I Algorithme de Prim

I.1 File de priorité implémenté par un tas (rappels de MP2I)

Une file de priorité (min) est une structure de données possédant les opérations suivantes :

- Extraire minimum : supprime et renvoie le minimum
- Ajouter élément
- Tester si la file de priorité est vide

On utilise souvent un tas pour implémenter une file de priorité. Un tas est un arbre binaire presque complet (tous les niveaux, sauf le dernier, sont complets) où chaque noeud est plus petit que ses fils. On stocke les noeuds du tas dans un tableau `t` tel que `t.(0)` est la racine, `t.(i)` a pour fils `t.(2*i + 1)` et `t.(2*i + 2)` si ceux-ci sont définis, et le père de `t.(j)` est `t.((j - 1)/2)` si $j \neq 0$.
On utilisera le type suivant pour représenter un tas : `type 'a tas = {t : 'a array; mutable n : int}`, où `n` est le nombre d'éléments du tas.

1. Écrire une fonction `swap : 'a tas -> int -> int -> unit` qui échange les éléments `tas.t.(i)` et `tas.t.(j)`.

> Solution :
>
> ```
> let swap tas i j =
>     let tmp = tas.t.(i) in
>     tas.t.(i) <- tas.t.(j);
>     tas.t.(j) <- tmp
> ```

On utilise deux fonctions auxiliaires pour implémenter les opérations sur un tas :

2. Écrire une fonction `up tas i` qui suppose que `tas` est un tas sauf `tas.t.(i)` qui peut être inférieur à son père et fait monter `tas.t.(i)` (en l'échangeant avec son père) de façon à obtenir un tas.

> Solution :
>
> ```
> let rec up tas i =
>     let p = (i - 1)/2 in
>     if i <> 0 && tas.t.(p) > tas.t.(i) then (
>         swap tas i p;
>         up tas p
>     )
> ```

3. En déduire une fonction `add tas x` qui ajoute l'élément `x` à `tas` (en tant que feuille la plus à droite) et la fait remonter pour conserver la propriété de tas.

> Solution :
>
> ```
> let add tas e =
>   tas.t.(tas.n) <- e;
>   up tas tas.n;
>   tas.n <- tas.n + 1
> ```

4. Écrire une fonction `down tas i` qui suppose que `tas` est un tas sauf `tas.t.(i)` qui peut être supérieur à un fils et fait descendre `tas.t.(i)` de façon à obtenir un tas.

> Solution :

```
let rec down tas i =
  let m = ref i in (* minimum parmi i et ses fils *)
  if 2*i + 1 < tas.n && tas.t.(2*i + 1) < tas.t.(!m)
  then m := 2*i + 1;
  if 2*i + 2 < tas.n && tas.t.(2*i + 2) < tas.t.(!m)
  then m := 2*i + 2;
  if !m <> i then (
    swap tas i !m;
    down tas !m
  )
```

5. En déduire une fonction `extract` qui extrait et renvoie le minimum d'un tas en remplaçant la racine par la dernière feuille et en la faisant descendre.

Solution :

```
let rec extract_min tas =
  swap tas 0 (tas.n - 1);
  tas.n <- tas.n - 1;
  down tas 0;
  tas.t.(tas.n)
```

6. Montrer que les fonctions `add` et `extract` sont en $O(\log(n))$ où n est le nombre d'éléments du tas.

I.2 Algorithme de Prim

Soit $G = (S, A)$ un graphe pondéré.

7. Soit $X \subsetneq S$ et $e \in A$ une arête de poids minimum ayant exactement une extrémité dans X. Montrer qu'il existe un arbre couvrant de poids minimum contenant e.

Solution : Soit $T' = T + e - e'$. Alors :

- T' est connexe. En effet, si $x, y \in S$ alors il existe un chemin P de u à v dans T. Si P passe par e', on remplace e' par le reste du chemin dans C. Ainsi, il existe un chemin de x à y dans T'.
- T' est un arbre couvrant car il contient $n-1$ arêtes et est connexe.
- $w(T') = w(T) + w(e) - w(e') \leqslant w(T)$ car $w(e) \leqslant w(e')$.

e appartient donc bien à T' qui est un arbre couvrant de poids minimum.

8. En déduire un algorithme en pseudo-code pour trouver un arbre couvrant de poids minimum de G.

Solution :

Algorithme de Prim

$r \longleftarrow$ sommet arbitraire de G
$S \longleftarrow \{r\}$
$T \longleftarrow \emptyset$
Pour $i = 0$ à $n-2$ **:**
  $e = \{u, v\} \longleftarrow$ de poids minimum telle que $u \in S$ et $v \notin S$
  $T \longleftarrow T \cup \{e\}$
  $S \longleftarrow S \cup \{v\}$
Renvoyer T

9. Écrire une fonction `prim : (int * float) list array -> int array` qui implémente cet algorithme, où :

- G est représenté par une liste d'adjacence pondérée `g` : `g.(i)` contient une liste de couples `(j, w)` où j est un sommet adjacent à i et w est le poids de l'arête entre i et j.
- `prim g` renvoie un tableau `p` tel que `p.(i)` contient le prédécesseur de i dans un arbre couvrant de poids minimum.

Solution :

```
let prim g =
  let n = Array.length g in
  let p = Array.make n (-1) in
  let tas = {t = Array.make n (max_float, -1, -1); n = 0} in
  add tas (0., 0, 0);
  let k = ref 0 in
  while !k < n do
    let w, u, v = extract_min tas in
    if p.(v) = -1 then (
      incr k;
      p.(v) <- u;
      List.iter (fun (u, w) -> add tas (w, v, u)) g.(v)
    )
  done;
  p
```

10. Donner la complexité de votre algorithme. Comparer avec l'algorithme de Kruskal.
11. Comment modifier l'algorithme de Prim pour obtenir l'algorithme de Dijkstra permettant de calculer les distances dans un graphe pondéré par des poids positifs ?

