MP2I – DS5 2023-2024 – 4h – sujet A

Pour chaque question, le langage imposé par l'énoncé doit être respecté. La rédaction doit être soignée.

Les fonctions doivent être documentées et commentées à bon escient. Le code doit être indenté et d'un seul tenant (pas de flèche pour dire qu'un bout de code se situe ailleurs par exemple). Le code illisible ne sera pas corrigé.

Sauf précision contraire, les réponses doivent être justifiées.

Tout document interdit. Calculatrice interdite. Téléphone éteint et rangé dans le sac. Pas de montre connectée.

Ce qui est illisible ou trop sale ne sera pas corrigé.
Soulignez ou encadrez vos résultats.

Toute fonction demandée dans une question peut être utilisée dans une question ultérieure, même si elle n'a pas été implémentée. Vous pouvez à tout moment choisir d'introduire des fonctions intermédiaires non explicitement demandées, mais elles doivent être précisément documentées.

On considère le jeu *Lights Out*[1] qui se joue sur une grille carrée de côté N pour un certain entier $N > 3$, chaque case contenant une ampoule. Le site Wikipedia décrit ce jeu de la manière suivante : "Quand le jeu commence, un nombre aléatoire [...] de ces lumières s'allument. Appuyer sur l'une des lumières basculera la position des lumières adjacentes à celle-ci [ainsi que la lumière concernée]. Le but du jeu est d'éteindre toutes les lumières, de préférence avec le moins de coups possibles."
Une *configuration* de *Lights Out* est la donnée des cases allumées et éteintes de la grille. La *grille finale* est la configuration correspondant à une grille où toutes les lumières sont éteintes.

PREMIÈRE PARTIE : TABLES DE HACHAGE

Cette partie est à traiter intégralement en `OCaml`.
Pour rappel, le module `Array` possède une fonction `make : int -> 'a -> 'a array` qui permet de créer un tableau dont toutes les cases ont la même valeur, en spécifiant sa longueur et cette valeur.

On cherche dans cette partie à créer une structure permettant de conserver la liste des configurations déjà obtenues lors d'une partie de *Lights Out*, avec pour chacune le nombre minimal de coups pour l'atteindre à partir de la configuration initiale, de sorte à ce que les opérations de recherche, d'ajout et de mise à jour soient rapides.
On va utiliser à cette fin un dictionnaire. Pour rappel, une structure de dictionnaire est un ensemble de couples (clef, élément), les clefs (nécessairement distinctes) appartenant à un même ensemble K et les éléments à un ensemble E.
Dans l'exemple de *Lights Out*, les clefs sont les configurations du jeu et les éléments sont des entiers contenant la longueur d'une suite de coups allant de la configuration initiale à la grille finale.

Dans la suite, un dictionnaire sera réalisé à l'aide d'une table de hachage possédant w alvéoles et dont les collisions sont résolues par chaînage. On appelle w la *largeur de la table* et on note $h_w : K \to [\![0, w-1]\!]$ la fonction de hachage associée à la table.
On suppose dorénavant que N (la longueur du côté de la grille) est donné par une variable entière `n`.

1 Tables de hachage de largeur fixée

Dans cette sous-section, on fixe une largeur de hachage w. On définit le type suivant :

```
type ('a, 'b) table_hachage = {
    hache: 'a -> int;
    donnees: ('a * 'b) list array;
    largeur: int
  }
```

1. Et je ne peux pas faire un sujet sur ce jeu sans remercier Balthazar qui m'a involontairement inspirée pour ce choix.

1.1 Quelques fonctions utilitaires

Question 1 : Écrire une fonction `mem1 x q` de signature `'a -> ('a * 'b) list -> bool` testant l'existence d'un couple dont le premier élément est `x` dans la liste `q`, de complexité temporelle dans le pire des cas linéaire en la longueur de la liste (à justifier brièvement).

Question 2 : Prouver que la fonction `mem1` de la question précédente possède une correction totale.

Question 3 : Écrire une fonction `assoc x q` de signature `'a -> ('a * 'b) list -> 'b` renvoyant, s'il existe, l'élément `y` du premier couple de la forme `(x,y)` appartenant à la liste `q`, et levant l'exception `Not_found` sinon.

Question 4 : Écrire une fonction `suppr x q` de signature `'a -> ('a * 'b) list -> ('a * 'b) list` qui s'évalue en une liste contenant les mêmes éléments que `q`, excepté le premier couple de la forme `(x, y)` de la liste `q` s'il en existe.

