Graphes orientés et applications

OPTION INFORMATIQUE - TP n° 3.3 - 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 Ordre dans un graphe orienté acyclique

■ **Définition 1 — 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'**ordonnancement de tâches**, dans un projet industriel 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.

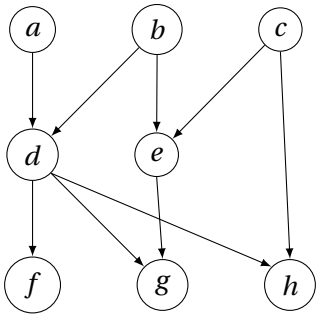

FIGURE 1 – Exemple de graphe orienté acyclique

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 $\preceq$ sur un graphe orienté acyclique.

Formulé mathématiquement :

$$\forall (v,u) \in V^2, (v,u) \in E \Longrightarrow v \preceq u \tag{1}$$

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

B Tri topologique et détection de cycles dans un graphe orienté

L'algorithme de tri topologique permet de construire un ordre dans un graphe orienté acyclique. C'est en fait un parcours en profondeur du graphe qui construit une pile en ajoutant le concept de date à chaque sommet : un date de début qui correspond au début du traitement du sommet et une date de fin qui correspond à la fin du traitement du sommet par l'algorithme. La pile contient à la fin les sommets dans un ordre topologique, les sommets par ordre de date de fin de traitement.

Au cours du parcours en profondeur, un sommet passe tout d'abord de l'ensemble des sommets non traités à l'ensemble des sommets en cours de traitement (date de début). Puis, lorsque la descente est finie (plus aucun arc ne sort du sommet courant), le sommet passe de l'ensemble en cours de traitement à l'ensemble des sommets traités (date de fin).

Algorithme 1 Algorithme de tri topologique

```
1 : Fonction TOPO_SORT(G = (V, E))
2 :     pile ← une pile vide
3 :     état ← un tableau des états des nœuds          ▷ pas traité, en cours de traitement ou traité
4 :     Les case du tableau état sont initialisées à «pas traité»
5 :     date ← 0                                       ▷ Date initiale
6 :     pour chaque sommet v de G répéter
7 :         si v n'est pas traité alors
8 :             TOPO_DFS(G = (V, E), pile, v, état, date)
9 :     renvoyer pile
10 : Fonction TOPO_DFS(G = (V, E), pile, v, état, date)
11 :    etat[v] ← «en cours de traitement»
12 :    Date de début de v ← date                      ▷ Par forcément nécessaire
13 :    pour chaque voisin u de v répéter
14 :        si u n'est pas traité alors TOPO_DFS(G = (V, E), pile, u, état, (date + 1))
15 :    etat[v] ← «traité»
16 :    Incrémenter la date de fin de v                ▷ Par forcément nécessaire
17 :    EMPILER(v, pile)
```

B1. Définir une variable sous la forme d'une liste d'adjacence qui représente le graphe de la figure 2.

B2. 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`.

B3. Coder l'algorithme de tri topologique en utilisant le type `vertex_state` et en le testant sur le graphe de la question précédente. Dans un premier temps, on n'implémentera pas les dates. Bien décomposer l'algorithme en deux fonctions, l'une récursive qui implémente un parcours en profondeur, l'autre itérative sur les sommets du graphe. Cette dernière renvoie la pile qui a enregistré l'ordre topologique.

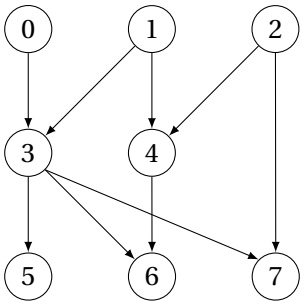

FIGURE 2 – Graphe orienté acyclique pour le tri topologique

B4. Modifier le tri topologique pour détecter les cycles dans un graphe orienté. On lèvera une exception lorsque le cycle est détecté. On peut observer que si l'algorithme découvre un sommet en cours d'exploration (`Exploring`), un cycle existe dans le graphe. Tester l'algorithme sur le graphe suivant :

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

B5. À quoi pourrait servir les dates de l'algorithme? Modifier l'algorithme pour qu'il renvoie les dates associées aux sommets.

B6. 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]; [] ;[]|] ;;
```

B7. Quelle est la complexité de cet algorithme?

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

■ **Définition 2 — 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.

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 :

$$F_1 : (a \vee b) \wedge (b \vee \neg c) \wedge (\neg a \vee c) \tag{2}$$

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 $(v_1, v_2, \ldots, v_n)$. 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 $\neg v_1 \Longrightarrow v_2$ ou $\neg v_2 \Longrightarrow v_1$. Cette transformation utilise le fait que la formule $a \Longrightarrow b$ 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 $\neg v_i$.

C1. En construisant le graphe de la formule suivante, statuer sur sa satisfaisabilité.

$$F_2 : (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b) \wedge (a \vee \neg c) \tag{3}$$

C2. 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
*)
```

C3. En déduire une fonction `check_sat2` qui teste la satisfaisabilité de la formule F_2.

C4. Ajouter une clause pour rendre la formule F_2 non satisfaisable et la tester sur l'algorithme.

C5. Quelle est la complexité de votre algorithme ? Y-a-t-il un avantage à l'utiliser par rapport à l'algorithme de Quine ?