Informatique *Cours*
Algorithmique des graphes - Pt1 Ipesup

Objectifs

À l'issue de cette leçon, l'étudiant doit être capable de :
- Comprendre et implémenter le parcours en profondeur (`dfs`) sur un graphe.
- Déterminer les sommets atteignables, tester l'existence d'un chemin et analyser la complexité $O(V + E)$.
- Calculer les composantes connexes d'un graphe non orienté.
- Exploiter l'ordre postfixe et l'appliquer au tri topologique des graphes acycliques.

Algorithmique des graphes

Nous présentons dans cette section plusieurs algorithmes sur les graphes, parmi les plus fondamentaux. Nous nous limitons à du code OCaml, pour deux raisons principales : d'une part, la possibilité d'écrire des fonctions locales et anonymes nous permet un code compact ; d'autre part, nous profitons avantageusement de l'existence de structures de données polymorphes, comme par exemple des files de priorité. Cependant, il serait tout à fait possible de proposer du code C pour ces mêmes algorithmes, au prix d'un code un peu plus verbeux et de structures de données adaptées qu'il faudrait commencer par se donner.

Parmi les questions auxquelles on veut pouvoir répondre pour un graphe donné, les suivantes sont naturelles :

- existe-t-il un chemin de u à v ?
- quels sont tous les sommets atteignables depuis u ?
- existe-t-il un cycle partant de u ?
- quel est le plus court chemin de u à v ?

Nous allons voir que l'on peut y répondre avec des algorithmes qui s'écrivent très simplement et qui sont fondamentaux : le parcours en profondeur et le parcours en largeur. C'est par cela que nous allons commencer, avant de voir quelques algorithmes plus complexes.

1 Parcours en profondeur

Le parcours en profondeur (en anglais *depth-first search* ou DFS) est un algorithme fondamental sur les graphes, avec de très nombreuses applications. Il consiste en une exploration du graphe à partir d'un sommet donné, que l'on va appeler la source.

Tant qu'il est possible de progresser en suivant un arc, on le fait, et sinon on fait machine arrière pour considérer d'autres arcs. Et pour éviter de reconsidérer plusieurs fois les mêmes sommets, et en particulier de tomber dans un cycle, on marque les sommets qui ont déjà été visités. C'est aussi simple que cela !

Le programme suivante contient le code OCaml d'une fonction `dfs` qui réalise un parcours en profondeur à partir du sommet source.

Programme - parcours en profondeur

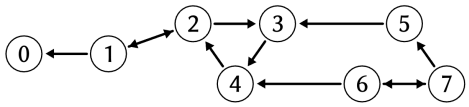
```
1 let dfs (g: digraph) (source: int) : bool array =
2   let visited = Array.make (size g) false in
3   let rec dfs v =
4     if not visited.(v) then (
5       visited.(v) <- true;
6       List.iter dfs (succ g v);
7     ) in
8   dfs source;
9   visited
```

Les sommets visités par le parcours sont marqués dans le tableau de booléens `visited`, qui est renvoyé au final. La fonction récursive locale `dfs` réalise le parcours. Elle reçoit un sommet v en argument. S'il a déjà été visité, on ne fait rien. Sinon, on marque le sommet comme étant visité (ligne 5), puis on parcourt récursivement tous ses voisins (ligne 6). On lance le parcours en appelant `dfs` sur le sommet source (ligne 8).

Il est crucial de marquer le sommet visité *avant* d'explorer récursivement ses voisins. Sans cela, on se retrouverait à parcourir éternellement tout cycle du graphe qui est accessible depuis la source.

Exemple

Considérons le graphe suivant, sur lequel on lance un parcours en profondeur à partir du sommet 7.



En supposant que la fonction `succ` renvoie systématiquement les voisins par ordre croissant, on a les appels récursifs suivants à `dfs` :

```
dfs 7
| dfs 5
| | dfs 3
| | | dfs 4
| | | | dfs 2
| | | | | dfs 1
| | | | | | dfs 0
| | | | | dfs 2 déjà vu
| | | | dfs 3 déjà vu
| dfs 6
| | dfs 4 déjà vu
| | dfs 7 déjà vu
```

Comme on le constate, la fonction `dfs` est parfois appelée plusieurs fois sur le même sommet. La première fois, on marque le sommet visité et on traite ses voisins. Les fois suivantes, en revanche, on ne fait rien, ce qui est indiqué ci-dessus avec « déjà vu ».

