Graphes orientés et applications

OPTION INFORMATIQUE - TP nº 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 $\to^*$ la relation d'accessibilité du graphe, C est une composante fortement connexe de $G = (S, A)$ si et seulement si :

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

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 :

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

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 $\neg v_i \Longrightarrow v_j$ ou $\neg v_j \Longrightarrow v_i$. 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 $\neg F : \neg(a \Longrightarrow \neg a) \vee \neg(\neg a \Longrightarrow a)$. Or, cette formule F est toujours satisfaisable. En effet, $\neg F$ s'écrit

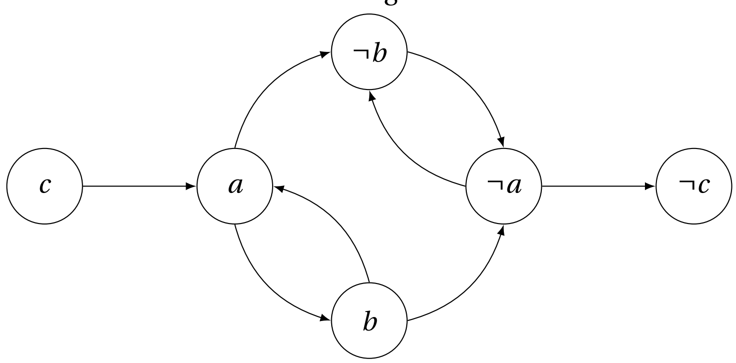
$$\neg(\neg a \vee \neg a) \vee \neg(a \vee a) = a \vee \neg a \tag{3}$$

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.

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

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



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
*)
```

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 ;;
```

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

Solution :

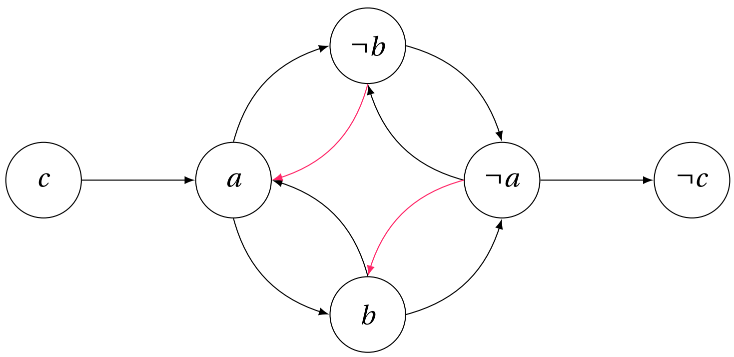
```
let is_inf x = x = max_int;;

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));;
```

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

Solution : La formule modifiée peut être :

$$F_2 : (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b) \wedge (a \vee \neg c) \wedge (a \vee b) \quad (5)$$

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|];
              |] ;;
```

A5. 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.

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'**ordonnancement 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.

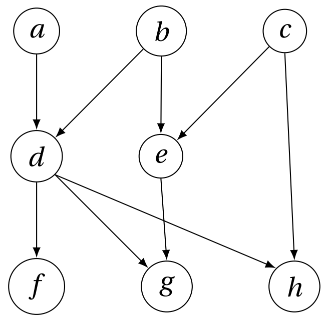


FIGURE 1 – Exemple de graphe orienté acyclique

R La relation d'accessibilité $\to^*$ 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 $u \to^\star v$ et $v \to^\star u$ sans que $u = v$. Dans un graphe orienté acyclique, la relation d'accessibilité $\to^*$ peut devenir une relation d'ordre $\preceq$ telle que $u \to^\star v$ implique $u \preceq v$.

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, pour un graphe $G = (S, A)$:

$$\forall (u, v) \in V^2, (u, v) \in A \Longrightarrow u \preceq v \tag{6}$$

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.

Solution :

Algorithme 1 Tri topologique

