TP XXXVIII

PARCOURS DE GRAPHES EN OCAML

Les exercices marqués d'une flèche ▸ doivent être vus comme faisant partie du cours.

1 Parcours

Dans tout le sujet, les graphes sont supposés donnés sous forme d'un tableau de listes d'adjacence :

```
type sommet = int
type graphe = sommet list array
```

1.1 Parcours en profondeur

▸ **Exercice XXXVIII.1 – Parcours en profondeur récursif** p. 663

Écrire une fonction `dfs` telle que `dfs pre post g x0` effectue un parcours en profondeur du graphe `g` à partir du sommet `x0`, en exécutant `pre x` à l'ouverture du sommet `x` et `post x` à la fermeture.
Il s'agit simplement de traduire le pseudo-code donné dans le cours en OCaml.

```
dfs : (sommet -> unit) -> (sommet -> unit) -> graphe -> sommet -> unit
```

Exercice XXXVIII.2 – Sur un exemple p. 663

On considère les graphes suivants :

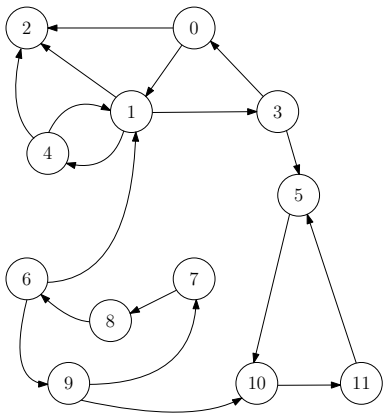


FIGURE XXXVIII.1 – Le graphe `g0`

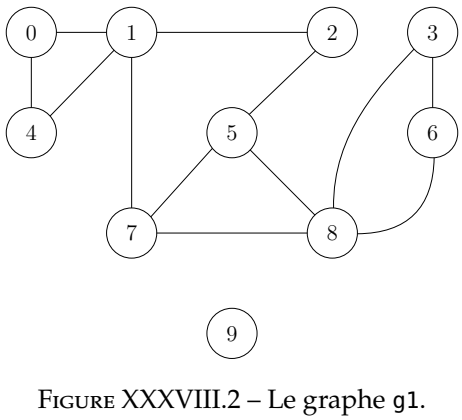


FIGURE XXXVIII.2 – Le graphe `g1`.

Ces graphes sont stockés de façon à ce que les listes d'adjacence soient *en ordre croissant*.
On définit :

```
let ouvre x = Printf.printf "Ouverture %d\n" x
let ferme x = Printf.printf "Fermeture %d\n" x
```

Déterminer à la main l'affichage produit par `dfs ouvre ferme g0 0` et par `dfs ouvre ferme g1 5` puis vérifier sur l'ordinateur.

I.2 Parcours en largeur

Exercice XXXVIII.3 – Parcours en largeur à l'aide d'une file p. 664

1. Écrire une fonction `bfs` effectuant un parcours en largeur. On traduira le pseudo-code du cours en utilisant le module `Queue` ; le premier argument de `bfs` correspond à la fonction Traitement et doit être appelé au moment où l'on extrait un sommet de la file.

```
Queue.create : unit -> 'a Queue.t (* crée une file vide *)
Queue.is_empty : 'a Queue.t -> bool
Queue.pop : 'a Queue.t -> 'a
Queue.push : 'a -> 'a Queue.t -> unit
```

```
bfs : (sommet -> unit) -> graphe -> sommet -> unit
```

2. Vérifier que `bfs ouvre g0 0` et `bfs ouvre g1 5` donnent bien ce que vous pensiez.
3. Si l'on ne souhaite pas utiliser le module `Queue`, comment peut-on réaliser de manière efficace une file impérative ? fonctionnelle ? On ne demande pas d'implémenter ces structures mais simplement de se remémorer les techniques vues depuis le début de l'année.

Exercice XXXVIII.4 – Parcours en largeur avec frontière explicite p. 664

Écrire une fonction de parcours en largeur utilisant le principe suivant :

- on a toujours un ensemble *vus* codé par un tableau de booléens ;
- on utilise deux listes appelées `frontiere` et `nouveaux` ;
- au début d'une itération, `frontiere` contient tous les sommets situés à une certaine distance k du sommet initial et `nouveaux` est vide ;
- on parcourt `frontiere`, et pour chaque sommet on rajoute ses voisins non explorés à `nouveau` ;
- à la fin de ce parcours, on passe à l'itération suivante avec la liste `nouveau` qui devient la nouvelle liste `frontiere`.