Dans certains cas, on retombe sur un sommet en cours de traitement. C'est le cas ici pour 2, 3 et 7, et cela manifeste la présence d'un cycle dans le graphe. Dans d'autres cas, en revanche, on retombe sur un sommet par un chemin parallèle. C'est le cas ici pour 4, qu'on avait déjà atteint par le chemin $7 \to 5 \to 3 \to 4$. C'est ce qu'on a illustré à droite avec les arcs en pointillés.

Une propriété fondamentale du parcours en profondeur est qu'il visite exactement les sommets atteignables depuis la source, avec une complexité optimale.

Propriété

Un appel à `dfs g source` détermine exactement l'ensemble des sommets accessibles depuis le sommet source, c'est-à-dire les sommets v pour lesquels il existe un chemin source $\to^* v$.

Démonstration. Commençons par noter que l'appel à `dfs source` termine. En effet, chaque appel à `dfs v` termine immédiatement ou fait diminuer strictement le nombre de sommets non marqués dans `visited`.
Montrons maintenant que, après l'appel à `dfs u`, si $u \to^* v$ alors le sommet v est marqué. On raisonne par récurrence sur la longueur du chemin.

- Pour une longueur 0, alors $v = u$ et c'est immédiat.
- Pour une longueur $n > 0$, on a $u \to^{n-1} w \to v$. Par hypothèse de récurrence, le sommet w est marqué, ce qui veut dire que `dfs w` a été appelée. Dès lors, l'arc $w \to v$ a été examiné et `dfs v` a été appelée. Par conséquence, le sommet v est marqué.

Il faut ensuite montrer la réciproque, à savoir que seuls des sommets atteignables sont marqués. Montrons que l'appel à `dfs u` ne marque que des sommets v tels que $u \to^* v$.
Cette fois, on le montre par récurrence sur le nombre d'appels à `dfs`.

- Pour un unique appel, `dfs u` marque le sommet u (éventuellement) et on a bien $u \to^* u$.
- Sinon, `dfs u` appelle `dfs w` avec $u \to w$. Par hypothèse de récurrence, `dfs w` ne marque que des sommets tels que $w \to^* v$, et donc uniquement des sommets tels que $u \to^* v$.

On a donc bien montré que l'appel initial à `dfs source` marque exactement les sommets atteignables depuis source. □

On a donc répondu à la question « quels sont tous les sommets atteignables depuis u ? ». En particulier, on répond également à la question « existe-t-il un chemin de u à v ? » en lançant un parcours en profondeur à partir de u pour vérifier ensuite que v a été atteint.

```
let exists_path g u v =
  (dfs g u).(v)
```

Bien entendu, on pourrait s'arrêter dès que v est atteint. C'est là une modification très simple du parcours en profondeur. La complexité reste cependant la même dans le pire des cas : il faut parfois explorer tous les sommets atteignables à partir de u pour déterminer s'il existe un chemin jusqu'à v.

Notons par ailleurs que le programme se contente de déterminer l'existence d'un chemin entre la source et tout autre sommet, mais il ne renvoie pas de tel chemin lorsqu'il existe.

Il est cependant simple de conserver, pour chaque sommet, l'arc qui a permis de l'atteindre pendant le parcours et d'exploiter ensuite cette information pour reconstruire le chemin si on le souhaite.

Complexité. La fonction `dfs` commence par construire un tableau de taille V, ce qui coûte $\Theta(V)$ en temps. Ensuite, il faut compter le coût des appels à `dfs`. Lorsque `dfs` est appelée sur un sommet déjà visité, ce coût est constant. Sinon, le coût propre de l'appel (i.e., hors des appels récursifs) est proportionnel au nombre de successeurs de v. Or, chaque arc $v \to w$ n'est considéré qu'au plus une fois, la première fois que le sommet v est visité. Dès lors, la complexité est en $O(E)$. Bien entendu, la complexité peut être bien moindre que E si peu de sommets sont atteignables depuis la source. Au total, on a donc une complexité en $O(V+E)$. Elle est optimale, car on peut être amené à parcourir l'intégralité du graphe.

