

# Arbres couvrants

OPTION INFORMATIQUE - TP n° 3.6 - 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 fonctionnel ou impératif 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 fonctionnel, c'est-à-dire sans références et sans structures itératives.

## 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 croître 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:    $T \leftarrow \emptyset$  ▷ la sortie : l'ensemble des arêtes de l'arbre recouvrant  
3:    $S \leftarrow s$  un sommet quelconque de  $V$   
4:   tant que  $S \neq V$  répéter  
5:      $(u, v) \leftarrow \min(w(u, v), u \in S, v \in V \setminus S)$  ▷ Choix glouton!  
6:      $S \leftarrow S \cup \{v\}$   
7:      $T \leftarrow T \cup \{(u, v)\}$   
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 variant 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 tour de boucle,  $|S| = 1$  et à chaque tour le cardinal de  $S$  augmente de 1. Donc,  $v$  est une

fonction à valeurs entières **strictement** décroissante. Elle atteint donc 0 (Th. de la limite monotone) lorsque la condition de boucle est invalidée, c'est-à-dire lorsque  $|S| = |V|$ . 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 choisis  $n - 1$  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 l'invariant suivant : «à chaque tour de boucle, le graphe  $(S, T)$  est un arbre couvrant de poids minimal. »

- **Initialisation** : à l'entrée de la boucle, on a  $|S| = 1$  et  $T = \emptyset$ , donc  $(S, T)$  est un arbre couvrant de poids minimal.
- **Conservation** : plaçons nous maintenant à un tour de boucle quelconque et supposons que, à l'entrée de ce tour,  $(S, T)$  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  $(S \cup \{b\}, T \cup \{(a, b)\})$  est nécessairement un arbre couvrant de poids minimal (cf. propriété de la coupe). Donc, à chaque tour de boucle, l'invariant est vérifié.
- **Terminaison** : 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.

- A3. Peut-on évaluer la complexité de l'algorithme de Prim?

**Solution :** À ce stade, c'est prématuré. En effet, on n'a fait aucune hypothèse sur les structures de données utilisées. Il va donc falloir faire des choix et ces choix vont grandement conditionner le résultat.

## B Implémentation de l'algorithme de Prim

Pour ce premier essai, on utilise une représentation des graphes par liste d'adjacence. 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)]];
```

```

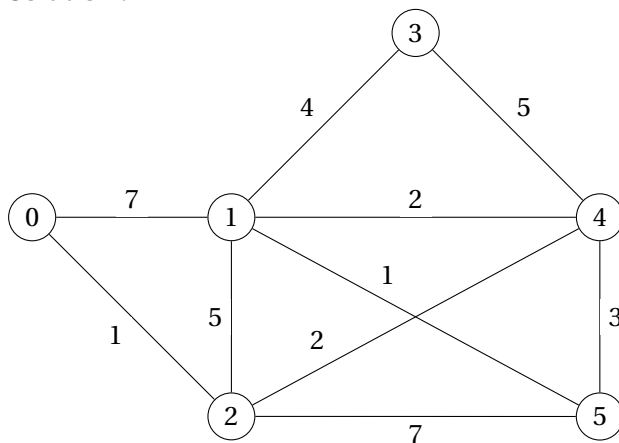
[(4,5); (1,4)];
[(5,3); (3,5); (2,2); (1,2)];
[(4,3); (2,7)];;

