Arbres couvrants

MPSI/MP OPTION INFORMATIQUE - TP n° 3.4 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ expliquer le fonctionnement de l'algorithme de Prim
- ☞ expliquer le fonctionnement de l'algorithme de Kruskal
- ☞ programmer dans un style purement fonctionnel en OCaml

Dans tout ce TP, on considère des graphes pondérés non orientés représentés par des listes d'adjacence. L'objectif est d'exploiter au maximum le module `List` d'OCaml et de programmer dans un style purement fonctionnel, c'est à dire sans références.

A Algorithme de Prim

L'algorithme de Prim est un algorithme glouton optimal qui s'applique aux graphes pondérés connexes. L'algorithme part d'un sommet et fait croitre un arbre en choisissant un sommet dont la distance est la plus faible et qui n'appartient pas à l'arbre, garantissant ainsi l'absence de cycle.

Algorithme 1 Algorithme de Prim, arbre recouvrant

```
1 : Fonction PRIM(G = (V, E, w))
2 :     T ← ∅                                  ▷ la sortie : l'ensemble des arêtes de l'arbre recouvrant
3 :     S ← s un sommet quelconque de V
4 :     tant que S ≠ V répéter
5 :         (u, v) ← min(w(u, v), u ∈ S, v ∈ E)                          ▷ Choix glouton !
6 :         S ← S ∪ {v}
7 :         T ← T ∪ {(u, v)}
8 :     renvoyer T
```

A1. Montrer que l'algorithme de Prim termine.

Solution :

$$\begin{aligned} v : \mathbb{N} &\longrightarrow \mathbb{N} \\ |S| &\longmapsto |V| - |S| \end{aligned}$$

La fonction v est un varant de boucle pour la boucle tant que de l'algorithme de Prim. En effet, un graphe possède toujours un nombre fini de sommets, donc $|V|$ est un entier naturel. Au premier tout de boucle, $|S| = 1$ et à chaque tour le cardinal de S augmente de 1. Donc, v est

une fonction **strictement** décroissante qui atteint 0 lorsque la condition de boucle est validée. L'algorithme de Prim se termine.

A2. Montrer que l'algorithme de Prim est correct, c'est à dire qu'il calcule un arbre couvrant de poids minimal.

Solution : On peut dans un premier temps remarquer que l'algorithme construit bien un arbre car :

1. il effectue exactement $n-1$ tours de boucle et donc on a choisit $n-1$ arêtes parmi E,
2. il construit un graphe connexe puisqu'il choisit toujours d'ajouter un sommet voisin via une arête commune à l'ensemble des sommets visités S.
3. (redondant) il est acyclique car il choisit un nouveau sommet parmi ceux qui n'ont pas encore été visités.

Il nous reste à montrer que l'arbre construit est de poids minimal. On choisit le variant suivant : à chaque tour de boucle, le graphe (S,T) est un arbre couvrant de poids minimal.

À l'entrée de la boucle, on a $|S|=1$ et $T=\emptyset$, donc (S,T) est un arbre couvrant de poids minimal.

Plaçons nous maintenant à un tour de boucle quelconque et supposons que, à l'entrée de ce tout, (S,T) soit un arbre couvrant minimal. Soit $e=(a,b)$ l'arête que l'algorithme souhaite ajouter à T. Comme b n'appartient pas encore à S, $T\cup\{(a,b)\}$ est toujours un arbre qui couvre un sommet de plus : b. Comme (a,b) est l'arête de poids minimal parmi celles qui ne sont pas encore utilisées, alors $T\cup\{(a,b)\}$ est un arbre couvrant de poids minimal. Donc, à chaque tour de boucle, l'invariant est vérifié.

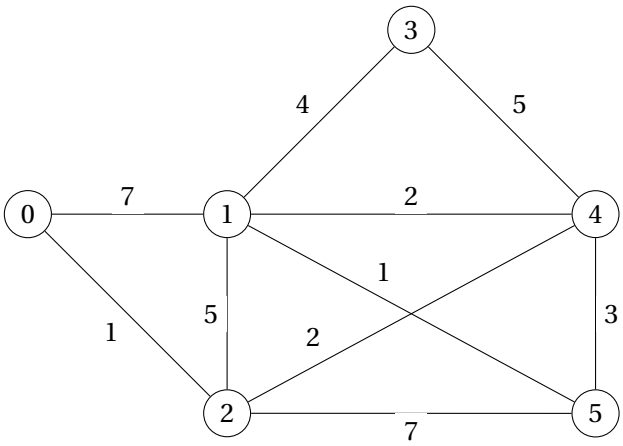
En particulier, il est vérifié à la fin de la boucle et on a alors $S=V$. T est donc un arbre couvrant G de poids minimal.

