Partiel du 12 mars 2018

Documents autorisés - Appareil électronique interdits

Pensez à utiliser les résultats des questions précédentes même si vous n'avez pas proposé d'implémentation. Attention, le barème est seulement indicatif.

Le but du partiel est d'implémenter un algorithme de segmentation d'images qui utilise une structure de données *union-find*. Dans ce devoir, nous allons nous intéresser aux partitions d'un ensemble que l'on nommera univers. Par exemple $P = \{\{1,3,5\},\{2,4\},\{6\}\}$ est la partition de l'univers $\{1,2,3,4,5,6\}$ en trois ensembles $\{1,3,5\}$, $\{2,4\}$ et $\{6\}$. On fera la distinction entre le terme univers utilisé pour parler de tous les éléments que l'on considère et le terme ensemble, utilisé pour parler des éléments de la partition P. Pour chaque ensemble de la partition, l'un des éléments, choisi arbitrairement, est désigné comme étant le représentant de l'ensemble. Dans notre exemple, un choix (parmi d'autres) de représentants est 3, 2 et 6. Ainsi, 3 représente $\{1,3,5\}$, 2 représente $\{2,4\}$ et 6 représente $\{6\}$.

1 Union Find avec des listes

1. (5 points) Notre objectif est de définir une structure de données dans laquelle nous chercherons à effectuer trois opérations :

   MakeSet(P, x) : Ajoute un ensemble ne contenant que x à la partition P. Notez qu'ici, x ne doit pas appartenir à l'univers partitionné par P. Si ce n'est pas le cas, la fonction renvoie une erreur.
   (ex. $MakeSet(P,7) = \{\{1,3,5\},\{2,4\},\{6\},\{7\}\}$).

   Find(P,x) : retourne le représentant de l'ensemble auquel appartient x et une erreur si x n'est pas dans l'univers.
   (ex. $Find(P,1) = 3$).

   Union(P,x,y) : produit une nouvelle partition, identique à P, à l'exception que l'ensemble dans P contenant x et l'ensemble dans P contenant y ne forment qu'un seul ensemble dans la nouvelle partition. Notez que si x et y appartiennent au même ensemble, $Union(P,x,y) = P$.
   (ex. $Union(P,3,2) = \{\{1,2,3,4,5\},\{6\}\}$).

   De plus on notera $Empty = \{\}$ comme étant l'unique partition de l'univers ne contenant aucun élément $U = \{\}$. Une telle structure de données est appelée *Union-find*.

   Dans cette partie, vous implémenterez en OCaml les partitions par un type *'a t = 'a list list*. (ex. P s'écrit alors $[[1;3;5];[2;4];[6]]$). Le représentant d'un ensemble sera fixé comme étant le premier élément de la liste correspondant à cet ensemble. (ex. $Find(P,5) = 1$).

   Écrire un module `UnionFindList` qui implémente union-find dont la signature est compatible avec la suivante :

```
module type UNIONFIND =
sig
  type 'a t
  val empty : 'a t
  val make_set : 'a -> 'a t -> 'a t
  val find : 'a t -> 'a -> 'a
  val union : 'a t -> 'a -> 'a -> 'a t
end
```

2. (4 points) Le module `UnionFindList` ainsi créé nous permet d'effectuer les trois opérations définissant un union-find. Par la suite nous supposerons que le type 'a sera donné par l'unique type du module `ELEMENT`.

```
module type ELEMENT =
  sig
      type t
  end
```

En pratique lorsque l'on va vouloir utiliser les fonctions définies dans le module `UnionFindList`, nous chercherons à garder en mémoire la partition sur laquelle nous travaillons. Pour cela nous utiliserons des modules compatibles avec la signature suivante :

```
module type PARTUF =
  sig
    type t
    type u
    val part : u ref
    val make_set : t -> unit
    val find : t -> t
    val union : t -> t -> unit
  end
```