La complexité en espace est au moins V, la taille du tableau `visited`. À cela, il faut ajouter la taille occupée sur la pile d'appels par les appels imbriqués à la fonction `dfs`. Cela peut être aussi grand que $\Theta(V)$ pour un chemin incluant tous les sommets du graphe. On a donc au total une complexité en espace $\Theta(V)$.

En particulier, la fonction `dfs` est susceptible de faire déborder la pile d'appels sur de très longs chemins. Le cas le plus simple est celui d'un graphe totalement linéaire

$$\text{source} \to v_1 \to v_2 \to v_3 \to \cdots \to v_n$$

avec plusieurs dizaines de milliers de sommets.

Graphe non orienté

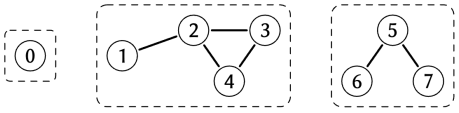
Il est important de comprendre que le programme montré fonctionne tout aussi bien, sans changement, sur un graphe non orienté avec la convention que nous avons choisie pour la représentation d'un graphe non orienté. Avec la même complexité $O(V+E)$, on détermine ainsi les sommets atteignables à partir d'un sommet donné dans un graphe non orienté.

1.1 Composantes connexes d'un graphe non orienté

Dans un graphe non orienté, une *composante connexe* est un sous-ensemble de sommets qui sont connectés deux à deux, et qui est maximal pour l'inclusion.

Exemple

Voici un exemple de graphe non orienté, avec trois composantes connexes identifiées par des pointillés :



On note que les composantes connexes forment une partition de l'ensemble des sommets. En effet, un sommet ne peut appartenir à deux composantes, sans quoi les sommets de ces deux composantes seraient tous connectés en passant par ce sommet.

On se propose de calculer les composantes connexes sous la forme d'un tableau donnant, pour chaque sommet, le numéro de sa composante connexe. S'il y a N composantes, elles

sont numérotées $0, 1, \ldots, N-1$, dans un ordre arbitraire. Dans le cas du graphe ci-dessus, une réponse possible est le tableau

0	1	1	1	1	2	2	2

mais toute autre permutation de $0, 1, 2$ dans ce tableau serait également valable.

Le parcours en profondeur permet de déterminer très facilement les composantes connexes d'un graphe non orienté. En effet, si on lance un parcours en profondeur à partir d'un sommet arbitraire, alors il va visiter exactement la composante de ce sommet. C'est la propriété du parcours en profondeur que nous avons établie plus haut. Pendant ce parcours, il suffit de marquer les sommets visités avec le numéro 0 (première composante). Puis, s'il reste des sommets non visités, on relance un deuxième parcours en profondeur à partir d'un sommet non visité, ce qui va déterminer une deuxième composante. Et ainsi de suite.

Le programme suivante contient un code OCaml qui réalise cette idée.

Programme - composantes connexes d'un graphe non orienté

```
1  let components (g: graph) : int * int array =
2    let n = size g in
3    let nc = ref 0 in
4    let num = Array.make n 0 in
5    let visited = Array.make n false in
6    let rec dfs v =
7      if not visited.(v) then (
8        visited.(v) <- true;
9        num.(v) <- !nc;
10       List.iter dfs (succ g v)
11     ) in
12   let component v =
13     if not visited.(v) then (dfs v; incr nc) in
14   for v = 0 to n - 1 do component v done;
15   !nc, num
```

La référence `nc` (ligne 3) compte les composantes et le tableau `num` (ligne 4) est la numérotation qui sera renvoyée au final (ligne 15). Le tableau `visited` (ligne 5) marque les sommets déjà visités par le parcours en profondeur. Les lignes 12–14 lancent le parcours en profondeur sur tous les sommets qui ne sont pas déjà marqués. Le parcours en profondeur est réalisé par la fonction récursive `dfs` ligne 6. Il est tout à fait analogue à celui du programme precedente. Le seul ajout est la ligne 9, qui affecte au sommet v le numéro de sa composante. ‘