Remarque
La manière la plus simple de coder cette fonction en OCaml est purement fonctionnelle.

2 Accessibilité, composantes connexes

Exercice XXXVIII.5 – Accessibilité p. 665

1. Écrire une fonction `accessible` telle que l'appel `accessible g x y` détermine si `y` est accessible depuis `x` dans le graphe (*a priori* orienté) `g`, à l'aide d'un parcours en profondeur.

```
accessible : graphe -> sommet -> sommet -> bool
```

2. Modifier cette fonction pour qu'elle réponde dès que possible.
   Remarque
   Le plus simple est d'utiliser une exception.

▸ Exercice XXXVIII.6 – Composantes connexes

p. 666

1. Écrire une fonction `tab_composantes : graphe -> int array` qui prend en entrée un graphe *supposé non orienté* et renvoie un tableau `t` tel que $t_i = t_j$ si et seulement si les sommets x_i et x_j sont dans la même composante connexe du graphe.
2. Écrire une fonction `listes_composantes : graphe -> sommet list list` qui renvoie les composantes connexes sous forme de liste de listes.
   On ne réutilisera pas la fonction précédente et on n'hésitera pas à passer par des `list ref`.

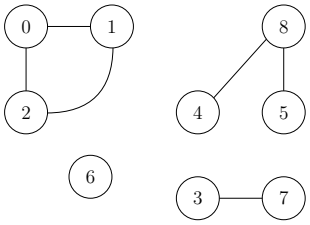


Figure XXXVIII.3 – Graphe `g2`.

```
utop[68]> liste_composantes g2;;
- : sommet list list = [[6]; [5; 8; 4]; [7; 3]; [2; 1; 0]]
utop[69]> tab_composantes g2;;
- : sommet array = [|0; 0; 0; 3; 4; 4; 6; 3; 4|]
```

2.1 Construction de l'arborescence d'un parcours

Il est très souvent utile de générer explicitement l'arborescence (l'arbre enraciné) associé à un parcours de graphe : en particulier, cela permet ensuite de reconstituer facilement des chemins. La manière la plus simple de procéder est généralement de stocker l'arbre *orienté des feuilles vers la racine* : cela peut se faire facilement à l'aide d'un tableau de taille n (nombre de sommets) :

- si le sommet i n'est pas dans l'arborescence, la case i du tableau contiendra une valeur particulière (`None`, ou -1, ou...);
- si le sommet i est un nœud autre que la racine de l'arbre, la case i contiendra l'indice du parent de i;
- si le sommet i est la racine de l'arbre, la case i contiendra i.

▸ Exercice XXXVIII.7 – Arbre de parcours, reconstitution de chemins

p. 666

On reprend les types utilisés plus haut :

```
type sommet = int
type graphe = sommet list array
```

1. Écrire une fonction `arbre_dfs` effectuant un parcours en profondeur d'un graphe G à partir d'un sommet x_0 passé en argument, et renvoyant un tableau `t` codant l'arbre de parcours en profondeur associé de la manière suivante :
   - `t` est de longueur $|V|$;
   - `t.([x0) = x0;`
   - si x n'est pas accessible depuis x_0, `t.(x) = - 1;`
   - si x a été exploré depuis y, `t.(x) = y`.

```
arbre_dfs : graphe -> sommet -> sommet array
```

2. Écrire une fonction `arbre_bfs` ayant les mêmes spécifications que `arbre_dfs` (sauf bien sûr que l'on renverra un arbre de parcours en largeur).
3. Écrire une fonction `chemin` qui prend en entrée un arbre enraciné en x_0 comme ci-dessus et un sommet x, et renvoie un chemin `Some [x0; ... ; x]` s'il en existe un, `None` sinon.

