TD9 - Union-Find (Éléments de réponses)

Exercice 1

Question 1. Il faut bien lire le code, notamment pour comprendre quel élément devient le représentant en cas d'égalité des rangs. La séquence suivante convient (mais elle n'est pas la seule) :

```
let uf = create 8
let () = union uf 3 1
let () = union uf 5 7
let () = union uf 5 0
let () = union uf 3 5
let () = union uf 4 2
let () = union uf 4 6
```

Question 2. On ajoute au type `uf` un champ mutable pour le nombre de classes, et la fonction `num_classes` renvoie sa valeur. Dans la fonction `create`, on l'initialise à `n`. Enfin, on le décrémente dans la fonction `union` dès lors que les deux représentants sont distincts.

Exercice 2

Le code s'inspire de celui donné pour la première implémentation donnée dans le premier exercice.

```
let singleton () = ref (Root 0)

let rec find x = match !x with
  | Root _ -> x
  | Link y -> let rx = find y in x := Link rx; rx

let union x y =
  let x = find x in
  let y = find y in
  if x != y then match !x, !y with
    | Root rx, Root ry ->
        if rx < ry then x := Link y
        else (y := Link x; if rx = ry then x := Root (rx + 1))
    | _ -> assert false
```

Deux éléments sont à remarquer ici : on a utilisé l'égalité physique pour comparer les deux représentants dans `union`; la dernière ligne exprime que le résultat de `find` ne peut être que de la forme `Root`, un invariant que le typage d'OCaml ne permet pas de capturer.

Exercice 3

Question 1. On peut supposer qu'on va construire un labyrinthe $n \times n$ avec la valeur de n reçue sur la ligne de commande.

```
let n = int_of_string Sys.argv.(1)
let uf = create (n * n)
```

Pour construire toutes les portes du labyrinthe, on peut utiliser un triplet (i, j, d) où i est la ligne, j la colonne et d la « direction » de la porte, donnée par exemple par le type suivant :

```
type direction = H | V
```

La valeur `H` désigne une porte entre (i, j) et $(i, j + 1)$ et la valeur `V` une porte entre (i, j) et $(i + 1, j)$. On commence par construire un tableau contenant toutes les portes possibles

```
let doors =
  let l = ref [] in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      if i < n-1 then l := (i, j, V) :: !l;
      if j < n-1 then l := (i, j, H) :: !l;
    done;
  done;
  Array.of_list !l
```

puis on le mélange, par exemple avec l'ex. ?? p. ?? (le code est omis). On se donne deux matrices de booléens indiquant si les portes sont fermées.

```
let horiz = Array.make_matrix n n true
let vert  = Array.make_matrix n n true
```

Enfin, on ouvre les porte en appliquant l'algorithme proposé.

```
let () =
  let f (i, j, hv) =
    let k = i * n + j in
    let k' = if hv = H then k + 1 else k + n in
    if find uf k <> find uf k' then (
      if hv = H
      then horiz.(i).(j) <- false
      else vert.(i).(j) <- false;
      union uf k k';
    )
  in
  Array.iter f doors
```

Il ne reste plus qu'à dessiner le labyrinthe, ce qui l'on peut faire en ASCII sur la sortie standard ou bien avec une bibliothèque graphique de son choix. On laisse cela au lecteur.

Question 2. Justifions qu'il s'agit là d'un labyrinthe parfait, par récurrence sur la dernière boucle du programme ci-dessus. L'hypothèse de récurrence est qu'à tout moment deux cases sont reliées par un chemin si et seulement si elles appartiennent à la même classe et que ce chemin est alors unique. Initialement, c'est trivialement vrai car chaque classe ne contient qu'une seule case. Supposons la propriété vraie et effectuons un tour de boucle. Si les deux cases `cell` et `next` choisies sont déjà dans la même classe, on ne fait rien et la propriété reste donc trivialement vérifiée. Si en revanche on réunit les deux classes, considérons deux cases a et b dans cette nouvelle classe. Si a et b sont toutes deux dans l'ancienne classe de `cell`, appelons-la C, alors elles sont reliées par un unique chemin dans C. Si un autre chemin existait, il devrait emprunter deux fois `w` et ce ne serait pas un chemin. De même si a et b sont toutes deux dans la classe de `next`. Si enfin a est dans la classe de `cell` et b dans la classe de `next` (ou le contraire), alors par hypothèse il existe un unique chemin de a à `cell` et un unique chemin de `next` à b, donc un unique chemin de a à b.