Complexité La complexité de ce programme est $O(V + E)$, car chaque arc est examiné une seule fois, à savoir lorsque `dfs` est appelée sur son sommet source la première fois. Quand on dit « chaque arc » ici, on entend qu'il y a deux arcs entre deux sommets connectés, un arc $u \to v$ et un arc $v \to u$. Le premier sera examiné lorsque u sera visité pour la première fois et le second sera examiné lorsque v sera visité pour la première fois.

Le calcul des composantes connexes doit, dans le pire des cas, examiner tous les arcs du graphe, car un seul arc peut modifier l'ensemble des composantes connexes. Par ailleurs, il faut examiner tous les sommets car on associe à chaque sommet un numéro de composante.

Dès lors, la complexité de notre programme est optimale.

1.2 Ordre postfixe d'un parcours en profondeur

Lorsque l'on réalise le parcours en profondeur d'un graphe, on peut noter dans quel ordre se terminent les visites des sommets. Le programme suivante contient le code OCaml d'une fonction `post_order` qui réalise cette idée.

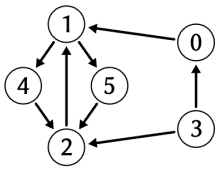
Programme - ordre postfixe d'un parcours en profondeur

```
let post_order (g: digraph) : int list =
  let visited = Array.make g.nbv false in
  let order = ref [] in
  let rec dfs v =
    if not visited.(v) then (
      visited.(v) <- true;
      List.iter dfs (succ g v);
      order := v :: !order
    ) in
  for v = 0 to g.nbv - 1 do dfs v done;
  !order
```

C'est exactement un parcours en profondeur, où la seule modification est la ligne `order := v :: !order` qui ajoute le sommet v au début de la liste `order` une fois que sa visite est terminée. Cette liste est renvoyée au final, une fois que le parcours en profondeur a visité tous les sommets.

Exemple

Examinons l'exécution de ce programme sur un exemple.




```
dfs 0
|  dfs 1
|  |  dfs 5
|  |  |  dfs 2
|  |  |  |  dfs 1 déjà vu
|  |  |  2 terminé
|  |  5 terminé
|  |  dfs 4
|  |  |  dfs 2 déjà vu
|  |  4 terminé
⋮
```

```
⋮
|  1 terminé
0 terminé
dfs 1 déjà vu
dfs 2 déjà vu
dfs 3
|  dfs 0 déjà vu
|  dfs 2 déjà vu
3 terminé
dfs 4 déjà vu
dfs 5 déjà vu
```

Au final, la liste renvoyée par `post_order` est donc `[3, 0, 1, 4, 5, 2]`.
Il est important de noter que l'ordre dans lequel les voisins d'un sommet sont visités, de même que l'ordre dans lequel on parcourt l'ensemble des sommets pour appeler `dfs` initialement, donnera un résultat différent. Ainsi, on a supposé ici que les deux voisins 5 et 4 du sommet 1 étaient examinés dans cet ordre. Le résultat final aurait été différent si on avait lancer `dfs` d'abord sur 4 puis sur 5.
De même, le résultat final aurait été différent si la boucle `for` avait plutôt parcouru les sommets par ordre décroissant.

1.3 Tri topologique

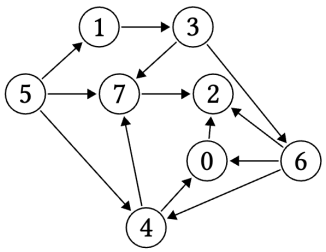
Un graphe orienté permet notamment de modéliser un problème d'*ordonnancement* : un arc $u \to v$ dans le graphe modélise le fait que la tâche u doit être effectuée avant la tâche v. Dès lors, un problème naturel, et utile en pratique, consiste à déterminer s'il existe un ordre dans lequel les tâches peuvent être effectuées. Il est clair que, si le graphe contient un cycle, alors il n'existe pas de solution. Dans le cas contraire, c'est-à-dire lorsque le graphe est acyclique, nous allons montrer qu'il existe toujours une solution. On appelle cela un *tri topologique*.

