TP-Colle-1 (MPI) : Jeu des 7 couleurs

Environnement

Nous travaillerons en OCaml mode interprété pour ce TP-colle. Nous vous laissons le choix de l'environnement, selon vos habitudes. Vous pouvez utiliser :

- L'interpréteur en ligne BetterOCaml : https://BetterOCaml.ml/
- Emacs, muni du mode Tuareg.

Consignes et mise en place

- Veuillez trouver dans le répertoire partagé le fichier `7colors.ml` (aussi disponible sur la page Cahier-de-Prépa : `https://cahier-de-prepa.fr/info-kleber/`). Si vous utilisez un interpréteur en ligne, importez-y ce fichier, ou copiez-son contenu.
- Vous répondrez aux questions de programmation en écrivant le code dans votre outil.
- Vous répondrez aux questions de cours à l'aide de la fiche-réponse à la fin du sujet.
- Vous répondrez aux questions à développer à l'oral lors de votre passage au tableau (pour les concernés). Utilisez un brouillon pour vous préparer des notes.

Dans les questions à développer ET dans votre implémentation, on considérera qu'une structure Unir-Trouver porte sur tous les entiers d'un intervalle $[\![0; N-1]\!]$ pour un certain N.

I La structure Unir-Trouver

1. *(Question de cours)* Présentez le principe de la structure Unir-Trouver dans son implémentation par arbres (sans optimisation). Veillez à préciser comment s'effectuent les deux opérations Unir et Trouver, et leur complexité.
2. *(Question de cours)* Quelles optimisations peut-on faire dans cette implémentation ? Quelle complexité totale obtient-on lorsqu'on effectue m opérations sur des partitions d'un ensemble à n éléments ?
3. Implémentez un type `unirtrouver` qui réalise une structure de données Unir-Trouver. (Vous pouvez choisir l'implémentation que vous venez de décrire, ou une plus naïve...)
4. Implémentez une fonction `init : int -> unirtrouver`, telle que `init n` crée une structure Unir-Trouver à `n` éléments dans son état initial.
5. Implémentez la fonction `trouver : unirtrouver -> int -> int` adaptée à votre structure de données.
6. Implémentez la fonction `unir : unirtrouver -> int -> int -> unit` adaptée à votre structure de données.

Le jeu des 7 couleurs

Nous allons implémenter des fonctions qui permettent de réaliser le jeu suivant : on considère une image carrée, dont les lignes et les colonnes sont numérotées de 0 à $n-1$.

- Règle 1 : Initialement, le joueur A (resp. B) possède la case $(0,0)$ (resp. $(n-1,n-1)$) . Toutes les autres cases reçoivent une couleur aléatoire, représentée par un entier entre 0 et 6 inclus.
- Règle 2 : Les joueurs jouent tour par tour en commençant par le joueur A.
- Règle 3 : A chaque tour, le joueur concerné choisit une couleur entre 0 et 6 : tant qu'il existe une case de cette couleur adjacente à son territoire, il l'intègre à son territoire. Son tour se termine s'il n'y a plus de case à annexer.

- Règle 4 : Le jeu s'arrête dès qu'un joueur possède plus de la moitié des cases. Celui-ci est déclaré vainqueur.

Voici un exemple de début de partie :

A 0 0 5 2	A A A 5 2	A A A 5 2	A A A A 2	A A A A 2
0 0 1 5 3	A A 1 5 3	A A 1 5 3	A A 1 A 3	A A B A 3
5 5 1 1 3	5 5 1 1 3	5 5 1 1 3	A A 1 1 3	A A B B 3
3 3 3 1 1	3 3 3 1 1	3 3 3 1 1	3 3 3 1 1	3 3 3 B B
4 4 4 4 B	4 4 4 4 B	B B B B B	B B B B B	B B B B B
Etat initial	A joue 0	B joue 4	A joue 5	B joue 1

II Création d'images

On fournit le type `image`, qui représente une instance du problème ci-dessus, muni de trois champs :

- `taille`, le nombre de lignes/colonnes de l'image
- `plateau`, la partition réalisée par Unir-Trouver qui permettra de *construire des classes de cases de la même couleur.*
- `couleurs`, une table de hachage qui, pour chaque classe, contient le lien "représentant → couleur".