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

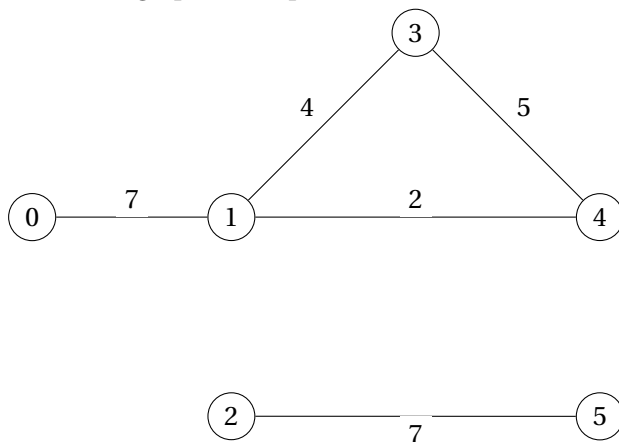
```

B1. Dessiner les deux graphes  $\text{l}_{cg}$  et  $\text{u}_{clg}$ . Quelle différence y-a-t-il entre les deux?

**Solution :**



Le second graphe n'est pas connexe :



Dans l'optique de programmer l'algorithme de Prim, on va créer quelques fonctions utiles. Tout d'abord, sans le but d'opérer un choix glouton, on souhaite trier les listes d'adjacence en fonction du poids de l'arc, du plus petit au plus grand. On choisit d'utiliser la fonction `List.sort` de la bibliothèque `List`. La documentation de la fonction est :

- `val sort : ('a -> 'a -> int) -> 'a list -> 'a list`
- Sort a list in increasing order according to a comparison function. The comparison function must return 0 if its arguments compare as equal, a positive integer if the first is greater, and a

negative integer if the first is smaller (see `Array.sort` for a complete specification). For example, `compare` is a suitable comparison function. The resulting list is sorted in increasing order. `List.sort` is guaranteed to run in constant heap space (in addition to the size of the result list) and logarithmic stack space.

- The current implementation uses Merge Sort. It runs in constant heap space and logarithmic stack space.

B2. Écrire une fonction de signature `compare_edges : 'a * int -> 'b * int -> int` qui compare deux arêtes  $(j,w)$  d'une liste d'adjacence et se comporte comme stipulé dans la documentation ci-dessus. L'objectif est de pouvoir utiliser ce comparateur avec `List.sort`.

**Solution :**

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

B3. Écrire une fonction de signature `sort_edges : ('a * int)list -> ('a * int)list` qui utilise `List.sort` et la fonction précédente. Cette fonction permet de trier dans l'ordre croissant de poids les éléments d'une liste d'adjacence. Par exemple  $[(3,4); (4,2); (2,5); (0,7)]$  devient  $[(4, 2); (3, 4); (2, 5); (0, 7)]$ .

**Solution :**

```
let sort_edges edges = List.sort compare_edges edges;;
```

B4. Écrire une fonction de signature `select_edge : (int * int)list -> bool array -> (int * int) option` qui sélectionne l'élément  $(j,w)$  de la liste d'adjacence de poids  $w$  minimum et dont le sommet  $j$  n'a pas été visité. On peut savoir si les sommets sont visités grâce au tableau `visited` de booléen passé en paramètre. Cette fonction renvoie un type optionnel car il est possible de ne pas trouver d'éléments. Elle s'appuie sur la fonction `sort_edges` pour trier la liste puis sur `List.find_opt` dont la documentation est donnée ci-dessous.

- `val find_opt : ('a -> bool) -> 'a list -> 'a option`
- `find f l` returns the first element of the list `l` that satisfies the predicate `f`. Returns `None` if there is no value that satisfies `f` in the list `l`.

**Solution :**

```
let select_edge edges visited = let sorted = sort_edges edges in
  List.find_opt (fun (j,_) -> not visited.(j)) sorted;;
```

B5. À l'aide du squelette de code ci-dessous, programmer l'algorithme de Prim dans une approche fonctionnelle récursive. Bien gérer tous les cas. Tester les deux graphes proposés ci-dessus.

```
let rec_prim g =
  let n = List.length g in
  let visited = Array.make n false in
  visited.(0) <- true;
```