Définition - tri topologique

Pour un graphe orienté acyclique, un *tri topologique* est une liste ordonnée de ses sommets telle que, pour tout arc $u \to v$ dans le graphe, le sommet u apparaît avant le sommet v dans la liste.

Exemple

Considérons par exemple le graphe orienté acyclique suivant :



Alors, la liste 5, 1, 3, 6, 4, 7, 0, 2 est un tri topologique de ce graphe. Mais la liste 5, 1, 3, 6, 4, 0, 7, 2 en est un également. Il n'y a pas nécessairement unicité du tri topologique.

Il se trouve que nous avons déjà une solution au problème du tri topologique, à savoir le programme écrit dans la section précédente. Montrons-le.

Propriété

Sur un graphe orienté acyclique, l'ordre postfixe renvoyé par le programme 1.2 est un tri topologique.

Démonstration. Soit $u \to v$ un arc du graphe et montrons que u apparaît avant v dans la liste renvoyée par la fonction `post_order`. Cela revient à montrer que `dfs u` termine après `dfs v`. Considérons le moment où `dfs u` est appelée pour la première fois.

- Si l'appel à `dfs v` a déjà été fait et est déjà terminé, alors v est déjà dans la liste et donc u se retrouvera bien avant.
- Si l'appel à `dfs v` est déclenché par cet appel à `dfs u`, directement ou indirectement, alors l'appel à `dfs v` terminera avant l'appel à `dfs u`, et là encore la propriété voulue sera établie.
- Enfin, le dernier cas de figure est un appel `dfs u` déclenché, directement ou indirectement, par un appel à `dfs v`. Mais dans ce cas, il existe un chemin de v à u, et donc un cycle, ce qui est impossible.

Si l'ordre postfixe détermine bien un tri topologique, il est important de noter que l'ordre postfixe est défini sur n'importe quel graphe orienté, y compris en présence de cycles.

2 Parcours en largeur

Le parcours en largeur (en anglais *breadth-first search* ou BFS), consiste à explorer le graphe « en cercles concentriques » en partant d'un sommet particulier s appelé la source. On parcourt d'abord les sommets situés à une distance d'un arc de s, puis les sommets situés à une distance de deux arcs de s, et ainsi de suite.

Pour réaliser le parcours en largeur, on va utiliser une *file* contenant des sommets. Au début de cette file, on trouve des sommets situés à distance d de la source, et à la fin de la file on trouve des sommets situés à distance $d+1$ de la source.

À chaque étape, on retire un sommet de la file et on examine ses voisins. Ceux d'entre eux qui ne sont pas encore visités sont, par définition, des sommets situés à distance $d+1$ de la source et on les ajoute donc à la file. Une fois qu'on aura examiné tous les sommets à distance d, on passera automatiquement aux sommets à distance $d+1$ et on commencera à ajouter des sommets à distance $d+2$ dans la file, et ainsi de suite. Initialement, la file contient uniquement le sommet source, à distance 0 de lui-même.

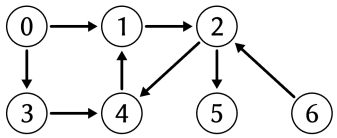
Le programme suivante contient le code OCaml d'une fonction `bfs` qui réalise un parcours en largeur à partir du sommet source. On utilise une file réalisée par un module `Queue` (voir cours File). Le tableau `dist` contient, pour chaque sommet atteint par le parcours, sa distance à la source. On l'initialise avec la valeur 0 pour la source et la valeur `max_int` pour tous les autres sommets. Lorsqu'un sommet v sort de la file (ligne 7), on examine ses voisins (ligne 9). Pour chaque voisin w non encore atteint, ce que l'on détermine avec le test `dist.(w) = max_int` (ligne 10), on lui affecte sa distance (ligne 11) et on l'ajoute à la file (ligne 12). Au final, on renvoie le tableau des distances.

