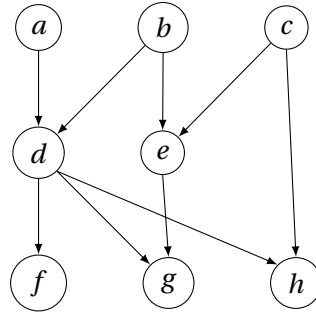


FIGURE 1 – Exemple de graphe orienté acyclique

Dans un graphe orienté **acyclique**, les arcs définissent un **ordre partiel**, le sommet à l'origine de l'arc pouvant être considéré comme le prédécesseur du sommet à l'extrémité de l'arc. Par exemple, sur la figure 1, *a* et *b* sont des prédécesseurs de *d* et *e* est un prédécesseur de *g*. Mais ces arcs ne disent rien de l'ordre entre *e* et *h*, l'ordre n'est pas total.

L'algorithme de tri topologique permet de créer un ordre total \leq sur un graphe orienté acyclique. Formulé mathématiquement, pour un graphe $G = (S, A)$:

$$\forall (u, v) \in V^2, (u, v) \in A \implies u \leq v \quad (6)$$

Sur l'exemple de la figure 1, plusieurs ordre topologiques sont possibles. Par exemple :

- a,b,c,d,e,f,g,h
- a,b,d,f,c,h,e,g

C Tri topologique et détection de cycles dans un graphe orienté

L'algorithme de tri topologique (cf. algorithme 1) utilise le parcours en profondeur d'un graphe pour marquer au fur et à mesure les sommets dans l'ordre topologique.

C1. Définir une variable `g` de type `int list array` qui représente le graphe de la figure 2 sous la forme d'une liste d'adjacence.

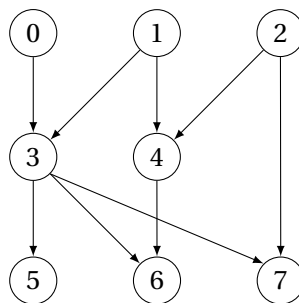


FIGURE 2 – Graphe orienté acyclique pour le tri topologique

C2. Définir un type somme `vertex_state` qui reflète l'état d'un sommet du graphe au cours de l'algorithme. On pourra choisir les constructeurs `To_Explore`, `Exploring` et `Explored`.

Algorithme 1 Tri topologique

```

1: Fonction TRI_TOPOLOGIQUE( $G$ )                                     ▷  $G$  est un graphe orienté
2:    $L \leftarrow$  une liste vide
3:   Marquer tous les sommets de  $G$  comme «non exploré»
4:   pour chaque sommet  $s$  in  $G$  répéter
5:     si  $s$  est «non exploré» alors
6:       VISITER( $G, s, L$ )
7:   renvoyer  $L$                                                          ▷ L'ordre topologique
8: Procédure VISITER( $G, s, L$ )                                         ▷ Parcours en profondeur depuis  $s$ 
9:   Marquer  $s$  «en cours d'exploration»
10:  pour chaque voisin  $u$  de  $s$  dans  $G$  répéter
11:    si  $u$  est «non exploré» alors
12:      VISITER( $G, u, L$ )
13:    sinon si  $u$  est «en cours d'exploration» alors
14:      Interrompre le programme car un cycle a été détecté
15:    sinon
16:      Ne rien faire
17:  Marquer  $s$  «exploré»
18:  Ajouter  $s$  en tête de la liste  $L$ 

```

C3. Écrire une fonction de signature `topological_sort : int list array -> int list` implémentant l'algorithme de tri topologique **1** en utilisant le type `vertex_state`. On pourra décomposer l'algorithme en deux fonctions, `visit` étant une fonction auxiliaire de `topological_sort`.

C4. Vérifier que cet algorithme détecte bien les cycles sur le graphe suivant :

```
let gc = [| [3] ; [3;4] ; [4;7] ; [5;6;7] ; [6] ; [] ; [0] ; [] |] ;;
```

C5. Tester l'algorithme sur le graphe :

```
let big = [| [3] ; [3;4] ; [3;4] ; [6] ; [3;7;9] ; [6] ; [8;9;10] ; [9] ;
  [10;11]; [11]; [] ;[] |] ;;
```

On cherche à dépasser le simple résultat de l'algorithme précédent. On souhaite déterminer précisément quelles sont les opérations que l'on pourrait exécuter parallèlement. Dans ce but, on met en place un horodatage des sommets :

- lorsqu'on lance la visite d'un sommet découvert «non exploré», celui-ci se voit attribué la date 0.
- chaque fils découvert dans la fonction `visit` se voit attribuer la date de son père plus un.
- lorsque l'exploration d'un sommet est finie, on ajoute un à date.

C6. Écrire une fonction de signature `date_topological_sort : int list array -> int list * int array` qui renvoie l'ordre topologique ainsi que les dates associées à chaque sommet.

C7. Quelles sont les opérations que l'on peut effectuer en parallèle sur le graphe de la figure **2**? Même question pour le graphe de la figure **3**

C8. Quelle est la complexité de cet algorithme? Comparer cette complexité à celle de l'algorithme de Floyds-Warshall. Pour détecter les cycles dans un graphe orienté, quel algorithme faudra-t-il choisir?

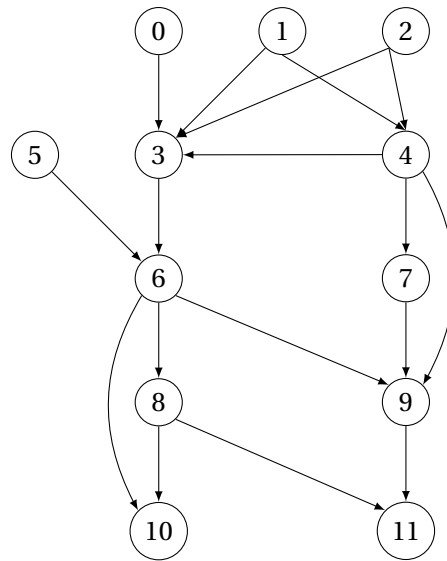


FIGURE 3 – Graphe orienté acyclique pour le tri topologique

★ D Trouver les composantes fortement connexes (suite)

D1. Soit \mathcal{C} la relation définie sur les sommets d'un graphe orienté G par : $(u, v) \in \mathcal{C}$ si et seulement si u et v font partie d'une même composante fortement connexe. Montrer que \mathcal{C} est une relation d'équivalence.

D2.