Manipulation de partitions et arbres couvrants

les 04 et 18 novembre

L'objectif de ce TP est d'implémenter l'algorithme de Kruskal de recherche d'un arbre couvrant de poids minimal. Pour cela, il faut savoir gérer des partitions. Nous commencerons donc par implémenter la structure Union-Find, puis nous l'utiliserons pour implémenter l'algorithme de Kruskal avant de nous intéresser à deux autres algorithmes de recherche d'arbre couvrant de poids minimal.

1 Partitions et Union-Find

Une solution pour gérer les composantes connexes d'un graphe est d'utiliser une structure union find.

Considérons la structure en C suivante :

```
struct partition {
   int nb_elements;
   int nb_sets;
   int* arr;
   int* rank;
};

typedef struct partition uf;
```

où le champs `nb_sets` désigne le nombre de classes de la partition.

▷ **Question 1.** Ecrire une fonction `uf* create(int n)` qui initialise une partition pour un ensemble de taille n, $\{0, \dots, n-1\}$ où chaque élément forme sa propre classe. ◁

▷ **Question 2.** Ecrire une fonction `int find(uf* u,int e)` qui renvoie un représentant canonique d'un élément `e` dans la partition `u`, cette fonction utilisera la compression des chemins. ◁

▷ **Question 3.** Ecrire une fonction `void fusion (uf* u, int e1, int e2)` qui modifie la partition afin de réunir les classes des éléments `e1` et `e2`. Cette fonction optimisera les rangs. ◁

▷ **Question 4.** On considère un graphe non orienté donné par ses listes d'adjacence, autrement dit représenté par le type :

```
struct graph {
   int n;
   int* degrees;
   int** adj;
};
typedef struct graph graph_t;
```

Utiliser la structure d'union-find pour rédiger une fonction qui calcule les composantes connexes de ce graphe.

`uf* composantes(graph_t* g)` ◁

▷ **Question 5.** Ecrire des fonctions de libération pour les structures union find et de graphes. ◁

Vous trouverez un fichier contenant la déclaration d'un petit graphe simple et une fonction d'affichage d'une structure union find qui vous permettra de tester vos fonctions.

Dans les parties suivantes, nous allons travailler avec des graphes pondérés. On commence donc par définir un type arête pondérée :

```
typedef double poids;
typedef int sommet;
struct edge {
   sommet x;
   sommet y;
   poids rho;
};
typedef struct edge edges;
```

2 Algorithme de Kruskal

On considère un graphe non orienté G connexe et pondéré à poids positifs. Un arbre couvrant du graphe $G = (V, A)$ est un arbre $Ar = (V, A')$ connexe tel que $A' \subset A$. L'objectif de ce TP est de trouver un arbre couvrant minimal (soit, de poids minimum) de G.

Dans l'algorithme de Kruskal, on utilise une représentation des graphes différente de celles que l'on a manipulées jusqu'à maintenant : liste des arêtes. Ainsi le type graphe pondéré ici est défini par :

```
struct graphe {
   int nb_sommets;
   int nb_aretes;
   edges* aretes;
};

typedef struct graphe graphe;
```

▷ **Question 6.** Utiliser la fonction `qsort` pour trier votre tableau d'arêtes dont la signature est expliquée ci-dessous :

```
void qsort(void *ptr, size_t count, size_t size,
int comp(const void*, const void*));
```

- `ptr` est un pointeur vers le début de la zone à trier (autrement dit, c'est un "tableau"), et son type est `void*`. Ce type est celui d'un pointeur générique : on ne sait pas quel est le type des objets que contient la zone à trier.
- `count` indique le nombre d'éléments qu'il faut trier. Son type est `size_t`, qui est simplement un type entier non signé suffisamment grand pour contenir la taille de n'importe quel tableau.
- `size` indique la taille (en octets) d'un objet à trier. La taille totale (en octets) de la zone à trier est donc `count * size`. Dans les fonctions que nous écrivons usuellement et qui prennent en entrée un tableau, ce paramètre est absent car inutile : comme le type du tableau est connu (disons `double*`), la taille d'un élément l'est également (`sizeof(double)`).
- Le dernier paramètre, `comp` est une fonction, qui sera utilisée pour comparer entre eux les éléments du tableau à trier. La fonction passée en paramètre doit avoir le prototype suivant :
  `int comp(const void* x, const void* y);`
  - Le mot clé `const` indique qu'on ne modifiera pas les valeurs pointées par `x` et `y`.
  - `x` et `y` sont donc fondamentalement des `void*`, c'est-à-dire qu'ils pointent chacun vers un objet de type quelconque.
  - La fonction doit renvoyer un entier strictement négatif si $x < y$, nul si $x = y$, strictement positif si $y < x$ (pour l'ordre utilisé pour le tri). C'est la même convention que pour la fonction de comparaison à passer à `List.sort` ou `Array.sort` en OCaml.