   Remarque
   Il est inutile de passer x_0 en argument : c'est le seul indice pour lequel `t.(i) = i`.

```
chemin : sommet array -> sommet -> sommet list option
```

4. Que peut-on dire du chemin renvoyé par la fonction `chemin` si l'arbre utilisé est issu de la fonction `arbre_bfs` ?

3 Variantes des parcours

Exercice XXXVIII.8 – Parcours en profondeur itératif

p. 667

1. Compléter cette fonction pour qu'elle réalise un parcours en profondeur de `g` à partir de `i` :

```
let dfs_pile pre g i =
  let visites = Array.make g.nb_sommets false in
  let rec traite pile = match pile with
    | [] -> ...
    | x :: xs when not visites.(x) -> ...
    | x :: xs -> ... in
  ...
```

   La fonction `traite` devra être récursive terminale. On évitera d'utiliser `@` qui n'est pas terminal[a], mais on n'hésitera pas à faire appel à la fonction `List.rev_append`, qui l'est.
   Il n'y a pas cette fois d'argument `post`, car il n'y a pas de moyen évident de savoir quand le traitement d'un nœud est terminé.
2. Décrire l'évolution de `pile` au cours de l'appel `dfs_pile g1 0`.
3. Justifier que la taille de `pile` (et donc la complexité spatiale de `dfs_pile`) peut être de l'ordre de $|E|$.

   Remarque
   Le remède semble évident mais ne l'est pas tant que ça : *cf* exercices XXXVIII.9 et 15.19.
4. Une fonction récursive terminale peut très facilement être transformée en une fonction non récursive[b]. Il suffit ici d'utiliser une pile impérative à la place d'une pile fonctionnelle (*i.e.* d'une liste).
   On pourrait réaliser une pile impérative à l'aide d'une `list ref`, mais le plus simple est d'utiliser le module `Stack` :

```
Stack.create : unit -> 'a Stack.t (* crée une pile vide *)
Stack.is_empty : 'a Stack.t -> bool
Stack.pop : 'a Stack.t -> 'a
Stack.push : 'a -> 'a Stack.t -> unit
```

   Écrire une fonction `dfs_it` ayant les mêmes spécifications (et donc les mêmes arguments) que `dfs_pile` mais purement itérative.

a. On pourrait presque systématiquement le faire puisque la profondeur de récursion serait majorée par le degré maximal du graphe, qui est rarement gigantesque.
b. C'est tellement facile que le compilateur le fait automatiquement.

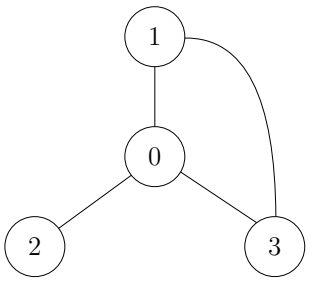
Exercice XXXVIII.9 – Parcours en pseudo-profondeur

p. 669

1. Modifier la fonction `dfs_it` de l'exercice XXXVIII.8 de manière à ce qu'un sommet soit rajouté au plus une fois sur la pile. On appellera la fonction obtenue `pseudo_dfs`. Que remarque-t-on par rapport au parcours en largeur écrit à l'exercice XXXVIII.3 ?

   Ce parcours est très couramment utilisé car il est rapide, peu gourmand en mémoire, et ne risque pas de résulter en un `stack overflow`. Cependant, il ne s'agit pas d'un vrai parcours en profondeur, comme le montrent les questions suivantes. Souvent, on souhaite juste parcourir le graphe et cela ne nous dérange nullement ; parfois, les propriétés du parcours en profondeur sont cruciales et le parcours en « pseudo-profondeur » ne convient pas.

2. On fait un parcours du graphe représenté ci-dessous à partir du nœud 0 (et l'on suppose que les listes de voisins sont stockées dans l'ordre croissant). Dans quel ordre les nœuds sont-ils traités (un nœud `x` est traité quand on appelle `pre x`) par `pseudo_dfs` ? Est-ce le même que pour `dfs_pile` ? que pour `dfs` ?



3. On se limite aux graphes non orientés pour simplifier. Montrer que l'arbre associé au parcours en pseudo-profondeur d'un graphe G depuis un sommet x est un arbre de parcours en profondeur si et seulement si la composante connexe de x dans G est un arbre.