```

let rec select_edges current tree =
  if ... (* STOP CONDITION *)
  then (* RETURN TREE *)
  else (* NEXT EDGE *)
in select_edges 0 [];

```

B6. Évaluer la complexité de l'algorithme de Prim ainsi implémenté. Pourrait-on faire mieux?

**Solution :** Soit  $n$  l'ordre du graphe. On effectue  $n - 1$  fois la fonction `select_edges` pour obtenir un arbre, c'est-à-dire autant de fois que de nœud. À chaque appel de la fonction, on exécute une fois `select_edge`.

Si  $a_k$  est le nombre de voisins d'un sommet  $k$  du graphe, la complexité de `select_edge` appliquée aux voisins de ce sommet  $k$  est celle de la fonction `sorted`,  $O(a_k \log a_k)$ , plus celle de la fonction `find_opt`,  $O(a_k)$ .

Dans le pire des cas, le graphe est complet et chaque sommet est relié à tous les autres par  $n - 1$  arêtes. La complexité totale de l'algorithme est donc :

$$C(n) = \sum_{k=0}^{n-2} (a_k \log a_k + a_k) \quad (1)$$

$$= \sum_{k=0}^{n-2} ((n-1) \log(n-1) + (n-1)) \quad (2)$$

$$= O(n^2 \log n + n^2) \quad (3)$$

$$= O(n^2 \log n) \quad (4)$$

$$(5)$$

Si on n'avait pas cherché intelligemment en triant les listes d'adjacence, on aurait trouvé une complexité en  $O(nm)$ , c'est-à-dire  $O(n^3)$  dans le pire des cas.

L'utilisation d'un tas est une alternative qui amène à une complexité similaire : si la structure d'adjacence est constituée de tableaux (Array) alors on peut faire de chaque tableau un tas-min.

- la création des tas coûte  $m$  (le nombre total d'arêtes du graphe)
- dans le pire des cas, on parcourt toutes les arêtes  $m$  et à chaque tour, on fait sortir du tas l'arête semi-visitée qui a le poids le plus faible et en même temps on remonte la suivante. Il n'y a pas plus d'arêtes que de nœuds dans ces tas, au pire  $n - 1$ . Ceci aurait un coût de  $m \log n$ .

Au final, utiliser un tas-min conduirait à une complexité en  $O(m + m \log n) = O(m \log n)$ . Dans le pire des cas, c'est-à-dire si le graphe est complet, on obtient donc le même résultat.

## C 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 la forêt et effectuer un choix d'arête, il ordonne les arêtes du graphe par ordre de pondération croissante.

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

**Algorithme 2** Algorithme de Kruskal, forêt d'arbres recouvrants

---

```

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

```

---

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 (UF) : c'est une structure très efficace<sup>1</sup> qui permet de réunir des ensembles disjoints en les étiquetant sous la même étiquette. On distingue ainsi les sommets du graphe qui sont connexes dans l'arbre en cours de construction et des autres.

La structure UF est essentiellement composée d'un tableau dont les éléments sont des tuples (étiquette de l'ensemble connexe, rang). Deux fonctions définissent la mécanique de la structure :

- `find_root` permet de trouver l'étiquette de la racine de l'ensemble auquel appartient un élément.
- `union` qui permet de réunir sous une même étiquette deux ensembles disjoints.

Selon la manière dont cette structure est implémentée (UF), le coût d'une union d'ensembles disjoints varie de  $O(n)$  à  $O(\alpha(n))$  où  $\alpha$  est l'inverse de la fonction d'Ackermann, c'est-à-dire une fonction qui croît infiniment lentement. Avec la meilleure implémentation, l'union est donc de complexité amortie quasi-constante (1).

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

Lorsqu'on connecte deux sommets par une arête, les sommets doivent appartenir au même ensemble connexe. On doit donc modifier `uf` et faire en sorte que les racines des deux sommets considérés portent la même étiquette. Par exemple, si on connecte le sommet 1 et le sommet 5, alors on aura `uf = [ (0,0), (1,1), (2,0), (3,0), (4,0), (1,0) ]` ou bien `uf = [ (0,0), (5,0), (2,0), (3,0), (4,0), (5,1) ]`.

Au fur et à mesure de l'algorithme, on va faire donc évoluer à la fois la liste des arêtes de l'arbre en construction 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 étiquettes des racines de ces sommets dans `uf` sont différentes. Par exemple, si on a `uf = [ (0,0), (5,0), (2,0), (3,0), (4,0), (5,1) ]`, on ne pourra pas ajouter l'arête (1,5) car les deux sommets sont déjà dans le même ensemble connexe.

À la fin de l'algorithme, la structure UF sera telle que :

- tous les sommets sont dans le même ensemble connexe si le graphe est connexe, par exemple `[ (0, 2); (0, 1); (0, 0); (0, 0); (1, 0); (0, 0) ]`.

---

1. On l'a déjà utilisée sans le savoir dans le TP labyrinthe du tronc commun!

- les sommets sont dans des ensembles connexes différents si le graphe n'est pas connexe, par exemple `[|(1, 0); (1, 1); (2, 1); (1, 0); (1, 0); (2, 0)|]`.

Pour utiliser la structure UF, on s'appuie sur le code suivant. Le code ci-dessous est plutôt orienté approche impérative, car la fonction `union` modifie directement le tableau, sans le renvoyer. Selon qu'on utilise une approche impérative ou récursive, ce code pourra être modifié à la marge.

