# 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 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 $v$.

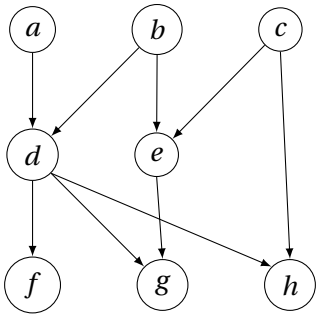



FIGURE 1 – Exemple de graphe orienté acyclique

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                          ▷ Contiendra les sommets dans l'ordre topologique
3 :     états ← un tableau des états des sommets      ▷ pas traité, en cours de traitement ou traité
4 :     dates ← un tableau des dates associées aux sommets
5 :     Les cases du tableau états sont initialisées à «pas traité»
6 :     Les cases du tableau dates sont initialisées à max_int   ▷ Date inconnue représentée par max_int
7 :     pour chaque sommet v de G répéter
8 :         si v n'est pas traité alors
9 :             TOPO_DFS(G = (V, E), pile, états, dates, v, 0)
10 :    renvoyer (pile, dates)
11 :
12 : Fonction TOPO_DFS(G = (V, E), pile, états, dates, v, date)
13 :    états[v] ← «en cours de traitement»
14 :    dates[v] ← date                               ▷ Au début de l'exploration la date de v vaut date
15 :    pour chaque voisin u de v répéter
16 :        si u n'est pas traité alors TOPO_DFS(G = (V, E), pile, états, dates, u, (date + 1))
17 :    états[v] ← «traité»
18 :    dates[v] += 1                                 ▷ Pour distinguer le début de la fin de l'exploration
19 :    EMPILER(v, pile)
```

B1. 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.

**Solution :**

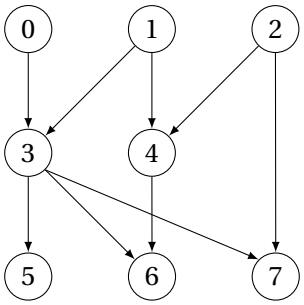



FIGURE 2 – Graphe orienté acyclique pour le tri topologique

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

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`.

**Solution :**

```
type vertex_state = To_Explore | Exploring | Explored;;
```

B3. Coder l'algorithme de tri topologique en utilisant le type `vertex_state` et en le testant sur le graphe `g`. Bien décomposer l'algorithme en deux fonctions, comme dans l'algorithme 1 : l'une itère sur les sommets du graphe et renvoie l'ordre topologique ainsi que les dates [1], l'autre est récursive, implémente un parcours en profondeur, modifie les états, les dates et la pile et renvoie `()`. Les signatures sont les suivantes :

- `topo_sort : int list array -> int list * int array`
- `topo_dfs : int list array -> int list ref -> vertex_state array -> int array -> int -> int -> unit`

B4. Modifier le tri topologique pour détecter les cycles dans un graphe orienté : il échoue alors en renvoyant le message `"CYCLE DETECTED"`. On peut observer que si l'algorithme de parcours en profondeur découvre un sommet en cours d'exploration (`Exploring`), alors 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. 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]; [] ;[]|] ;;
```

1. qui sont des entiers

**Solution :**

**Code 1 – Tri topologique et détection de cycles**

```
type vertex_state = To_Explore | Exploring | Explored;;
let g = [| [3] ; [3;4] ; [4;7] ; [5;6;7] ; [6] ; [] ; [] ; [] |] ;;
let gc = [| [3] ; [3;4] ; [4;7] ; [5;6;7] ; [6] ; [] ; [0] ; [] |] ;;
let big = [| [3] ; [3;4] ; [3;4] ; [6] ; [3;7;9] ; [6] ; [8;9;10] ; [9] ;
    [10;11]; [11]; [] ;[]|] ;;

let rec topo_dfs graph stack states dates d v =
    Printf.printf "Exploring vertex %i --- date --> %i \n" v d;
    states.(v) <- Exploring;
    dates.(v) <- d;
    let explore u =
      match states.(u) with
            | Explored ->   Printf.printf "Vertex %i --- already explored \n
                " v
            | Exploring ->  failwith "CYCLE_DETECTED"
            | To_Explore -> topo_dfs graph stack states dates (d + 1) u
    in  List.iter explore graph.(v);
        states.(v) <- Explored;
        dates.(v) <- dates.(v) + 1;
        stack := v::!stack;;


let topo_sort graph =
    let n = Array.length graph and stack = ref [] in
    let states = Array.make n To_Explore and dates = Array.make n max_int in
    for v = 0 to n - 1 do
        if states.(v) = To_Explore then topo_dfs graph stack states dates 0
            v
    done;
    (!stack, dates);;

topo_sort g;;
topo_sort big;;
topo_sort gc;;

(*let () = assert ((([2; 1; 4; 0; 3; 7; 6; 5], [|1; 1; 1; 2; 2; 3; 3; 3|]) =
     (topo_sort g)));;*)
(*let () = assert ((([5; 2; 1; 4; 7; 0; 3; 6; 9; 8; 11; 10],[|1; 1; 1; 2; 2;
     1; 3; 3; 4; 4; 5; 5|]) = (topo_sort big)));;*)
```