Ici `t` correspond au type des éléments de l'univers et `u` correspond au type de la partition.
Compléter le foncteur `Make`, donné ci-dessous, prenant en paramètre un module de signature `UNIONFIND` et un module de signature `ELEMENT` et retournant un module de signature `PARTUF` respectant les conditions suivantes. Le module doit contenir une valeur `part` initialisée à une partition vide (empty) et chaque opération `make_set`, `find` et `union`, qui réutilise les opérations de même nom dans le module de signature `UNIONFIND`, doit mettre à jour cette partition.

```
module Make (Uf : UNIONFIND) (Ty: ELEMENT) =
struct
  type t = Ty.t
  type u = Ty.t Uf.t
  let part = ref ( Uf.empty : u)
  let make_set x =
  let find x =
  let union x y =
end
```

2 Segmentation d'images

Le but de cet exercice est de faire de la segmentation d'images en utilisant la structure de données *d'union-find* décrite précédemment. Segmenter l'image, c'est décomposer l'image en zones d'intérêts, ces zones devant être connexes.

3. (1 point) Sachant que l'on représente le type pixel par `type pixel = {red : int; green : int ; blue : int}`, écrire une fonction de comparaison `equal : comp_pixel`, où
   `type comp_pixel = pixel -> pixel -> bool`, qui teste si deux pixels ont la même valeur.
4. (2 points) Une image étant représentée par une matrice de pixels, nous utiliserons le module `Coord` :

   ```
   module Coord =
   struct
       type t = int * int
   end
   ```

   Écrire un module `PartUF`, de signature `PARTUF`, utilisant l'implémentation de l'union-find par des listes et tel que le type `PartUF.t` corresponde au type `Coord.t`.
5. (4 points) On considère que chaque pixel est supposé avoir 4 voisins (droite, gauche, haut, bas), à l'exception des pixels sur les bords de l'image. On veut ici former des régions où tous les pixels ont la même valeur.
   On parcourt l'image de haut en bas et de gauche à droite. L'algorithme fonctionne de la façon suivante :
   1. Ajouter les coordonnées du pixel actuel comme nouvel ensemble de la partition.
   2. S'il existe un pixel à gauche et que ce pixel est comparable au pixel actuel, fusionner les ensembles correspondants.
   3. S'il existe un pixel au dessus et que ce pixel est comparable au pixel actuel, fusionner les ensembles correspondants.

   L'image étant stockée avec le type `type image = pixel array array`, écrire une fonction `segmentation` qui prend en entrée une image de type `image` et une fonction de comparaison de type `comp_pixel` et retourne une partition de type `PartUF.u` tel que les zones de l'image soient partitionnées suivant l'algorithme présenté ci-dessus.
6. (2 points) Écrire une fonction `representants : image -> comp_pixel -> (PartUF.t * pixel) list` qui à partir d'une segmentation d'image renvoie la liste des couples (r, p) tels que r est un représentant d'un ensemble et p est le pixel associé à l'ensemble.
7. (2 points) Donner une fonction de similarité `sim : int -> pixel -> pixel -> bool` qui indique que deux pixels sont similaires si la différence de leurs valeurs pour chacun des canaux de couleur est inférieure à α, un entier donné en premier argument.
   On dit que deux pixels arbitraires, pas forcément adjacents, sont α-connexes si et seulement s'il existe un chemin entre les deux pixels tels que deux pixels de ce chemin qui sont adjacents ont une similarité inférieure à α. On remarquera qu'il suffit

de tester la similarité des pixels haut et gauche au pixel courant. Écrire la fonction `segmentation_alpha_connexite alpha image` de type `int -> image -> PartUF.u` qui retourne une segmentation de l'image source en composantes α-connexes.

8. (1 point) (*bonus*) Dans cette question, nous nous intéresserons à des fonctions du type `val_set`où `type val_set = PartUF.t list -> image -> pixel` qui calculent une valeur de pixel à partir de l'ensemble des éléments de la liste L donnée en premier paramètre. En vous inspirant de la question 6, écrire une fonction `regions : image -> comp_pixel -> val_set -> (PartUF.t * pixel)list`

9. (1 point) (*bonus*) Écrire une fonction `moyenne` de type `val_set` calculant la moyenne des valeurs de tous les pixels de l'ensemble considéré. En déduire une fonction `regions_moyenne` qui renvoie la liste des couples (r, p) tels que r est un représentant d'un ensemble et p est le pixel moyen de cet ensemble.

   Retrouver la fonction de la question 6 en utilisant la fonction de la question 8.