```
Fonction TRI_TOPOLOGIQUE(G)                          ▷ G est un graphe orienté
    L ← une liste vide
    Marquer tous les sommets de G comme «non exploré»
    pour chaque sommet s in G répéter
        si s est «non exploré» alors
            VISITER(G, s, L)
    renvoyer L                                       ▷ L'ordre topologique
Procédure VISITER(G, s, L)                           ▷ Parcours en profondeur depuis s
    Marquer s «en cours d'exploration»
    pour chaque voisin u de s dans G répéter
        si u est «non exploré» alors
            VISITER(G, u, L)
        sinon si u est «en cours d'exploration» alors
            Interrompre le programme car un cycle a été détecté
        sinon
            Ne rien faire
    Marquer s «exploré»
    Ajouter s en tête de la liste L
```

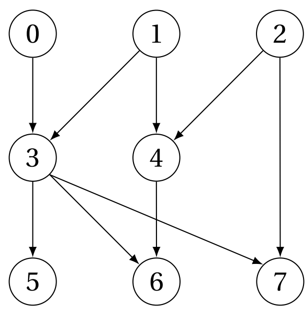


FIGURE 2 – Graphe orienté acyclique pour le tri topologique

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

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`.

Solution :

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

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`.

Solution :

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

  let topological_sort graph =
    let order = ref [] in
    let states = Array.make (Array.length graph) To_Explore in
    let rec visit s =
      match states.(s) with
      | Explored -> ()
      | Exploring -> failwith "CYCLE DETECTED"
      | To_Explore -> states.(s) <- Exploring;
                      List.iter visit graph.(s);
                      states.(s) <- Explored;
                      order := s::!order;  in
    for s = 0 to (Array.length graph) - 1 do
      if states.(s) = To_Explore then visit s
    done;
    !order;;
```

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]; [] ;[]|] ;;
```

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)));;*)
```

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.

Solution :

```
  let date_topological_sort graph =
    let order = ref [] in
    let dates = Array.make (Array.length graph) 0 in
    let states = Array.make (Array.length graph) To_Explore in
    let rec visit d s =
      match states.(s) with
      | Explored -> ()
      | Exploring -> failwith "CYCLE DETECTED"
      | To_Explore -> states.(s) <- Exploring;
                      dates.(s) <- d;
                      List.iter (visit (d+1)) graph.(s);
                      states.(s) <- Explored;
                      dates.(s) <- dates.(s) + 1;
                      order := s::!order;  in
    for s = 0 to (Array.length graph) - 1 do
      if states.(s) = To_Explore then visit 0 s
    done;
```

```
    (!order, dates);;
```

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

Solution :

```
([2; 1; 4; 0; 3; 7; 6; 5], [|1; 1; 1; 2; 2; 3; 3; 3|])
([5; 2; 1; 4; 7; 0; 3; 6; 9; 8; 11; 10], [|1; 1; 1; 2; 2; 1; 3; 3; 4; 4; 5;
    5|])
```

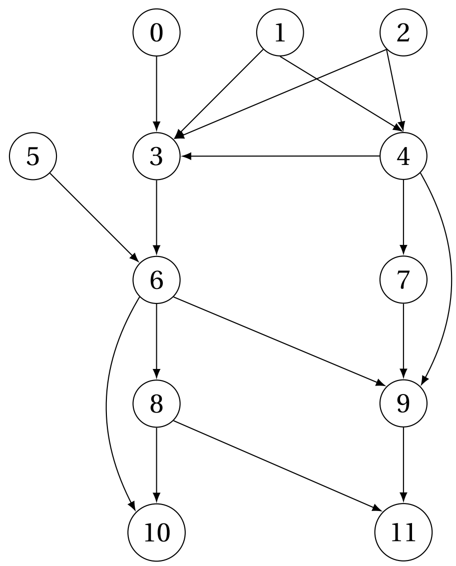


FIGURE 3 – Graphe orienté acyclique pour le tri topologique

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

Solution : Grâce à la représentation sous la forme de liste d'adjacence, on note que `topo_sort` parcours tous les sommets et que `date_topological_sort` 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.
Floyd-Warshall est en $O(n^3)$. S'il s'agit de détecter les cycles, le tri topologique est donc bien plus efficace.

★ 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.

Solution :

- Réflexivité : soit u un sommet de G. On a bien $(u, u) \in \mathcal{C}$, il existe un chemin de u à u.
- Symétrie : si $(u, v) \in \mathcal{C}$ alors il existe un chemin de u à v et un chemin de v à u, d'après la définition d'une partie fortement connexe.
- Transitive : si $(u, v) \in \mathcal{C}$ et si $(v, w) \in \mathcal{C}$, alors on a bien si $(u, w) \in \mathcal{C}$, puisqu'il existe un chemin de u à v, de v à w et réciproquement.

$\mathcal{C}$ est donc une relation d'équivalence.

D2.