Solutions

Correction de l'exercice XXXVIII.1 page 658

Il faut écrire des variantes de cette fonction jusqu'à ce que ça vous semble aussi naturel que de calculer la longueur d'une liste.

```
let dfs pre post g i =
  let n = Array.length g in
  let visites = Array.make n false in
  let rec visite j =
    if not visites.(j) then begin
        visites.(j) <- true;
        pre j;
        List.iter visite g.(j);
        post j
      end in
  visite i
```

Correction de l'exercice XXXVIII.2 page 658

```
dfs ouvre ferme g0 0
```

```
Ouverture 0
Ouverture 1
Ouverture 2
Fermeture 2
Ouverture 3
Ouverture 5
Ouverture 10
Ouverture 11
```

```
Fermeture 11
Fermeture 10
Fermeture 5
Fermeture 3
Ouverture 4
Fermeture 4
Fermeture 1
Fermeture 0
```

```
dfs ouvre ferme g1 5
```

```
Ouverture 5
Ouverture 2
Ouverture 1
Ouverture 0
Ouverture 4
Fermeture 4
Fermeture 0
Ouverture 7
Ouverture 8
```

```
Ouverture 3
Ouverture 6
Fermeture 6
Fermeture 3
Fermeture 8
Fermeture 7
Fermeture 1
Fermeture 2
Fermeture 5
```

Correction de l'exercice XXXVIII.3 page 659

1.
```
let bfs pre g initial =
  let vus = Array.make (Array.length g) false in
  let file = Queue.create () in
  let ajoute x =
    if not vus.(x) then begin
        pre x;
        vus.(x) <- true;
        Queue.push x file
      end in
  ajoute initial;
  while not (Queue.is_empty file) do
    let x = Queue.pop file in
    List.iter ajoute g.(x);
  done
```

2. Pour `bfs ouvre g0 0`, on doit obtenir dans l'ordre 0,1,2,3,4,5,10,11. Pour `bfs ouvre g1 5`, on doit obtenir 5,2,7,8,1,3,6,0,4.
3. Une file impérative (de capacité fixée à la construction) peut être réalisée par un tableau circulaire, par une structure contenant un pointeur vers chaque extrémité, d'une liste simplement chaînée mutable, par une liste doublement chaînée... Une file fonctionnelle peut être réalisée par un couple de listes.

Correction de l'exercice XXXVIII.4 page 659

```
let bfs_avec_frontiere pre g x0 =
  let vus = Array.make (Array.length g) false in
  vus.(x0) <- true;
  let rec ajoute voisins nouveaux =
    match voisins with
    | [] -> nouveaux
    | x :: xs when vus.(x) -> ajoute xs nouveaux
    | x :: xs ->
      vus.(x) <- true;
      pre x;
      ajoute xs (x :: nouveaux) in
  let rec loop frontiere nouveaux =
    match (frontiere, nouveaux) with
    | [], [] -> ()
    | [], _ -> loop nouveaux []
    | x :: xs, _ -> loop xs (ajoute g.(x) nouveaux) in
  loop [x0] []
```

L'appel `ajoute voisins nouveaux` marque les sommets non encore visités de `voisins` et les « ajoute » à `nouveaux` (c'est-à-dire renvoie la concaténation de ces sommets non visités avec `nouveaux`).
Dans la boucle principale (fonction `loop`), la liste `frontiere` représente les sommets vus mais pas encore traités à distance k de `x0` et la liste `nouveaux` les sommets vus situés à distance $k+1$.

Correction de l'exercice XXXVIII.5 page 659

1.

```
let accessible g i j =
  let vus = Array.make (Array.length g) false in
  let rec explore v =
    if not vus.(v) then begin
      vus.(v) <- true;
      List.iter explore g.(v)
    end in
  explore i;
  vus.(j)
```

2. Pour s'arrêter dès que possible, le plus simple est d'utiliser une exception :

```
exception Trouve

let accessible_exn g i j =
  let vus = Array.make (Array.length g) false in
  let rec explore v =
    if v = j then raise Trouve;
    if not vus.(v) then begin
      vus.(v) <- true;
      List.iter explore g.(v)
    end in
  try
    explore i;
    false
  with
  | Trouve -> true
```

On peut bien sûr faire sans, comme dans la version ci-dessous. Deux remarques qui peuvent aider à la compréhension de ce code :