```
(* fst -> étiquette du sommet *)
let rec find_root e uf =
  if fst uf.(e) <> e
  then find_parent (fst uf.(e)) uf
  else e;;

(* fst -> étiquette du sommet, snd -> rang du sommet *)
let union r1 r2 uf =
  if snd uf.(r1) < snd uf.(r2)
  then uf.(r1) <- (r2, snd uf.(r1))
  else
    begin
      uf.(r2) <- (r1, snd uf.(r2));
      if snd uf.(r1) = snd uf.(r2)
      then uf.(r1) <- (fst uf.(r1), (snd uf.(r1)) + 1)
    end;;

(* créer une structure uf*)
uf = Array.init n (fun i -> (i, 0));;
(* union des deux ensembles des sommets i et j si pas déjà fait *)
let r1 = find_root i uf and r2 = find_root j uf in
  if r1 <> r2
  then union r1 r2 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.

On choisit renvoyer également la structure UF lié à l'algorithme afin de rapidement visualiser le résultat.

Pour la valeur de retour de l'algorithme, on choisit donc un tuple de type `(uf, forest)` où `uf` est de type `(int * int)array` 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.**

- C1. Écrire une fonction de signature `make_triplets : (int * 'a)list list -> (int * int * 'a)array` qui transforme les listes d'adjacence d'un graphe en un tableau de triplets représentant les arêtes de ce graphe. Concrètement, pour un sommet  $i$  et chaque arête  $(j, w)$ , on ajoute le triplet  $(i, j, w)$  au tableau des triplets.

#### Solution :

```
let make_triplets 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
```

```
Array.of_list (List.fold_left filter [] (List.mapi (fun i edges -> List.
  map (fun (j,w) -> (i,j,w) ) edges) lg));;
```

- C2. Écrire une fonction de signature `compare_edges : 'a * 'b * int -> 'c * 'd * int -> int` afin de l'utiliser pour trier le tableau de triplets en fonction du poids de l'arête avec `Array.sort`. On s'inspirera de ce qu'on a fait pour l'algorithme de Prim.

**Solution :**

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

- C3. Écrire une fonction de signature `sort_edges : ('a * 'b * int)array -> unit` qui trie le tableau de triplets selon le poids du triplet dans l'ordre croissant.

**Solution :**

```
let sort_edges edges = Array.sort compare_edges edges;;
```

- C4. Écrire une fonction de signature `imp_kruskal : (int * int)list list -> (int * int * int) list * (int * int)array` qui implémente l'algorithme de Kruskal. Les paramètres est le graphe donné sous la forme d'une liste d'adjacence. La fonction renvoie le tuple (uf, forest).

**Solution :**

```
let imp_kruskal g =
  let n = List.length g in
  let edges = make_triplets g in
  Array.sort compare_edges edges;
  let forest = ref [] in
  let uf = Array.init n (fun i -> (i, 0)) in
  for k = 0 to (Array.length edges) - 1 do
    let (i,j,w) = edges.(k) in
    let r1 = find_root i uf and r2 = find_root j uf in
    if r1 <> r2 then (union r1 r2 uf; forest := (i,j,w)::!forest;)
  done;
  (!forest, uf);;

imp_kruskal lcg;;
imp_kruskal uclg;;
```

- C5. Évaluer la complexité de l'algorithme de Kruskal ainsi implémenté.



**Solution :** Comme on parcourt toutes les arêtes et que la structure UF fait l'union en  $O(\alpha(n))$ , la complexité est en  $O(m\alpha(n))$  : c'est excellent...

C6. Démontrer la correction et la terminaison de l'algorithme de Kruskal.

**Solution :**

- Terminaison : la boucle **for** n'est exécutée que  $|E| - 1$  fois.
- Correction : L'algorithme de Kruskal construit un graphe acyclique. Or, on sait que pour tout graphe acyclique  $G = (V, E)$ , on a :  $|E| \leq |V| - 1$ . Donc le nombre d'arêtes du graphe généré est au maximum  $n - 1$  (cas d'un arbre). Comme le graphe n'est pas nécessairement connexe, l'algorithme construit une forêt et on peut avoir moins de  $n - 1$  arêtes.

Il suffit donc maintenant de montrer que cette forêt est de poids minimum. On procède comme pour l'algorithme de Prim. L'invariant peut être : «à chaque tour de boucle, le graphe  $(S, T)$  est une forêt couvrante de poids minimal. »