Programme – parcours en largeur

```
1  let bfs (g: digraph) (source: int) : int array =
2    let dist = Array.make (size g) max_int in
3    dist.(source) <- 0;
4    let q = Queue.create () in
5    Queue.enqueue q source;
6    while not (Queue.is_empty q) do
7      let v = Queue.dequeue q in
8      let d = dist.(v) in
9      List.iter
10      (fun w -> if dist.(w) = max_int then (
11        dist.(w) <- d + 1;
12        Queue.enqueue q w))
13      (succ g v)
14   done;
15   dist
```

Exemple

Illustrons le parcours en largeur sur le graphe suivant, en prenant comme source le sommet 0.



action	⟵ **file** ⟵
initialisation, dist.(0) ← 0	0
on retire le sommet 0, dist.(1) ← 1, dist.(3) ← 1	1 3
on retire le sommet 1, dist.(2) ← 2	3 2
on retire le sommet 3, dist.(4) ← 2	2 4
on retire le sommet 2, dist.(5) ← 3	4 5
on retire le sommet 4	5
on retire le sommet 5	

Au final, on renvoie le tableau suivant :

0	1	2	3	4	5	6
0	1	2	1	2	3	∞

Le sommet 6 n'a pas été atteint par le parcours et il conserve donc la valeur ∞ (`max_int`) dans le tableau.

Il est intéressant de bien visualiser cette idée d'exploration en cercles concentriques réalisée par le parcours en largeur. La file contient une partie des sommets à distance d qu'il reste à examiner et une partie des sommets à distance $d+1$ déjà découverts. Lorsqu'un sommet v situé à distance d sort de la file, ses voisins w se répartissent en plusieurs catégories. Certains voisins sont à une distance $\leq d$. Dans ce cas, ils ont nécessairement déjà été atteints par le parcours (comme voisins d'un sommet à distance $< d$) et ils sont donc ignorés par le test ligne 10. Les autres voisins sont nécessairement situés à une distance $d+1$. Ils se répartissent eux-mêmes en deux sous-ensembles. Certains ont déjà été atteints par le parcours et se trouvent donc déjà dans la file. Ils sont ignorés par le test ligne 10. Enfin, les derniers sont des sommets que l'on atteint pour la première fois. On leur attribue la distance $d+1$ et on les ajoute à la file.

Comme le parcours en profondeur, le parcours en largeur détermine l'ensemble des sommets atteignables depuis la source, mais visités dans un ordre différent.

Propriété

Un appel à `bfs g source` détermine exactement l'ensemble des sommets accessibles depuis le sommet source, c'est-à-dire les sommets v pour lesquels il existe un chemin source $\longrightarrow^* v$, et il détermine pour chacun la longueur d'un plus court chemin en nombre d'arcs dans le tableau qui est renvoyé.

On peut donc déterminer si un sommet v est atteignable depuis la source en testant `dist.(v)` < `max_int` à l'issue du parcours en largeur. C'est là une alternative au parcours en profondeur.

Dit autrement, si `visited` est le tableau renvoyé par `dfs g s` et que `dist` est le tableau renvoyé par `bfs g s` alors, pour tout sommet v, on a

`visited.(v)` vaut true si et seulement si `dist.(v)` $<$ `max_int`

Mais on a une information plus précise avec le tableau `dist`, à savoir la distance exacte entre la source et v. Et, comme pour le parcours en profondeur, on peut maintenir facilement l'information qui permet de reconstruire, pour chaque sommet atteint par le parcours, le chemin depuis la source. On sait donc déterminer un plus court chemin entre deux sommets, lorsqu'ils sont reliés.

Complexité. La complexité est facile à déterminer. Chaque sommet est mis dans la file au plus une fois et donc examiné au plus une fois. Chaque arc est donc considéré au plus une fois, lorsque son origine est examinée. La complexité est donc $O(V + E)$, ce qui est optimal. La complexité en espace est $\Theta(V)$ car le tableau occupe un espace V et la file peut contenir jusqu'à $V - 1$ sommets dans le pire des cas. On a exactement la même complexité que le parcours en profondeur.