

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

■ **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.

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

■ **Définition 2 — Composante fortement connexe d'un graphe orienté** $G = (S, A)$. Une composante fortement connexe d'un graphe orienté G est un sous-ensemble C de ses sommets S , maximal au sens de l'inclusion, tel que pour tout couple de sommets $(s, t) \in C$ 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 si et seulement si C est maximale pour l'inclusion et :

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

Ⓡ On peut également formuler cette définition en termes de sous-graphe induit : dans un graphe non orienté, **une composante connexe est un sous-graphe induit maximal connexe**, c'est-à-dire un ensemble de points qui sont reliés deux à deux par un chemin.

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

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

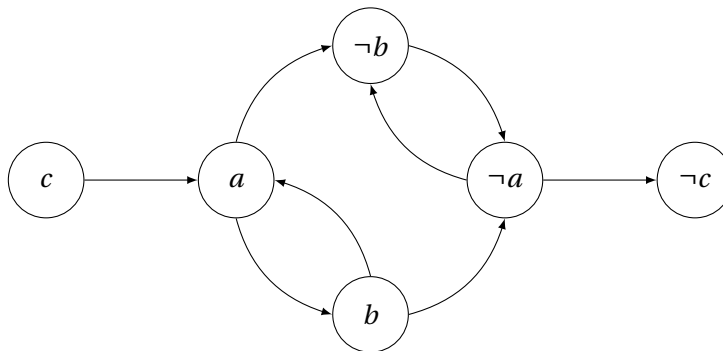
$$\neg(\neg a \vee \neg a) \vee \neg(a \vee a) = a \vee \neg a \quad (3)$$

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.

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

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));;
```

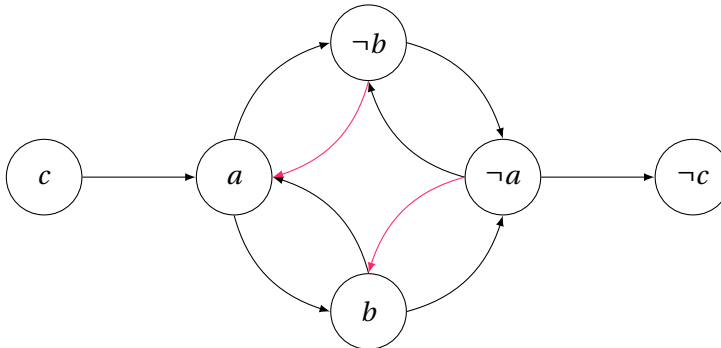


A4. Ajouter une clause pour rendre la formule F_2 non satisfaisable et la tester sur l'algorithme.

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 exhaustive dans la 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 algorithmes de Kosaraju ou Tarjan qui calculent 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

Les graphes orientés acycliques 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 u et v sont des tâches à exécuter et si (u, v) est un arc, ceci peut être interprété comme : il faut réaliser la tâche u avant la v , probablement car la tâche v utilise le résultat de u .

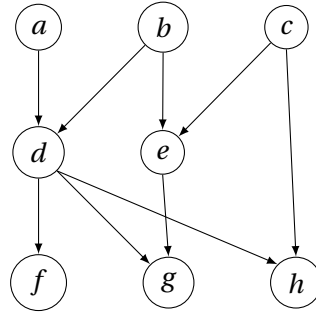


FIGURE 1 – Exemple de graphe orienté acyclique

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 antisymétrique, 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^a \leq telle que $u \rightarrow^* v$ implique $u \leq v$.

a. c'est-à-dire binaire, réflexive, antisymétrique et transitive

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 $G = (S, A)$:

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

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

- $a \leq b \leq c \leq d \leq e \leq f \leq g \leq h$
- $a \leq b \leq d \leq f \leq c \leq h \leq e \leq g$

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. Une pile est utilisée pour enregistrer l'ordre chronologique de découverte des sommets.

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.

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  $s$  de  $G$  répéter
5:     si  $s$  est «non exploré» alors
6:       EXPLORER( $G, s, L$ )
7:   renvoyer  $L$                                                      ▷ L'ordre topologique
8: Procédure EXPLORER( $G, s, L$ )                                       ▷ 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:      EXPLORER( $G, u, L$ )
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                                                 ▷ sommet déjà exploré
17:  Marquer  $s$  «exploré»
18:  Ajouter  $s$  en tête de la liste  $L$ 