- on utilise de manière cruciale le fait que la branche droite d'un « ou » n'est évaluée que si la branche gauche est fausse – au besoin, vous pouvez ré-écrire le code avec des `if..then..else` pour mieux comprendre ce qui se passe ;
- les fonctions `cherche` et `traite` sont *mutuellement récursives*, elles doivent donc être définies simultanément par un `let rec cherche v = ... and traite = ...`.

```
let accessible_bis g i j =
  let vus = Array.make (Array.length g) false in
  let rec cherche v =
    (v = j) || (vus.(v) <- true; traite g.(v))
  and traite = function
    | [] -> false
    | x :: xs when vus.(x) -> traite xs
    | x :: xs -> cherche x || traite xs in
  cherche i
```

Correction de l'exercice XXXVIII.6 page 660

1.
```
let tab_composantes graphe =
  let n = Array.length graphe in
  let t = Array.make n (-1) in
  let rec assigne i sommet =
    if t.(sommet) = -1 then begin
      t.(sommet) <- i;
      List.iter (assigne i) graphe.(sommet)
    end in
  for i = 0 to n - 1 do
    assigne i i (* sans effet si i est déjà dans une composante *)
  done;
  t
```

L'appel `assigne i s` effectue un parcours en profondeur à partir du sommet `s`, en affectant tous les sommets rencontrés à la composante `i`.

2.
```
let liste_composantes graphe =
  let n = Array.length graphe in
  let vus = Array.make n false in
  (* la liste des composantes *)
  let composantes = ref [] in
  (* la composante actuelle *)
  let c = ref [] in
  let rec explore i =
    if not vus.(i) then begin
      vus.(i) <- true;
      c := i :: !c;
      List.iter explore graphe.(i)
    end in
  for i = 0 to n - 1 do
    if not vus.(i) then begin
      c := [];
      explore i;
      composantes := !c :: !composantes
    end
  done;
  !composantes
```

Correction de l'exercice XXXVIII.7 page 660

1.
```
let arbre_dfs g x0 =
  let parent = Array.make (Array.length g) (-1) in
  let rec explore u =
    let f v =
      if parent.(v) = -1 then (parent.(v) <- u; explore v) in
    List.iter f g.(u) in
  parent.(x0) <- x0;
  explore x0;
  parent
```

2.

```
let arbre_bfs g x0 =
  let parent = Array.make (Array.length g) (-1) in
  let file = Queue.create () in
  let ajoute pere x =
    if parent.(x) = -1 then begin
        parent.(x) <- pere;
        Queue.push x file
      end in
  ajoute x0 x0;
  while not (Queue.is_empty file) do
    let x = Queue.pop file in
    List.iter (ajoute x) g.(x);
  done;
  parent
```

3.

```
let chemin parent i =
  let rec aux i acc =
    if parent.(i) = i then i :: acc
    else aux parent.(i) (i :: acc) in
  if parent.(i) = -1 then None
  else Some (aux i [])
```

4. Si l'on utilise un arbre de parcours en largeur, le chemin renvoyé sera systématiquement un plus court chemin.

Correction de l'exercice XXXVIII.8 page 661

1.

```
let dfs_pile pre g i =
  let visites = Array.make (Array.length g) false in
  let rec traite = function
    | [] -> ()
    | x :: xs when not visites.(x) ->
       visites.(x) <- true;
       pre x;
       traite (List.rev_append g.(x) xs)
    | x :: xs -> traite xs in
  traite [i]
```

2. Pour bien pouvoir observer le comportement de la fonction, on affiche un message `"Traitement de ..."` quand on dépile un sommet pour la première fois et un message `"Déjà traité..."` quand on dépile un sommet déjà traité. On affiche la pile à chaque fois qu'on y ajoute des sommets.

```
let dfs_pile_avec_affichage g i =
  let visites = Array.make (Array.length g) false in
  let rec traite = function
    | [] -> ()
    | x :: xs when not visites.(x) ->
      visites.(x) <- true;
      let nv_pile = List.rev_append g.(x) xs in
      Printf.printf "Traitement de %d\n" x;
      Printf.printf "État de la pile : ";
      affiche nv_pile;
      traite (List.rev_append g.(x) xs)
    | x :: xs -> Printf.printf "Déjà traité : %d\n" x; traite xs in
  traite [i]
```

```
dfs_pile_avec_affichage g1 0
```

```
Traitement de 0
État de la pile : 4 1
Traitement de 4
État de la pile : 1 0 1
Traitement de 1
État de la pile : 7 4 2 0 0 1
Traitement de 7
État de la pile : 8 5 1 4 2 0 0
↪  1
Traitement de 8
État de la pile : 7 6 5 3 5 1 4
↪  2 0 0 1
Déjà traité : 7
Traitement de 6
État de la pile : 8 3 5 3 5 1 4
↪  2 0 0 1
Déjà traité : 8
Traitement de 3
État de la pile : 8 6 5 3 5 1 4
↪  2 0 0 1
Déjà traité : 8
```

```
Déjà traité : 6
Traitement de 5
État de la pile : 8 7 2 3 5 1 4
↪  2 0 0 1
Déjà traité : 8
Déjà traité : 7
Traitement de 2
État de la pile : 5 1 3 5 1 4 2
↪  0 0 1
Déjà traité : 5
Déjà traité : 1
Déjà traité : 3
Déjà traité : 5
Déjà traité : 1
Déjà traité : 4
Déjà traité : 2
Déjà traité : 0
Déjà traité : 0
Déjà traité : 1
```