Pour trier un tableau d'entiers par ordre croissant , on pourrait donc procéder ainsi :

```
#include <stdlib.h>
int compare_ints(const void *px, const void *py){
   // On transtype px en int*, puis l'on déréférence le pointeur.
   int x = *(int*)px;
   int y = *(int*)py;
   return x - y;
}

int main(void){
   int t[4] = {12, 1, 7, -3};
   qsort(t, 4, sizeof(int), compare_ints);
}
.
```

◁

▷ **Question 7.** Implémenter l'algorithme de Kruskal à l'aide d'une structure union-find. Quelle est sa complexité ?

`graphe* kruskal(graphe* g).`

◁

On pourra de nouveau tester les fonctions proposées à l'aide de l'exemple proposé. On pensera aussi à ajouter une fonction de libération pour le nouveau type graphe.

3 Algorithme de Boruvka

Nous allons traiter cette partie en Ocaml, un fichier compagnon vous fournit une structure union find dans ce langage.

Voici le principe de l'algorithme de Boruvka :
Soit $G = (S, A)$ un graphe connexe, on initialise $F = (S, \emptyset)$.

Tant que F n'est pas connexe :
Trouver toutes les arêtes sures de F (une par CC) et les insérer à F.

▷ **Question 8.** Rappeler la définition d'une arête sure pour un ensemble de sommets $C \subset S$ d'un graphe $G = (S, A)$. ◁

On définit le type `graphe` (graphe pondéré) par :

```
type graphe = (int*int) list array
```

On considère le code suivant :

```
let mystere uf g=
  let n = Array.length g in
  let a_s = Array.make n (-1,max_int,-1) in
  for i = 0 to n-1 do
    let ri = find uf i in
    let rec aux liste = match liste with
      |[]->()
      |(a,p)::suite when (ri = find uf a ) -> aux suite;
      |(a,p)::suite  -> let x,y,z = a_s.(ri) in if (y > p) then a_s.(ri)<- (a,p,i); aux suite;
    in
    aux g.(i);
  done;
  a_s;;
```

▷ **Question 9.** Que fait cette fonction ? ◁

▷ **Question 10.** Ecrire une fonction récursive `boruv : graphe->graphe->uf->graphe` qui prend en entrée un graphe connexe `g`, une foret `f` qui est un sous graphe de l'arbre couvrant de poids minimal de `g` contenant tous les sommets de `g` ainsi qu'une partition de l'ensemble des sommets de `g` qui coincide avec les composantes connexes de `f` et qui renvoie l'arbre couvrant de poids minimal de `g`. ◁

▷ **Question 11.** Ecrire une fonction `boruvka : graphe -> graphe` qui renvoie l'arbre couvrant de poids minimal d'un graphe passé en entrée.
◁

4 UNE VARIANTE DUE À KRUSKAL.

Soit $G = (S, A, f)$ un graphe pondéré non orienté connexe et $T = (S, B)$ un arbre couvrant minimal de G. On appelle :

- arête dangereuse une arête qui est de poids maximal dans un cycle de G;
- arête utile une arête qui n'appartient à aucun cycle de G.

On suppose pour les questions théoriques seulement que f est injective.

▷ **Question 12.** Montrer que B ne contient aucune arête dangereuse. ◁

▷ **Question 13.** Montrer que B contient toutes les arêtes utiles. ◁

L'algorithme que nous allons ici considérer suit le principe suivant :
On parcourt les arêtes par ordre décroissant de poids. Si on tombe sur une arête dangereuse, on la supprime. Les questions précédentes garantissent la correction d'un tel algorithme.

▷ **Question 14.** Ecrire une fonction `chemin : graphe->int->int->bool` telle que l'appel à `chemin g x y` détermine s'il existe un chemin entre `x` et `y` dans `g`. On pourra adapter en Ocaml la fonction qui calcule les composantes connexes faite en première partie. ◁

Pour tester si une arête $a = \{s, t\}$ est dangereuse ou non, on crée une copie de G dans laquelle on supprime a et toutes les arêtes de poids strictement supérieur à $f(a)$. S'il existe encore un chemin de s à t, c'est que a est dangereuse.