```

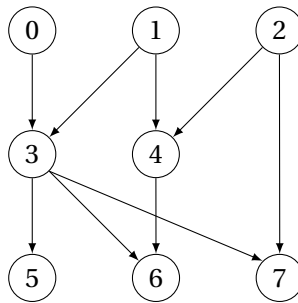


FIGURE 2 – Graphe orienté acyclique pour le tri topologique

Solution :

```
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 le tri topologique.

tant l'algorithme de tri topologique **1** en utilisant le type `vertex_state`. On pourra décomposer l'algorithme en deux fonctions, `explore` 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 explore s =
    match states.(s) with
    | Explored -> ()
    | Exploring -> failwith "CYCLE DETECTED"
    | To_Explore -> states.(s) <- Exploring;
                    List.iter explore graph.(s);
                    states.(s) <- Explored;
                    order := s::!order; in
  for s = 0 to (Array.length graph) - 1 do
    if states.(s) = To_Explore then explore 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] ; [] |] ;;
```

Solution :

```
[2; 1; 4; 0; 3; 7; 6; 5]
```

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 :

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

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 sommet découvert dans la procédure `explorer` se voit attribuer la date de son parent plus un.

- si un sommet est redécouvert alors qu'il a été exploré et si sa date est inférieure ou égale à la date courante de découverte, alors il faut mettre à jour ce sommet ainsi que tous ses descendants.
 - 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 explore d s =
    let rec update d u =
      dates.(u) <- d + 1;
      List.iter (update (d+1)) (List.filter (fun v -> dates.(v) <= (d+1))
        graph.(u)) in
    match states.(s) with
    | Explored -> if dates.(s) <= d then update d s (* mettre à jour le
      sous-arbre *)
    | Exploring -> failwith "CYCLE DETECTED"
    | To_Explore -> states.(s) <- Exploring;
      dates.(s) <- d;
      List.iter (explore (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 explore 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; 3; 2; 1; 4; 3; 5; 5; 6;
6|])

```

- C8. Quelle est la complexité de cet algorithme? Comparer cette complexité à celle de l'algorithme de Floyd-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` parcourt tous les sommets et que `date_topological_sort` parcourt 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.

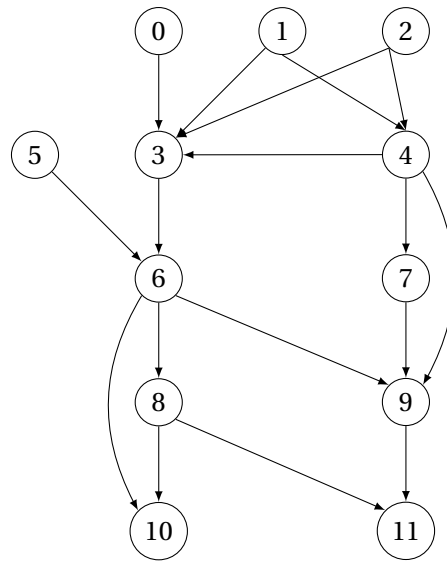


FIGURE 3 – Graphe orienté acyclique pour le tri topologique

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 u et v deux sommets de G et \mathcal{R} la relation binaire définie sur les sommets d'un graphe orienté G par : $u\mathcal{R}v$ si et seulement si u et v font partie d'une même composante fortement connexe \mathcal{C} .

Solution :

- Démonstration.*
- Réflexivité : soit u un sommet de G . On a bien $u\mathcal{R}u$, car u est trivialement connecté à u .
 - Symétrie : si $u\mathcal{R}v$ alors il existe un chemin de u à v , et v et u appartiennent à la même composante fortement connexe. D'après la définition d'une composante fortement connexe, on a donc $v\mathcal{R}u$.
 - Transitive : si $u\mathcal{R}v$ et si $v\mathcal{R}w$, alors on a bien $u\mathcal{R}w$ car u et w appartiennent à la même composante fortement connexe que v . On a $w\mathcal{R}v$ et $v\mathcal{R}u$: il existe donc un chemin de w à u .

\mathcal{R} est donc une relation d'équivalence. ■

(R) L'ensemble des composantes connexes d'un graphe orienté $G = (S, A)$ forme une partition de S .

■ **Définition 3 — Graphe quotient.** Le graphe quotient d'un graphe orienté $G = (S, A)$ est le graphe $G_q = (S_q, A_q)$ où :

- S_q est l'ensemble des composantes fortement connexes de G , c'est-à-dire chaque sommet de G_q est une composante connexe de G .
- $A_q = \{(c_1, c_2) \in S_q^2, c_1 \neq c_2 \text{ et } \exists (u, v) \in c_1 \times c_2, (u, v) \in A\}$

D2. Montrer que G_q est acyclique.

Solution :

Démonstration. On procède par l'absurde. Supposons que G_q , le graphe quotient de G soit cyclique. Cela signifierait qu'il existerait un cycle dans G_q et donc on pourrait trouver un sommet c_i de S_q tel qu'il existerait un chemin dans G_q tel que $c_i \rightarrow^* c_i$. Autrement dit, il existerait un cycle qui relierait des composantes connexes de G . Mais alors, ce cycle serait lui-même une composante connexe de G et devrait donc être un sommet de G_q . On aboutit alors une contradiction : G_q ne serait pas le graphe quotient de G .

C'est pourquoi un graphe orienté est un graphe acyclique de ses composantes fortement connexes. ■

D3. Implémenter l'algorithme de Tarjan (cf. algorithme 2) qui calcule les composantes fortement connexes d'un graphe orienté donné sous la forme d'une liste d'adjacence.

Solution :

