Arbres couvrants

OPTION INFORMATIQUE - TP n° 3.5 - 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 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 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.

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

Pour faciliter la programmation, on utilise une représentation des graphes par liste d'adjacence. Cependant, on observe que les algorithmes de Prim ou de Kruskal construisent des arbres à partir des arêtes et du poids associé. Disposer des triplets `(i,j,w)` représentant les deux sommets d'une arête `(i,j)` de poids `w` est donc pertinent.

On définit donc un type graphe comme suit :

```
1  type list_graph = ((int * int) list) list;;
2
```

```
  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)]];;
```

Puis on utilisera dans le code une fonction pour transformer un graphe liste d'adjacence sous la forme d'une liste de triplets grâce à la fonction suivante :

```
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);;

make_triplets_list lcg;;
make_triplets_list uclg;;
```

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

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.

On dispose des triplets associés à un graphe. Dans l'optique de programmer l'algorithme de Prim et de faire un choix glouton, on souhaite trier cette liste de triplets 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.

Il faut donc lui expliquer comment comparer deux triplets d'après leur poids.

A5. Écrire une fonction de signature `compare_edges : 'a * 'b * int -> 'c * 'd * int -> int` qui compare les poids de deux triplets `(i,j,w1)` et `(k,l,w2)`. La fonction renvoie un type `int` positif si `w2` est plus grand que `w1`, 0 s'ils sont égaux et négatif sinon.

A6. 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. On utilisera la fonction `List.sort` ainsi que les fonctions précédentes.

A7. Écrire une fonction 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 constituent l'arbre de poids minimal. Lorsque le graphe n'est pas connecté, elle échoue avec le message `"Not connected graph !"`. On pourra utiliser l'instruction `List.filter ((<>)t)triplets` pour supprimer un triplet d'une liste de triplets. On s'appuiera également sur le squelette de code suivant :

```
let prim g =
  let n = List.length g in
  let visited = Array.make n false in
  visited.(0) <- true;
  let rec select_triplet triplets =
        match triplets with
          |  ..
  in let rec set_edge tree triplets =
        if ... (* condition d'arrêt *)
        then ... (* on a trouvé l'arbre ! *)
        else let triplet = select_triplet triplets in
                match triplet with
                  | ...
                  |
  in let all_sorted_triplets = ...
  in set_edge [] all_sorted_triplets;;
```

A8. Évaluer la complexité de l'algorithme ainsi implémenté. Pourrait-on faire mieux?

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

Algorithme 2 Algorithme de Kruskal, forêt d'arbres recouvrants

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

B3. Écrire une fonction 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`. On cherchera à utiliser la fonction précédente.

B4. Démontrer la correction et la terminaison de l'algorithme de Kruskal.

B5. Évaluer la complexité de l'algorithme ainsi implémenté.