Attention, dans les champs du type `image`, pour une image de côté `n` chaque case est repérée non pas par ses coordonnées dans $[\![0, n-1]\!] \times [\![0, n-1]\!]$, mais par un unique identifiant entre 0 et $n^2 - 1$.

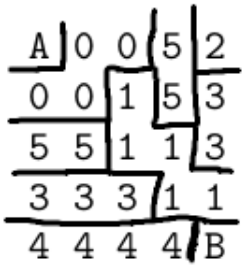
7. *(Question à développer à l'oral)* Proposez une fonction $f : [\![0, n-1]\!] \times [\![0, n-1]\!] \to [\![0, n^2-1]\!]$ injective, qu'on pourrait utiliser pour construire cet identifiant. Prouvez qu'elle est injective.
8. Ecrivez une fonction `identifiant : int -> int -> int -> int`, telle que `identifiant n i j` renvoie f(`i`, `j`) de la case de coordonnées `(i,j)` dans une image de côté `n`.

9. Ecrivez une fonction `nouvelle_image : int -> image`, telle que `nouvelle_image n` construit une nouvelle image à `n` lignes et `n` colonnes qui représente l'état initial du jeu (la structure `plateau`, en particulier, est laissée dans son état initial sans union).
   Indice : il faut créer vous même la table de hachage des couleurs, et y insérer la couleur de chaque case.
   Conseil : Vous représenterez le joueur A par la couleur `-1` et le joueur B par la couleur `-2`.

10. Ecrivez une fonction `trouver_image : image -> int -> int -> int*int`, telle que `trouver_image img i j` renvoie un couple `(repr,col)`, où `repr` est le représentant de la case de coordonnées `(i,j)` dans `plateau` et `col` est la couleur de ce représentant.
11. Ecrivez une fonction `afficher_image : image -> unit`, qui prend une image et l'affiche sous la même forme que les exemples donnés plus hauts. En particulier, vous afficherez les caractères `A` et `B` pour désigner les cases appartenant à chacun des joueurs.

III Unification de composantes

On s'intéresse dans cette partie aux composantes connexes d'une image. Une composante connexe est un ensemble connexe de taille maximale, composé de cases de la même couleur. On ne tient pas compte des diagonales. On peut les construire avec des opérations sur la structure Unir-Trouver `plateau` d'une image.
Ci-contre en exemple, les composantes connexes de l'état initial du jeu donné en exemple.



12. Ecrire une fonction `unir_meme_couleur : image -> int -> int -> int -> int -> unit` telle que `unir_meme_couleur img i j k l` réalise l'union des composantes des cases `(i,j)` et `(k,l)` dans l'image, en supposant qu'elles aient la même couleur.

    Indice : La table `couleurs` ne doit conserver que les couleurs des représentants des composantes. Il faut donc supprimer une entrée...

13. *(Question à développer à l'oral)* Proposez un algorithme pour calculer les composantes connexes présentes dans une image (le calcul est réalisé dans le champ `plateau`, que l'algorithme va modifier).
14. Ecrivez une fonction `construire_composantes_connexes : image -> unit`, qui implémente la fonction précédente.
15. Reformulez la Règle 3 du jeu des 7 couleurs en utilisant la notion de composante connexe.
16. Ecrivez une fonction `unir_avec_joueur : image -> int -> int -> int -> int -> unit` , telle que :
    - si la case de coordonnées `(i,j)` appartient à un des deux joueurs
    - si la case de coordonnées `(k,l)` appartient à une composante que le joueur s'apprête à annexer (on suppose qu'on a déjà vérifié que le coup était autorisé)

    l'appel `unir_avec_joueur img i j k l` fait annexer au joueur concerné la composante de `(k,l)`. Vous serez attentifs à la couleur de la nouvelle classe.