```
let tarjan_scc g =
  let n = Array.length g in
  let index = Array.make n (-1) in
  let lowlink = Array.make n (-1) in
  let on_stack = Array.make n false in
  let stack = ref [] in
  let cc_list = ref [] in
  let current_index = ref 0 in

  let rec connecter v =
    index.(v) <- !current_index;
    lowlink.(v) <- !current_index;
    current_index := !current_index + 1;
    stack := v :: !stack;
    on_stack.(v) <- true;

    let rec explore neighbours =
      match neighbours with
      | [] -> ()
      | w::t when index.(w) = -1 -> connecter w; lowlink.(v) <- min
        lowlink.(v) lowlink.(w); explore t
      | w::t -> if on_stack.(w) then lowlink.(v) <- min lowlink.(v)
        index.(w); explore t in
    explore g.(v);

    if index.(v) = lowlink.(v) then
      let rec construire_cc cc =
```

Algorithme 2 Algorithme de Tarjan pour le calcul des composantes fortement connexes

```

1: Fonction TARJAN( $g$ )
2:    $cc\_list \leftarrow$  une liste vide                                ▷ La liste des composantes (résultat)
3:    $S \leftarrow$  une pile vide
4:    $n \leftarrow$  l'ordre du graphe
5:    $index \leftarrow$  un tableau de taille  $n$  initialisé à  $-1$         ▷ La composante d'un sommet
6:    $on\_stack \leftarrow$  un tableau de booléens de taille  $n$  initialisé à Faux ▷ Atteste de la présence sur la pile
7:    $current\_index \leftarrow 0$                                      ▷ L'index de la composante en cours de construction
8:   pour  $v \leftarrow 0$  à  $n - 1$  répéter
9:     si  $index[v] = -1$  alors                                     ▷ Le sommet n'a pas ni exploré ni connecté
10:      CONNECTER( $v$ )
11:   renvoyer  $cc\_list$ 
12: Procédure CONNECTER( $v$ )
13:    $index[v] \leftarrow current\_index$ 
14:    $lowlink[v] \leftarrow current\_index$ 
15:    $current\_index \leftarrow current\_index + 1$ 
16:   EMPILER( $S, v$ )
17:    $on\_stack[v] \leftarrow Vrai$ 
18:   pour tout voisin  $w$  de  $v$  répéter
19:     si  $index[w] = -1$  alors
20:       CONNECTER( $w$ )
21:        $lowlink[v] \leftarrow \min(lowlink[v], lowlink[w])$ 
22:     sinon si  $on\_stack[w]$  alors
23:        $lowlink[v] \leftarrow \min(lowlink[v], index[w])$ 
24:   si  $index[v] = lowlink[v]$  alors                                ▷ On a trouvé une composante connexe
25:      $cc \leftarrow$  une liste vide représentant la nouvelle composante fortement connexe à créer
26:     repeat
27:        $w \leftarrow$  DÉPILER( $S$ )
28:        $on\_stack[w] \leftarrow Faux$ 
29:       AJOUTER( $cc, w$ )
30:     until  $w = v$ 
31:     AJOUTER( $cc\_list, cc$ )

```

```
match !stack with
| [] -> ()
| w::t -> stack := t;
        on_stack.(w) <- false;
        if w = v
        then cc_list := cc :: !cc_list
        else construire_cc (w :: cc);
in
construire_cc [v]; (* construire une nouvelle composante fortement
                   connexe partant de v en dépilant *)
in
for v = 0 to n - 1 do
  if index.(v) = -1 (* à connecter *)
  then connecter v
done;
!cc_list;;
```

D4. Quelle est la complexité de l'algorithme de Tarjan?

Solution : Cet algorithme parcourt une fois les sommets et chaque arcs. Donc la complexité est en $O(n + m)$.