B6. À quoi servent les dates de l'algorithme ?

**Solution :** Si le graphe représente un ordonnancement de tâches, les dates permettent de savoir dans quel ordre on peut les effectuer. Celles qui possèdent la même date peuvent être effectuées en parallèle. Dans le cadre de la gestion de projets complexes, cette information est capitale.

B7. Quelle est la complexité de cet algorithme ?

**Solution :** Grâce à la représentation sous la forme de liste d'adjacence, on note que `topo_sort` parcours tous les sommets et que `topo_dfs` parcours une fois chaque arête. On en déduit que la complexité est $O(n+m)$ si $n$ est l'ordre du graphe et $m$ sa taille.

## 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 $\neg v_i$.

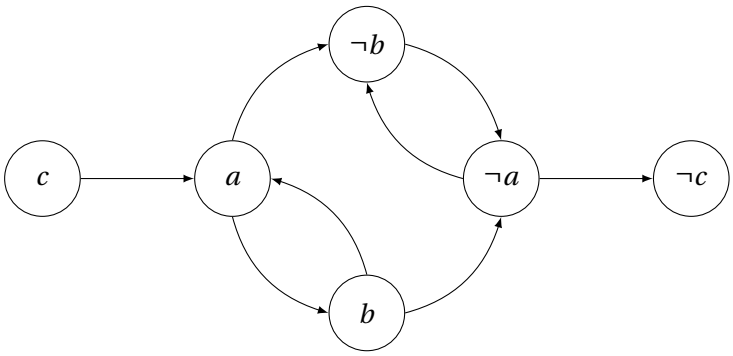
*Démonstration.* ($\Longleftarrow$) S'il existe une composante fortement connexe contenant $a$ et $\neg a$, alors cela signifie $F : (a \Longrightarrow \neg a) \wedge (\neg a \Longrightarrow a)$. Or cette formule n'est pas satisfaisable. En effet, si $a$ est vrai alors $(a \Longrightarrow \neg a)$ est 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 $(\neg a \Longrightarrow a)$ est faux, pour la même raison. Dans tous les cas, la formule est fausse. $F$ n'est pas staisfaisable.

($\Longrightarrow$), par contraposée. Supposons qu'il n'existe pas de composante fortement connexe contenant $a$ et $\neg a$. Cela peut se traduire en la formule $F : (a \Longrightarrow \neg a) \vee (\neg a \Longrightarrow a)$ : soit il n'existe aucun chemin de $a$ à $\neg a$, soit il existe un chemin dans un seul sens, mais pas dans les deux. Cette formule $F$ est toujours satisfaisable. En effet, si $a$ est vrai, alors $\neg a \Longrightarrow a$ est vraie, car *ex falso quodlibet*, et donc $F$ est vraie. Si $a$ est faux, alors $a \Longrightarrow \neg a$ est vraie pour la même raison. Dans tous les cas, $F$ est vraie. On peut également le montrer en remarquant que $F$ s'écrit $(a \vee \neg a) \vee (\neg a \vee a) = a \vee \neg a$. $F$ est donc satisfaisable. ■

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

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

**Solution :** La formule $F_2$ est satisfaisable car il n'existe pas de composante fortement connexe contenant une variable est sa négation.