IV Fonctions du jeu

Avec les fonctions d'union de composantes connexes définies plus haut, nous pouvons maintenant implémenter les fonctions qui permettent de jouer un coup.

17. Ecrivez une fonction `voisines : image -> int -> int -> int -> int*int list` telle que `voisines img i j col` renvoie la liste des coordonnées des cases voisines (sans compter les diagonales) de la case `(i,j)` qui ont pour couleur `col`.
18. Ecrivez une fonction `coup : image -> int -> int -> unit` telle que `coup img p col` fait jouer au joueur dont la couleur est `p` (donc -1 ou -2) le coup "couleur `col`", selon la règle que vous avez exprimée à la question 13.
19. *(Question à développer à l'oral)* Pourrait-on adapter le type `image` pour réaliser plus efficacement la fonction précédente ? On aurait notamment besoin de garder facilement l'information du voisinage de chaque composante connexe. Quels effet cela aurait sur les fonctions précédentes ?
20. Ecrivez une fonction `score : img -> int*int` qui, pour une image donnée, renvoie un couple `(s1,s2)` où `s1` (resp. `s2`) est le nombre de cases qui appartiennent au joueur `A` (resp. `B`).
21. Ecrivez une fonction `fin_de_partie : img -> bool` qui, pour une image donnée, renvoie un booléen qui teste si la partie est finie ou non (selon la Règle 4 du jeu).
22. Utilisez la fonction `jouer_partie` fournie dans le code pour simuler une partie.

Annexes

Vous trouverez ici des rappels concernant certains traits du langage OCaml

Types enregistrements

- Déclaration : `type nom = { champ_1 : type_1 ; ...; champ_k : type_k }`
- Pour un champ mutable : `mutable champ_1 : type_1`
- Créer un objet : `{ champ_1 = val_1 ; ...; champ_k = val_k }`
- Accéder à la valeur d'un champ : `objet.champ`
- Changer la valeur d'un champ : `objet.champ <- valeur`

Module Array

- `Array.init n f` crée le tableau `[| f(0) ; f(1) ; ...; f(n-1)|]`
- `Array.make n k` crée un tableau de taille `n` dont tous les éléments valent `k`
- `Array.length tab` renvoie la longueur du tableau `tab`
- `tab.(i)` récupère le `i`-ème élement du tableau `t`
- `Array.mem x tab` renvoie un booléen spécifiant si `x` apparaît dans `tab`
- `Array.map f tab` renvoie le tableau `[| f(a.(0)) ; f(a.(1)) ; ...; f(a.(n-1))|]`
- `Array.iter f tab` exécute dans cet ordre `f(a.(0)), f(a.(1)), ..., f(a.(n-1))`

Module Hashtbl

- `Hashtbl.create n` crée une table de hachage de taille initiale `n` (elle peut stocker plus d'éléments si nécessaire, ce travail est réalisé directement par OCaml sans que vous n'ayez à le comprendre)
- `Hashtbl.add tab x y` ajoute l'association (clé `x`, valeur `y`) dans la table `tab`.
- `Hashtbl.find tab x` renvoie l'entrée associée à `x` dans la table si elle existe, une erreur sinon.
- `Hashtbl.find_opt tab x` renvoie `Some y` où `y` est l'entrée associée à `x` dans la table si elle existe, et `None` sinon.
- `Hashtbl.mem tab x` renvoie un booléen spécifiant si `x` apparaît dans `tab`
- `Hashtbl.remove tab x` supprime l'entrée associée à `x` dans la table `tab`.
- `Hashtbl.iter f tab` exécute `f` sur chacun des couples (clé,valeur) présents dans la table.

Module Random

- `Random.self_init ()` initialise le générateur de nombre aléatoires. (Sinon, il vous générera toujours les mêmes nombres.)
- `Random.int n` choisit un entier aléatoirement entre 0 et $n-1$.

Fiche réponse

NOM :
Prénom :
Répondez ici aux questions de cours, en rédigeant vos réponses.