TP 8 : Structure Unir et Trouver

Exercice 1 : *Implémentation par tableau.* Dans cet exercice, on construit un type *Unir et Trouver* avec une implémentation à base de tableau. Pour chaque élément (les indices du tableau), le tableau associe son représentant.

1. Écrire un type `unirtrouver` pour cette implémentation.
2. Écrire une fonction `init : int → unirtrouver` qui prend en entrée le nombre d'éléments de la partition et renvoie une partition initiale où chaque élément est dans son propre sous-ensemble.
3. Écrire une fonction `trouver : unirtrouver → int → int` qui renvoie le représentant d'un élément de la partition.
4. Écrire une fonction `unir : unirtrouver → int → int → unit` qui unit les sous-ensembles de deux éléments.

Exercice 2 : *Coloration connexe.* Dans cet exercice, on souhaite colorier une image initialement en noir et blanc. Une image est représentée par un tableau en 2D dont les cases contiennent un entier représentant une couleur (0 pour le blanc, 1 pour le noir).

1. Définir un type image pour représenter une image 2D.
2. Écrire une fonction `nouvelle_image : int → image` tel que `nouvelle_image n` renvoie un tableau de taille $n \times n$ tel que chaque case contient la valeur 0 ou 1, aléatoirement. On rappelle l'existence du module Ocaml `Random` pour tirer des nombres au hasard.
3. Écrire une fonction `afficher_image_nb : image → unit` pour afficher l'image en utilisant le caractère `'.'` pour les cases blanches et `'#'` pour les cases noires. On obtiendrait par exemple l'affichage ci-dessous en Figure 1.

```
. . # # . # . . # .
# . # . # . # # # #
# # # # . . # # . .
. # . . # # . # . .
. . # # . # . # # #
# # # # # # . # . .
. # . # . . . # # .
. . # . # # # # . .
# # . . . # . # . #
. # . # # . . . . #
```

Figure 1. – Affichage « en noir et blanc »

```
a a b b c d e e f g
b a b h i j f f f f
b b b b j j f f k k
l b m m n n o f k k
l l n n p n o f f f
n n n n n n o f q q
r n s n o o o f f q
r r t u f f f f q q
v v u u u f q f q w
x v u y y q q q q w
```

Figure 2. – Affichage « en couleur »

On veut maintenant identifier les composantes connexes, c'est-à-dire les cases de même couleur qui sont connectées par un de leur côté commun.

4. Pour pouvoir utiliser la structure `unirtrouver` définie précédemment, nous avons besoin d'une fonction `identifiant : int → int → int → int` qui, pour une taille n de l'image, associe à chaque case (i, j) un identifiant unique (la fonction doit être injective). L'écrire et justifier qu'elle est injective.
5. Écrire une fonction `composantes : image → unirtrouver` qui renvoie une partition correspondant aux composantes connexes de l'image.
6. Écrire une fonction `recolore : image → unirtrouver → image` qui renvoie une image en couleur (la valeur des cases pouvant dépasser 1) correspondant à l'image d'entrée où chaque composante connexe identifiée dans la partition `unirtrouver` a été coloriée avec une couleur qui lui est propre.
7. Écrire une fonction `afficher_image : image → unit` qui affiche dans la console l'image en couleur, on pourra utiliser les symboles ASCII à partir de la lettre `'a'` pour chaque couleur. La coloration de l'image précédente sera alors celui de la Figure 2.
8. Tester l'ensemble avec :

```
let img = nouvelle _image 10 in afficher_image_nb img;
let taches = composantes img in let img_coloree = recolore img taches in afficher_image img_coloree
```

Exercice 3 : *Implémentation plus efficace.* On va maintenant améliorer l'implémentation de l'Exercice 1. Pour cela, on va appliquer les optimisations vues en cours. Pour chaque question ci-dessous, vous prendrez soin de vous assurer que le comportement de la coloration d'image fonctionne toujours correctement avant de passer à la fonction suivante.

1. La première optimisation est la représentation par arbre. Chaque case du tableau ne contient plus le représentant de l'élément, mais son parent dans l'arbre.
   - On trouve un représentant d'un élément en remontant de parent en parent, jusqu'à trouver la racine de l'arbre.
   - Un représentant est caractérisé par un élément qui est son propre parent.
   - L'union de deux éléments se fait en désignant le représentant de l'un comme parent du représentant de l'autre, de façon arbitraire.
2. La seconde optimisation est l'union par rang. Pour cela, on doit modifier la structure de donnée pour y ajouter un second tableau représentant les rangs des éléments. À l'initialisation, tous les rangs sont à 0, puis :

- Lors de l'union, on compare les rangs des représentants. Le représentant de plus petit rang devient le fils du représentant de l'autre.
- Dans le cas où les rangs sont égaux, on choisit continue à choisir arbitrairement, et on augmente le rang de la nouvelle racine de 1.

3. La dernière optimisation est la compression de chemin. Cette optimisation consiste à modifier l'opération **`trouver`**, de la façon suivante. Quand on cherche le représentant r d'un élément x, on doit remonter de parent en parent dans l'arbre. Tous les éléments rencontrés en chemin ont également r pour représentant. Ainsi, après avoir trouvé le représentant, on redescend dans l'arbre en modifiant le parent de chacun de ces éléments pour qu'il devienne directement r.

Avec toutes les optimisations, la complexité amortie des opérations **`unir`** *et* **`trouver`** *est en $\mathcal{O}(\alpha(n))$, où α est une fonction à la croissance extrêmement lente. Pour se donner un ordre d'idée de la lenteur extrême de la croissance de cette fonction, on a pour tout $n \leq 10^{80}$ (estimation du nombre d'atomes dans l'univers observable), on a $\alpha(n) \leq 4$. Cette complexité est optimale.*