Solution : Ces deux algorithmes ont des objectifs différents : l'algorithme de Dijkstra permet de trouver les plus courts chemins entre un sommet s et tous les autres sommet. Cependant, l'algorithme de Prim peut être obtenu à partir de l'algorithme de Dijkstra en modifiant seulement les priorités utilisées dans la file de priorité.

II Autour de la complexité de l'algorithme de Kruskal

Soit $G = (S, A)$ un graphe pondéré, $n = |S|$ et $p = |A|$.

1. Écrire une fonction `void tri(int* t, int n)` qui trie un tableau `t` contenant n entiers entre 0 et K, en $O(n + K)$.

Solution :

```
void tri(int* t, int n) {
    int* c = malloc(p * sizeof(int));
    for (int i = 0; i < n; i++) {
        c[t[i]]++;
    }
    int j = 0;
    for (int i = 0; i < p; i++) {
        for (int k = 0; k < c[i]; k++) {
            t[j++] = i;
        }
    }
    free(c);
}
```

2. On suppose que les poids des arêtes de G sont compris entre 0 et p et que les opération de Union-Find en $O(1)$. Expliquer comment implémenter l'algorithme de Kruskal en $O(n + p)$.

Solution : On utilise le tri de la question précédente.

3. Montrer que la fonction suivante (réutilisant le code de l'exercice précédent) transforme un tableau en un tas en $O(n)$:

```
let tab_en_tas t =
    let n = Array.length t in
    let tas = { t=t; n=n } in
    for i = n/2 - 1 downto 0 do
        down tas i;
    done;
    tas
```

4. Expliquer comment modifier l'algorithme de Kruskal en utilisant la question précédente pour améliorer sa complexité dans certains cas.

III Union par taille

Dans le cours, on a étudié l'optimisation de la structure d'Union-Find en utilisant l'union par rang, en attachant l'arbre de hauteur la plus petite à celui de hauteur la plus grande. On peut aussi utiliser l'union par taille, en attachant l'arbre de plus petit nombre de sommets à celui de plus grand nombre de sommets.
On choisit alors d'utiliser un tableau `uf` tel que `uf.(i)` contient :

- $-k$ si i est une racine, où k est le nombre de sommets dans l'arbre de racine i.
- le prédécesseur de i dans l'arbre sinon.

1. Écrire les fonctions `create`, `find` et `union` en utilisant l'union par taille (et pas d'autre optimisation).

> Solution :
>
> ```
> let create n =
>     let uf = Array.make n (-1) in
>     for i = 0 to n - 1 do
>         uf.(i) <- -1
>     done;
>     uf
>
> let rec find uf i =
>     if uf.(i) < 0 then i
>     else find uf uf.(i)
>
> let union uf i j =
>     let i = find uf i in
>     let j = find uf j in
>     if i <> j then (
>         if uf.(i) < uf.(j) then (
>             uf.(i) <- uf.(i) + uf.(j);
>             uf.(j) <- i
>         ) else (
>             uf.(j) <- uf.(i) + uf.(j);
>             uf.(i) <- j
>         )
>     )
> ```

2. Montrer que l'opération `find` et `union` sont en $O(\log(n))$ pour une partition d'un ensemble de n éléments.

> Solution : On montre par récurrence sur le nombre de sommets $|T|$ d'un arbre T de hauteur h que $|T| \geq 2^h$, de façon similaire à la preuve du cours que l'optimisation d'union par rang donne des arbres de hauteur $\mathrm{O}(\log(n))$.

IV Questions sur les arbres couvrants de poids minimum

Soit $G = (S, A)$ un graphe pondéré.

1. Un dominant de G est un ensemble $X \subseteq S$ tel que $\forall v \in S$, $v \in X$ ou v est adjacent à un sommet de X.
On note $d(G)$ la taille minimum d'un dominant de G.

Montrer que si G est connexe alors $d(G) \leq \frac{|S|}{2}$.

2. Soit C un cycle de G et $e = \{u, v\}$ une arête de C dont le poids est strictement supérieur au poids des autres arêtes de C. Montrer que e ne peut pas appartenir à un arbre couvrant de poids minimum de G.

> Solution : Soit T un arbre couvrant de de poids minimum. Supposons par l'absurde que $e \in T$. Alors $T - e$ contient deux composants connexes G_u et G_v. Comme $C - e$ est un chemin qui relie u et v, il existe une arête e' de $C - e$ entre un sommet de G_u et un sommet de G_v. $T - e + e'$ est un arbre couvrant car est connexe (une seule composante connexe) et contient $n - 1$ arêtes. Mais $w(T - e + e') = w(T) - w(e) + w(e') < w(T)$, ce qui est absurde.

3. (Propriété d'échange) Soient T_1, T_2 deux arbres couvrants de G et e_1 une arête de $T_1 - T_2$. Montrer qu'il existe une arête e_2 de T_2 telle que $T_1 - e_1 + e_2$ (le graphe obtenu en remplaçant e_1 par e_2 dans T_1) est un arbre couvrant de G.

> Solution : Démonstration très similaire à la question précédente.

4. Soit T_1 un arbre couvrant de poids minimum de G et T_2 le 2ème plus petit arbre couvrant, c'est-à-dire l'arbre couvrant de poids minimum en excluant T_1. Montrer que T_1 et T_2 diffèrent d'une arête et en déduire un algorithme pour trouver T_2.