Pour faciliter la programmation, on utilise une représentation des graphes par liste d'adjacence. Ce choix n'est pas le fruit du hasard : on observe que les algorithmes de Prim ou de Kruskal construisent des arbres à partir des arêtes. Il est donc important de disposer des arêtes. On définit donc un type graphe comme suit :

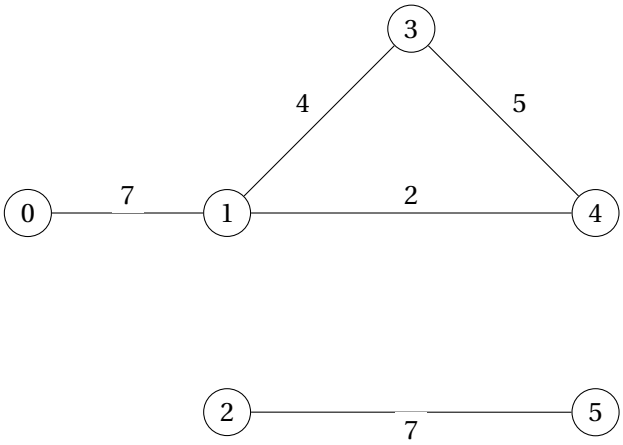
```
type list_graph = ((int * int) list) list;;

let lcg = [[(2,1); (1,7)];
           [(3,4); (4,2); (2,5); (0,7)];
           [(5,7); (4,2); (1,5); (0,1)];
           [(4,5); (1,4)];
           [(5,3); (3,5); (2,2); (1,2)];
           [(4,3); (2,7)]];;

let uclg = [[(1,7)];
            [(4,1); (3,4); (0,7)];
            [(5,7)];
            [(4,5); (1,4)];
            [(3,5); (1,2)];
            [(2,7)]];;
```

A3. Dessiner les deux graphes `lcg` et `uclg`. Quelle différence y-a-t-il entre les deux?

Solution :



Le second graphe n'est pas connexe :



A4. Écrire une fonction de signature `graph_order : 'int list -> int` dont le paramètre est un graphe sous la forme d'une liste d'adjacence qui renvoie l'ordre de ce graphe.

Solution :

```
let graph_order lg = List.length lg;;
```

A5. Écrire une fonction dont la signature est :
`make_triplets_list : (int * int)list list -> (int * int * int)list`
qui transforme un graphe sous la forme d'une liste d'adjacence en une liste de type `(i,j,w)`, où `i` et `j` sont des sommets et `w` le poids associé à l'arête `(i,j)`. Par exemple, le graphe `lcg` devient `[(0, 2, 1); (0, 1, 7); (1, 3, 4); (1, 4, 2); (1, 2, 5); (2, 5, 7); (2, 4, 2);(3, 4, 5); (4, 5, 3)]`.
Bien observer qu'on a enlevé les redondances : on considère que les arêtes sont dans les deux sens.

Solution :

```
let make_triplets_list lg =
  let filter triplets edges = triplets@List.filter_map (fun (i,j,w) ->
                  if not (List.mem (i,j,w) triplets)
                        && not (List.mem (j,i,w) triplets)
                  then Some (i,j,w)
                  else None
                  ) edges in
    List.fold_left filter []
      List.mapi (fun i edges ->  List.map (fun (j,w) -> (i,j,w)) edges) lg);
```

A6. L'algorithme de Prim termine sa boucle lorsque les ensemble S et V sont identiques. On représente ces ensembles par des listes `S` et `V` en OCaml. Écrire une fonction de signature
`same_list : int list -> int list -> bool`
qui renvoie `true` si deux listes contiennent exactement les mêmes éléments et `false` sinon.

Solution :

```
let same_list x y = (List.length x = List.length y)
                  &&(List.for_all (fun e -> List.mem e x) y)
                  &&(List.for_all (fun e -> List.mem e y) x);;
```

A7. Écrire une fonction de signature `compare_edges : int * int * int -> int * int * int -> int` qui compare les poids de deux triplets `(i,j,w1)` et `(k,l,w2)`. La fonction renvoie un entier positif si `w2` est plus grand que `w1`, 0 s'ils sont égaux et négatif sinon.

Solution :

```
  let compare_edges e1 e2 = let (_,_,w1) = e1 and (_,_,w2) = e2 in w1 - w2;;
```

A8. Écrire une fonction de signature
`sort_triplets : (int * int)list list -> (int * int * int)list`
qui prend en entrée un graphe sous la forme d'une liste d'adjacence et qui renvoie la liste des triplets `(i,j,w)` triés dans l'ordre croissant de poids.