▷ **Question 15.** Ecrire une fonction `supp_ar : graphe ->int->int->int->void` telle que `supp_ar g x y p` supprime l'arête (x, y, p) dans le graphe `g`. ◁

▷ **Question 16.** Ecrire une fonction `copie : graphe->graphe` qui copie un graphe. ◁

▷ **Question 17.** Ecrire une fonction `dangereuse : graphe->int->int->int->bool` telle que `dangereuse g x y p` teste si l'arête (x, y, p) est ou non dangereuse dans `g`. ◁

▷ **Question 18.** Ecrire une fonction `aretes : graphe-> (int*int*int) list` qui renvoie la liste des arêtes du graphe passé en argument. ◁

▷ **Question 19.** Implémenter l'algorithme précédent sous la forme d'une fonction `kruskal2 : graphe -> graphe` qui calcule et renvoie un arbre couvrant minimal. ◁

▷ **Question 20.** Déterminer sa complexité temporelle. ◁

5 Algorithme de Prim

Cette partie aura pour objectif d'implémenter l'algorithme de Prim et nécessitera de disposer d'une file de priorité pour laquelle on peut modifier la valeur de priorité d'un élément déjà présent dans la file (on ajoute donc une fonctionnalité plus avancée à `ajouter` et `supprimer_min`). Cette mise à jour de la valeur de priorité implique une réorganisation de la file de priorité ce qui n'est pas trop difficile à condition de savoir où se trouve le noeud dans le tas. Pour cela, on devra donc maintenir un tableau `tab_pos` qui retiendra dans quelle case du tableau représentant le tas (ie la file de priorité) se trouve chacun des sommets du graphe. Il va donc falloir adapter les fonctions de gestion de la file de priorité car chaque opération de file impliquera une modification éventuelle du tableau `tab_pos`.

On rappelle qu'une file de priorité est une structure de donnée qui permettra facilement de récupérer un élément de priorité (ici) minimale, de supprimer cet élément et d'insérer un nouvel élément.

On considère donc le type suivant :

```
type heap = {mutable taille : int; values : (int*int) array}
```

▷ **Question 21.** Ecrire une fonction `swap : (int*int) array->int->int->int array->unit` telle que `swap tableau pos1 pos2 tab_pos` échange le contenu des cases d'indices `pos1` et `pos2` de `tableau` et met à jour le tableau `tab_pos` pour que dans sa case d'indice `k` se trouve la position de l'élément de type `(k,_)` dans `tableau`. On suppose que le tableau `tab_pos` contenait déjà les positions des éléments de `tableau` et on met à jour les positions de ceux qui ont été échangés. On suppose aussi qu'il n'y a qu'un seul élément de la forme `(k,_)`. ◁

▷ **Question 22.** Ecrire une fonction `insere : int*int->heap->int array->unit` telle que `insere x tas tab_pos` insère le couple `x` dans `tas` en mettant à jour le tableau des postions pour qu'il reste cohérent avec les modifications effectuées dans le tas. ◁

▷ **Question 23.** Ecrire une fonction `delete_min : heap->int array->int` telle que `delete_min tas tab_pos` supprime l'élément de poids (deuxième valeur du couple) minimal, mette à jour le tableau `tab_pos` et renvoie la valeur (première valeur du couple) de poids minimal. ◁

▷ **Question 24.** Ecrire une fonction `modif_poids : heap->int->int->int array->unit` telle que `modif_poids tas s p tab_pos` modifie **à la baisse** le poids associé au sommet `s` (ce dernier devient `p`) et mette à jour le tableau `tab_pos` en fonction des modifications apportées à l'organisation du tas. ◁

On utilisera de nouveau le type suivant :

```
type graphe = (int*int) list array
```

▷ **Question 25.** Ecrire une fonction `prim : graphe -> int -> graphe` qui calcule un arbre couvrant de poids minimal pour son entrée. ◁

Un graphe sur lequel tester votre fonction :

```
let graphe1 = [|[((1,4));(7,8)];[(0,4);(2,8);(7,11)];[(1,8);(3,7);(5,4);(8,2)];
[(2,7);(4,9);(5,14)];[(3,9);(5,10)];[(4,10);(3,14);(2,4);(6,2)];
[(5,2);(8,6);(7,1)];[(6,1);(8,7);(1,11);(0,8)];[(2,2);(6,6);(7,7)]|];;
```

Deux arbres couvrants de poids minimaux pour ce graphe sont :

`[|[1]; [2; 0]; [3; 5; 8; 1]; [4; 2]; [3]; [6; 2]; [7; 5]; [6]; [2]|]`

et `[|[7; 1]; [0]; [3; 8; 5]; [4; 2]; [3]; [2; 6]; [5; 7]; [6; 0]; [2]|]`