TD9 - Union-Find

Exercice 1

On donne ci-dessous le code d'une implémentation de la structure *union-find* en OCaml.

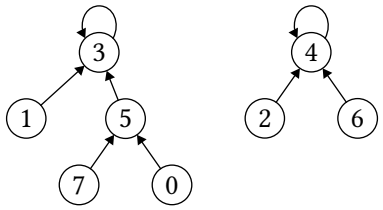
```
type uf = {
  link: int array;
  rank: int array;
}

let create n =
  { link = Array.init n (fun i -> i);
    rank = Array.make n 0; }

let rec find uf i =
  let p = uf.link.(i) in
  if p = i then i else (
    let r = find uf p in
    uf.link.(i) <- r;
    r
  )
```

```
let union uf i j =
  let ri = find uf i in
  let rj = find uf j in
  if ri <> rj then
    if uf.rank.(ri) < uf.rank.(rj)
    then uf.link.(ri) <- rj
    else (
      uf.link.(rj) <- ri;
      if uf.rank.(ri) = uf.rank.(rj)
      then uf.rank.(ri) <- uf.rank.(ri) + 1
    )
```

Question 1. Donner une séquence d'opérations `union` qui conduise à la situation représentée sur la figure suivante ?



Question 2. Comment modifier la structure *union-find* pour ajouter une opération `num_classes` donnant le nombre de classes distinctes. On s'efforcera de fournir cette valeur en temps constant, en maintenant la valeur comme un champ supplémentaire.

Exercice 2

Une autre réalisation de la structure *union-find* consiste à ne pas utiliser de tableaux, mais à représenter directement chaque élément comme une structure mutable contenant un pointeur vers un autre élément dès lors qu'il n'est pas le représentant de la classe. Cela peut être par exemple un type de la forme suivante.

```
type elt = node ref
and node = Root of int | Link of elt
```

Ici, un élément est une référence OCaml, contenant soit `Root` k pour désigner le représentant d'une classe de rang k, soit `Link` x pour pointer vers un autre élément x. L'interface est alors la suivante.

```
type elt
val singleton: unit -> elt
val find: elt -> elt
val union: elt -> elt -> unit
```

La comparaison des éléments se fait maintenant avec l'égalité physique ==. Donner le code de ces trois fonctions.

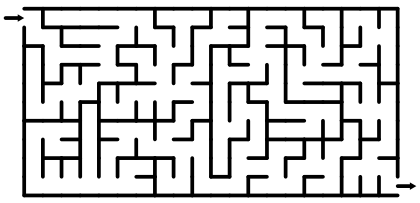
Remarque. Il y a deux fonctions d'égalité en Caml : = et == qui sont toutes deux polymorphes de type : 'a -> 'a -> bool. Elles ne doivent pas être confondues car elles sont sémantiquement différentes.

- == teste l'identité de ses arguments, en testant l'*égalité physique* des représentations mémoires. == est très efficace car ce prédicat correspond généralement à une instruction de la machine. Il s'applique à tout couple de valeurs du même type sans provoquer d'erreurs : pour toutes les valeurs non allouées on teste l'identité bit-à-bit de leur représentation, et pour toutes les valeurs allouées on teste si elles sont stockées à la même adresse mémoire.
- = teste l'*égalité sémantique* : les arguments sont-ils isomorphes ? Pour cela, = parcourt en parallèle les deux données jusqu'à trouver une différence, ou avoir terminé le parcours. C'est pourquoi = peut boucler en cas de valeurs cycliques.

◁

Exercice 3

On peut utiliser la structure *union-find* pour construire efficacement un labyrinthe parfait, c'est-à-dire un labyrinthe où il existe un chemin et un seul entre deux cases. Voici un exemple de tel labyrinthe.



On procède de la manière suivante. On crée une structure *union-find* dont les éléments sont les différentes cases. L'idée est que deux cases sont dans la même classe si et seulement si elles sont reliées par un chemin. Initialement, toutes les cases du labyrinthe sont séparées les unes des autres par des portes fermées. Puis on considère toutes les paires de cases adjacentes (verticalement et horizontalement) dans un ordre aléatoire. Pour chaque paire (c_1, c_2) on compare les classes des cases c_1 et c_2. Si elles sont identiques, on ne fait rien. Sinon, on ouvre la porte qui sépare c_1 et c_2 et on réunit les deux classes avec `union`.

Question 1. Écrire un code qui construit un labyrinthe selon cette méthode.
Indication : pour parcourir toutes les paires de cases adjacentes dans un ordre aléatoire, le plus simple est de construire un tableau contenant toutes ces paires, puis de le mélanger aléatoirement en utilisant le mélange de Knuth *(vu en première année).*

Question 2. Justifier qu'à l'issue de la construction, chaque case est reliée à toute autre case par un unique chemin.