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
*)
```

---

**Solution :** L'idée la plus simple est de savoir s'il existe un chemin dans le graphe entre une variable et sa négation. Pour cela, on peut utiliser l'algorithme de Floyd-Warshall (un exemple de programmation dynamique). S'il existe une composante fortement connexe entre une variable et sa négation, alors les deux coefficients associés $w_{i,\neg i}$ et $w_{\neg i,i}$ sont finis.

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

  let mf2 = [|[|0; 1; max_int; max_int; 1; max_int |] ;
          [|1; 0; max_int; 1; max_int; max_int|] ;
          [|1; max_int; 0; max_int; max_int; max_int|] ;
          [|max_int; max_int; max_int; 0; 1; 1|];
          [|max_int; max_int; max_int; 1; 0; max_int|];
          [|max_int; max_int; max_int; max_int; max_int; 0|];
          |] ;;

  let mf2_mod = [|[|0; 1; max_int; max_int; 1; max_int |] ;
          [|1; 0; max_int; 1; max_int; max_int|] ;
          [|1; max_int; 0; max_int; max_int; max_int|] ;
```

```
            [|max_int; 1; max_int; 0; 1; 1|];
            [|1; max_int; max_int; 1; 0; max_int|];
            [|max_int; max_int; max_int; max_int; max_int; 0|];
            |] ;;

    let floyd_warshall m =
      let w_sum wi wj =
        if wi = max_int || wj = max_int then max_int else wi + wj
      in let w = Array.copy m and n = Array.length m
        in
          for k = 0 to n-1 do
            for i = 0 to n-1 do
              for j = 0 to n-1 do
                w.(i).(j) <- min (w.(i).(j)) (w_sum w.(i).(k) w.(k).(j))
              done;
            done;
          done;
      w ;;
```

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

**Solution :**

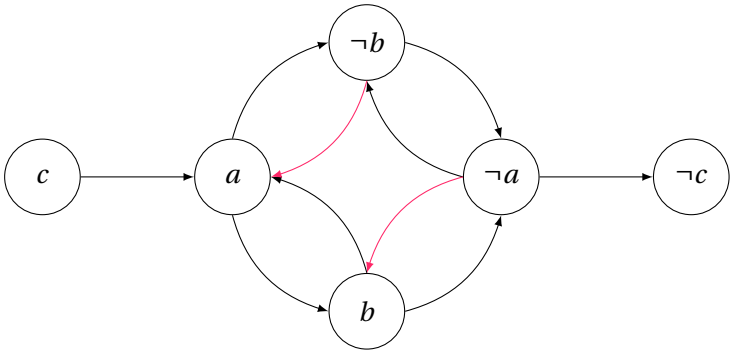
```
let is_inf x = x = max_int;;

let check_sat2 m = (is_inf m.(0).(3) || is_inf m.(3).(0))
                   && (is_inf m.(1).(4) || is_inf m.(4).(1))
                   && (is_inf m.(2).(5) || is_inf m.(5).(2));;
```

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

**Solution :** La formule modifiée peut être :

$$F_2 : (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b) \wedge (a \vee \neg c) \wedge (a \vee b) \tag{4}$$

On ajoute ainsi un arc $(\neg a, b)$ et un autre $(\neg b, a)$. Ce qui correspond au graphe :




On voit clairement que $\{a, \neg b, \neg a, b\}$ forme une composante fortement connexe. Le graphe correspondant est :

```
let mf2_mod = [|[|0; 1; max_int; max_int; 1; max_int |] ;
                [|1; 0; max_int; 1; max_int; max_int|] ;
                [|1; max_int; 0; max_int; max_int; max_int|] ;
                [|max_int; 1; max_int; 0; 1; 1|];
                [|1; max_int; max_int; 1; 0; max_int|];
                [|max_int; max_int; max_int; max_int; max_int; 0|];
              |] ;;
```

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

**Solution :** Une table de vérité d'une formule logique à $n$ variables contient $2^n$ lignes. Une recherche exhaustive dans la table est donc un algorithme en $O(2^n)$, c'est à dire de complexité exponentielle dans le pire des cas. L'algorithme de Quine ne fait pas mieux qu'une recherche exhausitve dans table de vérité dans le pire des cas. Mais en pratique, il permet d'éviter de parcourir un certain nombre de branches de l'arbre d'exploration.

L'algorithme de FLoyd-Warshall est en $O(n^3)$. Il est donc meilleur dans le pire des cas. On peut encore faire mieux avec les algoritmes de Kosaraju ou Tarjan qui calcule les composantes fortement connexes en $O(n+m)$.

On peut donc conclure que SAT-2 est un problème de décision polynomial. C'est une restriction à des clauses de deux variables du problème général SAT qui lui est NP-complet.