OPTION INFORMATIQUE
Devoir surveillé 3

1 Coloration de graphe et logique propositionnelle

Soient $n \in \mathbb{N}^*$, G un graphe non orienté avec n sommets numérotés de 0 à $n-1$. Soit $k \in \mathbb{N}, k > 1$, on appelle **couleur** un entier dans $[\![0, k-1]\!]$, et **k-coloriage** de G une fonction associant à chaque sommet une couleur de façon à ce que deux sommets voisins n'aient pas la même couleur. On dit que G est **k-coloriable** s'il admet un k-coloriage.

1. On considère dans cette question le graphe G_1 représenté par listes d'adjacence par

   ```
   [| [1;2;3]; [0;2;5]; [0;1;4]; [0;4;5]; [2;3;5]; [1;3;4] |]
   ```

   Montrer que G_1 est 3-coloriable mais n'est pas 2-coloriable.
2. On se propose de traduire le problème de la k-coloriabilité d'un graphe G par des formules de la logique propositionnelle. Pour cela, on introduit un ensemble de variable

   $$V = \{C_{i,j} \mid i \in [\![0, n-1]\!], j \in [\![0, k-1]\!]\}$$

   où la variable $C_{i,j}$ code le fait que le sommet i a reçu la couleur j.
   Combien de lignes aurait une table de vérité énumérant toutes les valuations booléennes construites sur V ?
3. Soit $\phi = \bigwedge_{i=0}^{n-1} \bigwedge_{j=1}^{k-1} \left(C_{i,j} \to \left(\bigvee_{i'=0}^{n-1} C_{i',j-1} \right) \right)$
   (a) Traduire de façon simple, sans la paraphraser, la propriété du coloriage que décrit ϕ.
   (b) La formule ϕ est-elle satisfiable ? Démontrer votre réponse.
   (c) La formule ϕ est-elle valide ? Démontrer votre réponse.
4. Écrire une formule de logique propositionnelle ψ_1 traduisant le fait que chaque sommet doit recevoir au moins une couleur.
5. Écrire une formule ψ_2 traduisant le fait que chaque sommet ne peut pas recevoir plus d'une couleur.
6. Écrire une formule ψ_3 traduisant le fait que deux sommets voisins ne peuvent pas recevoir une même couleur
7. Justifier que G est k-coloriable si et seulement si la formule $\psi_1 \wedge \psi_3$ est satisfiable.

2 Pile avec oubli

On s'intéresse ici à une structure de données de pile avec oubli, similaire à une pile avec capacité, mais dans laquelle un empilement est toujours possible. Dans le cas où on empile une nouvelle valeur alors que la capacité est déjà atteinte, la valeur en fond de pile est retirée en compensation. Cette structure de données est par exemple utilisée par les éditeurs de documents pour stocker les versions d'un fichier et permettre d'annuler les dernières modifications.
Par souci de simplicité, on considère des piles d'entiers. On suppose que la capacité `c` est une variable globale. Les opérations disponibles sont :

- `creer_pile ()` : renvoie une pile avec oubli de capacité c,
- `est_vide p` : détermine si la pile `p` est vide,
- `empiler p v` : modifie la pile `p` en ajoutant la valeur `v` au sommet. Si la capacité est dépassée, la valeur en fond de pile est retirée.
- `depiler p` : modifie la pile `p` en retirant le sommet, et renvoie cette valeur. Si la pile est vide, une erreur est provoquée à la place.

Implémenter ces opérations en Caml à l'aide de tableaux. Chaque opération devra avoir une complexité en $O(1)$. On expliquera sur un exemple la représentation utilisée.

3 Arbres binaires de recherche

Dans cette partie, on considère des arbres binaires définis par le type Caml

```
type 'a arbre = V | N of 'a arbre * int * 'a * 'a arbre;;
```

Pour un noeud $N(g, c, x, d)$, g correspond au fils gauche de ce noeud, c à la clé associée au noeud, x à la valeur associée au noeud, et d au fils droit de ce noeud.

Si a est un arbre et n un entier, on notera $a < n$ si toutes les clés de a sont strictement inférieures à n. On définit de même $a \leq n, a > n, a \geq n$.

1. En utilisant ces notations, donner une définition inductive de la propriété pour un arbre binaire d'être un arbre binaire de recherche (on utilisera des inégalités strictes sur les clés).
2. (a) Écrire une fonction `valeur_associee` prenant en argument un arbre binaire de recherche et une clé entière, et renvoyant sa valeur associée. Une erreur sera levée si la clé n'apparaît pas dans l'arbre.
   (b) Déterminer la complexité de cette fonction.
3. (a) Écrire une fonction `maptree` prenant en argument une fonction `f : int -> int` et un arbre binaire a et renvoyant l'arbre obtenu à partir de a en appliquant f à chaque clé.
   (b) Donner une condition nécessaire et suffisante sur f pour que `maptree f` transforme tout arbre binaire de recherche en arbre binaire de recherche.
   (c) Démontrer rigoureusement le caractère nécessaire et suffisant de la condition précédente.
4. (a) Écrire une fonction récursive `scission` prenant en argument un arbre binaire de recherche a et un entier n et renvoyant un couple d'arbres binaires de recherche a_1, a_2 contenant ensemble les mêmes couples (clé,valeur) que a et vérifiant $a_1 \leq n < a_2$.
   (b) Déterminer la complexité de la fonction précédente.
5. En déduire une fonction récursive `fusion` prenant en argument deux arbres binaires de recherche a_1 et a_2 et renvoyant un arbre binaire de recherche contenant l'ensemble de leurs couples (clé,valeur). On supposera que a_1 et a_2 ne contiennent aucune clé en commun.

4 Minimum exclus

L'enjeu de cet exercice est de calculer le minimum exclus (ou **mex**) d'un tableau d'entiers naturels, c'est-à-dire le plus petit entier naturel qui n'apparaît pas dans le tableau. Par exemple, le mex de `[|2; 1; 0; 5|]` est 3, le mex de `[|4;7|]` est 0.

1. Donner sans justification le mex de `[|3; 6; 0|]`
2. (a) Écrire une fonction `appartient` prenant en argument un tableau d'entiers t et un entier x, et testant si x est dans t.
   (b) En déduire une fonction `mex1` calculant le mex d'un tableau d'entiers naturels.
   (c) Déterminer la complexité, dans le pire cas et le meilleur cas, de la fonction précédente, en fonction de la longueur n du tableau et de son maximum m.
3. (a) Écrire une fonction `presences` prenant un tableau t d'entiers naturels et renvoyant un tableau p de $m+1$ booléens, où m est le maximum de t, tel que `p.(k)` indique si k est présent dans t.
   (b) En déduire une autre implémentation `mex2` du minimum exclus.
   (c) Déterminer la complexité dans le pire et le meilleur cas de `mex2`, en fonction de n et m.
   (d) Démontrer formellement, à l'aide d'un invariant de boucle, la correction de `mex2` (en supposant que `presences` est correcte).
4. Écrire une variante `mex3` améliorant la complexité dans le meilleur ou le pire cas des versions précédentes, en utilisant un dictionnaire. On utilisera les appels suivants (tous de complexité constante) :
   - `Hashtbl.create 0` renvoie un dictionnaire vide ;
   - `Hashtbl.add h c v` ajoute au dictionnaire `h` la clé `c` associée à la valeur `v` ;
   - `Hashtbl.mem h c` teste si la clé `c` apparait dans le dictionnaire `h`.

   On justifiera qu'on obtient bien les complexités désirées.