Solution :

```
let sort_triplets lg = List.sort compare_edges (make_triplets_list lg);;
```

A9. Écrire une fonction de signature
`filter_triplet : int list -> int * int * int -> (int * int * int)`
Le premier paramètre d'entrée est une liste de sommet, le second un triplet `(i,j,w)`. La fonction renvoie des types optionnels :

1. `Some (i,j,w)` si $i \in S$ et $j \notin S$,
2. `Some (j,i,w)` si $j \in S$ et $i \notin S$,
3. `None` sinon.

L'objectif est d'utiliser ensuite cette fonction dans une expression de type `List.filter_map`. Cette fonction renvoie une liste qui n'est pas de la même taille que la liste d'entrée : le filtre écarte certaines valeurs. Ce filtre renvoie donc parfois un triplet, parfois rien.

Solution :

```
let filter_triplet s (u,v,w) =
  if (List.mem u s) && not (List.mem v s)
    then Some(u,v,w)
  else if (List.mem v s) && not (List.mem u s)
    then Some(v,u,w)
  else None;;
```

A10. Écrire une fonction de signature
`select_min_triplet : int list -> (int * int * int)list -> int * int * int`
Le premier paramètre est la liste des sommets S, le second la liste des triplets triés dans l'ordre croissant des poids. Cette fonction renvoie le triplet de poids le plus faible dont un des sommets est dans S.

Solution :

```
  let select_min_triplet s triplets =
      List.nth (List.filter_map (filter_triplet s) triplets) 0;;
```

A11. Écrire une fonction purement fonctionnelle de signature
`pure_prim : (int * int)list list -> (int * int * int)list`
qui implémente l'algorithme de Prim. Le paramètre d'entrée est un graphe sous la forme d'une liste d'adjacence. Cette fonction renvoie la liste des triplets qui constitue l'arbre de poids minimal. Par exemple pour le graphe `lcg`, la sortie vaut :
`[(1, 3, 4); (4, 5, 3); (4, 1, 2); (2, 4, 2); (0, 2, 1)]`

Solution :

```
let pure_prim lg =
  let n = List.length lg in
  let all_nodes = List.init n (fun i -> i) in
  let all_triplets = sort_triplets lg in
  let rec set_edge s tree =
    if same_list all_nodes s
    then tree
    else let (i,j,w) = select_min_triplet s all_triplets in
         set_edge (j::s) ((i,j,w)::tree)
  in set_edge [ 0 ] [];;  (* on pourrait choisir n'importe quel sommet ! *)
```

A12. Tester la fonction `pure_prim` sur le graphe `uclg`. Que se passe-t-il ? Est-ce normal ?

> **Solution :** Une exception est levée `Exception: Failure "nth".` car la liste des triplets est vide bien que S soit différent de V. C'est normal car l'algorithme de Prim ne fonctionne pas sur des graphes non connexes : étant donné qu'on part d'un sommet au hasard et qu'on parcourt tout ceux qui sont connectés, on ne peut pas appliquer cet algorithme à un graphe non connexe.

A13. Comment pourrait-on utiliser une file de priorités pour améliorer les performances de cet algorithme ?

B Algorithme de Kruskal

L'algorithme de Kruskal est un algorithme glouton optimal qui s'applique aux graphes pondérés. Le graphe peut ne pas être connexe et dans ce cas on obtient un forêt d'arbres recouvrants. Pour construire l'arbre et effectuer un choix d'arête, il ordonne les arêtes du graphe par ordre de pondération croissante.

Algorithme 2 Algorithme de Kruskal, arbre recouvrant

```
1 : Fonction KRUSKAL(G = (V, E, w))
2 :     T ← ∅                                  ▷ la sortie : l'ensemble des arêtes de l'arbre recouvrant
3 :     pour k de 1 à |E| répéter
4 :         e ← l'arête de pondération la plus faible de E          ▷ Choix glouton !
5 :         si (S, T ∪ {e}, w) est un graphe acyclique alors
6 :             T ← T ∪ {e}
7 :     renvoyer T
```

La principale difficulté de l'algorithme de Kruskal réside au niveau du test d'acyclicité : comment savoir si le nouvel arbre qu'on projette de construire est bien acyclique ? On pourrait utiliser un parcours en profondeur car dès qu'on trouve un sommet déjà découvert un cycle est détecté. Cependant ce n'est pas optimal au niveau de la complexité. C'est pourquoi il est préférable d'utiliser une structure de type Union-Find : c'est un structure simple et efficace qui permet de garder la trace des sommets qui sont connexes dans le graphe.