3. Imaginons qu'on parcourt un graphe complet à n sommets, avec des listes d'adjacence croissantes. On note $x_1, \dots, x_n$ les sommets dans l'ordre de leur traitement (*i.e.* de leur ajout à `vus`). Quand on traite x_i, on ajoute ses $n-1$ voisins à la pile ; avant de traiter x_{i+1}, on aura éliminé au plus $i-1$ sommets de la pile (les sommets $x_1, \dots, x_{i-1}$ qui ont déjà été traités). Juste après avoir traité x_n, la pile contient donc au moins $n(n-1) - \sum_{i=1}^{n}(i-1) = \frac{n(n-1)}{2}$ sommets. La complexité spatiale peut donc être de l'ordre de n^2 (et pas plus, puisque la complexité temporelle est en $O(n+p) = O(n^2)$).

   Remarque

   Le fait que la complexité *temporelle* soit en n^2 pour un graphe complet est normal (ce serait le cas avec n'importe quel parcours, car la taille du graphe est de cet ordre). La complexité *spatiale*, en revanche, serait en $O(n)$ pour un parcours en profondeur récursif (ou pour un parcours en largeur).

4.

```
let dfs_it pre g i =
  let visites = Array.make (Array.length g) false in
  let p = Stack.create () in
  Stack.push i p;
  while not (Stack.is_empty p) do
    let x = Stack.pop p in
    if not visites.(x) then begin
        visites.(x) <- true;
        pre x;
        List.iter (fun v -> Stack.push v p) g.(x);
      end
  done
```

Correction de l'exercice XXXVIII.9 page 662

1. On obtient exactement la même fonction qu'à l'exercice XXXVIII.3, sauf que l'on a remplacé la file par une pile.

```
let pseudo_dfs pre g x0 =
  let vus = Array.make (Array.length g) false in
  let pile = Stack.create () in
  let rec ajoute x =
    if not vus.(x) then begin
      vus.(x) <- true;
      Stack.push x pile
  end in
  ajoute x0;
  while not (Stack.is_empty pile) do
    let x = Stack.pop pile in
    pre x;
    List.iter ajoute g.(x)
  done
```

2. Pour `pseudo_dfs`, les sommets seront traités dans l'ordre $0, 3, 2, 1$ car 1 n'est pas rajouté au sommet de la pile quand on traite 3 (il a déjà été marqué pendant le traitement de 0). Cet ordre est impossible pour un parcours en profondeur, qui impose de terminer l'exploration à partir de 3 avant de commencer celle à partir de 2.
3. Supposons que la composante connexe de x ne soit pas un arbre. Dans ce cas, tout arbre de parcours en profondeur à partir de x doit nécessairement comporter au moins un arc arrière reliant un sommet t à l'un de ses ancêtres y (autre que son père z). Il est impossible d'obtenir un tel arbre avec `pseudo_dfs` : en effet, t aurait été marqué lors du traitement de y (au plus tard), et n'aurait donc pas été ajouté à la pile lors du traitement de z.

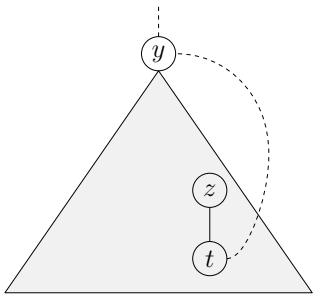


Figure XXXVIII.5 – Arc arrière, impossible avec un parcours en pseudo-profondeur.