On se place dans le cadre du graphe `lcg` défini plus haut. C'est un graphe à six sommets. On va donc créer une structure Union-Find à partir de la liste `uf = [0,1,2,3,4,5]`. Chaque élément de la liste représente le numéro de l'ensemble connexe auquel appartient le sommet du graphe. Au début de l'algorithme, comme on n'a pas encore sélectionné d'arête, tous les sommets apparaissent déconnectés, chacun dans un ensemble différent. Donc le sommet 0 est dans l'ensemble 0, le sommet 1 dans l'ensemble 1…

Lorsqu'on connecte deux sommets par une arête, les sommets appartiennent au même ensemble connexe. On doit donc modifier `uf` et faire en sorte que les deux sommets portent le même numéro d'ensemble connexe. Par exemple, si on connecte le sommet 1 et le sommet 5, alors on aura `uf = [0,1,2,3,4,1]` ou bien `uf = [0,5,2,3,4,5]`.

Au fur et à mesure de l'algorithme, on va faire donc évoluer à la fois la liste des arêtes de l'arbre et la structure `uf`. À chaque tour de boucle, on choisit une arête et on vérifie que ses sommets n'appartiennent pas au même ensemble connexe, c'est à dire que les valeurs associées à ces sommets dans `uf` sont différentes. Par exemple, si on a `uf = [0,5,2,3,4,5]`, on ne pourra pas ajouter l'arête $(1,5)$.

B1. On dispose d'une liste `uf` de type Union-Find. Écrire une fonction de signature
`union : int -> int -> int list -> int list`
qui procède à l'union des ensembles de numéro `e1` et `e2` donnés en paramètres. Le troisième paramètre est la liste `uf`. La fonction renvoie une nouvelle liste Union-Find dans laquelle `e1` a remplacé `e2`. Par exemple, si `uf = [0,0,2,3,4,4]` et si on souhaite unir les ensembles de numéro 0 et 4, la fonction renverra `uf = [0,0,2,3,0,0]`.

Solution :

```
let union e1 e2 uf = List.map (fun v -> if v=e2 then e1 else v) uf;;
```

Le résultat de l'algorithme de Kruskal est une forêt d'arbre. On représente une forêt d'arbre comme les arbres précédemment, c'est à dire par une liste de triplets `(i,j,w)` où `i` et `j` sont des sommets et `w` le poids associé à l'arête `(i,j)`. La liste représente donc indifféremment un arbre ou une forêt. La seule différence est que, dans le cas d'une forêt, tous les sommets ne sont pas connexes.

Pour l'algoritme, on choisit un tuple de type `(uf, forest)` où `uf` est de type `int list` et `forest` de type `(int * int * int)list`. Au fur et à mesure de l'algorithme, la forêt s'épaissit, la structure `uf` s'homogénéise.

B2. Écrire une fonction de signature
`set_union_edge : int list * (int * int * 'int)list -> int * int * 'int -> int list * (int * int * 'int)list`
qui :

1. ajoute un triplet à `(uf, forest)` si l'arête ne crée pas un cycle,
2. ne modifie pas `(uf, forest)` sinon.

Les paramètres sont donc constitués d'un tupe `(uf, forest)` et d'un triplet. La fonction renvoie `(uf,forest)` dans tous les cas.

Solution :

```
let set_union_edge (uf, forest) (i,j,p) =
  let ui = List.nth uf i and uj = List.nth uf j in
    if ui = uj then (uf, forest) else (union ui uj uf, ((i,j,p)::forest));;
```

B3. Écrire une fonction purement fonctionnelle de signature
`pure_kruskal : (int * int)list list -> (int * int * int)list`
qui implémente l'algorithme de Kruskal. La fonction renvoie la forêt sous la forme d'une liste de triplets `forest`.

Solution :

```
let pure_kruskal lg =
  let all_triplets = sort_triplets lg and n = List.length lg in
  snd(List.fold_left set_union_edge ((List.init n (fun i -> i)),[])
      all_triplets)
  ;;
```

```
(* on fait évoluer le union-find en même temps que la forêt en créant un
    couple (uf,forest) *)
(* le folding set_union_edge permet d'accumuler les résultats à partir d'une
     forêt vide et d'un Union-Find totalement disjoint. *)
(* on pourrait maintenant utiliser un tas pour éviter le sort ...*)
pure_kruskal lcg;;
pure_kruskal uclg;;

let rec_pure_kruskal lg =
  let all_triplets = sort_triplets lg and n = List.length lg in
  let rec rec_set_union_edge (uf, forest) triplets =
    match triplets with
      | [] -> (uf, forest)
      | (i,j,w)::t ->   let ui = List.nth uf i and uj = List.nth uf j in
                if ui = uj
                then rec_set_union_edge (uf, forest) t
                else rec_set_union_edge (union ui uj uf, ((i,j,w)::forest))
                    t in
  snd(rec_set_union_edge ((List.init n (fun i -> i)),[]) all_triplets)
;;
rec_pure_kruskal lcg;;
rec_pure_kruskal uclg;;
```

B4. Comment pourrait-on utiliser une file de priorités pour améliorer les performances de cet algorithme ?

B5. Démontrer la correction et la terminaison de l'algorithme de Kruskal.