1.2 Implantation de la structure de dictionnaire

Question 5 : Écrire une fonction `creer_table h w : ('a -> int) -> int -> ('a, 'b) table_hachage` telle que `creer_table h w` renvoie une nouvelle table de hachage vide de largeur `w` et munie de la fonction de hachage `h`.

Question 6 : Écrire une fonction `recherche t k : ('a, 'b) table_hachage -> 'a -> bool` renvoyant un booléen indiquant si la clé `k` est présente dans la table `t`.

Question 7 : Écrire une fonction `element t k : ('a, 'b) table_hachage -> 'a -> 'b` renvoyant l'élément `e` associé à la clef `k` dans la table `t`, si cette clef est présente dans la table, et levant l'exception `Not_found` sinon.

Question 8 : Écrire une fonction `ajout t k e : ('a, 'b) table_hachage -> 'a -> 'b -> unit` ajoutant l'entrée `(k, e)` à la table de hachage `t`. On n'effectuera aucun changement si la clef est déjà présente.

Question 9 : Écrire une fonction `suppression t k : ('a, 'b) table_hachage -> 'a -> unit` supprimant l'entrée de la clef `k` dans la table `t`. On n'effectuera aucun changement si la clef n'est pas présente.

Question 10 : Écrire une fonction `mise_a_jour t k e : ('a, 'b) table_hachage -> 'a -> 'b -> unit` modifiant la valeur associée à `k` dans la table de hachage `t`, de sorte que cette valeur soit `e`. Si la clef n'est pas présente, le couple `(k, e)` sera ajouté à la table de hachage.

1.3 Étude de la complexité de la recherche d'un élément

Nous étudions ici la complexité de la recherche d'une clef dans une table de hachage où les collisions sont gérées par chaînage.

Question 11 : Montrer que dans le pire des cas, la complexité de recherche d'une clef dans une telle table de hachage est en $\Theta(n)$, où n est le nombre de couples stockés dans la table de hachage.

Si la fonction de hachage h_w est bien choisie, on peut espérer que les clefs vont se répartir de façon relativement uniforme dans les alvéoles, ce qui donnera une complexité bien meilleure.
Nous faisons donc ici l'hypothèse de *hachage uniforme simple* : pour une clef donnée, la probabilité d'être hachée dans l'alvéole i est $1/w$, indépendante des autres clefs. On note n le nombre de clefs stockées dans la table et on appelle $\alpha = n/w$ le *facteur de remplissage* de la table. On suppose de plus, que le calcul de la valeur hachée d'une clef se fait en temps constant.

Question 12 : On se donne une clef k non présente dans la table. Montrer que la complexité en moyenne de la recherche de k dans la table est en $O(1+\alpha)$.

Question 13 : On prend au hasard une clef présente dans la table ; toutes les clefs sont équiprobables. Montrer qu'alors la recherche de la clef se fait en $O(1+\alpha)$, en moyenne sur toutes les clefs présentes.

1.4 Une fonction de hachage h_w pour le jeu *Lights Out*

Construisons un fonction de hachage h_w, pour une liste de couples de $[\![0, N-1]\!]^2$, cette liste correspondant aux coordonnées des cases allumées dans une configuration de *Lights Out*, en lisant les cases de gauche à droite et de haut en bas (comme quand on lit un texte en Français). Un hachage naturel d'une liste comportant les couples

$(a_i, b_i)_{0\leq i<p}$ avec $0 \leq a_i, b_i < N-1$ est donné par :

$$P_w(N) = \left(\sum_{i=0}^{p-1}(a_i + b_i N)N^{2i}\right) \text{ modulo } w$$

On suppose que le type `int` d'`OCaml` est suffisant pour représenter tous les entiers manipulés dans l'énoncé.

Question 14 : Écrire une fonction récursive `hachage_liste w q : int -> (int * int) list -> int` calculant la quantité précédente. Cette fonction doit être linéaire en la longueur de la liste passée en argument.

DEUXIÈME PARTIE : JEUX À UN JOUEUR ET SOLUTIONS OPTIMALES

Cette partie est à traiter intégralement en `C`.
On suppose qu'on dispose des types suivants :

- `configuration` pour représenter une configuration du jeu ;
- `ensemble` pour représenter un ensemble (au sens mathématique du terme : aucun doublon possible) de configurations.

On suppose également disposer des fonctions suivantes pour gérer les ensembles de configuration :

```
/** renvoie un ensemble vide */
ensemble ensemble_vide();
/** teste si l'ensemble e est vide */
bool est_vide(ensemble e);
/** ajoute la configuration c dans l'ensemble e */
void ajout(ensemble *e, configuration c);
/** teste si la configuration c appartient à l'ensemble e */
bool appartient(ensemble e, configuration c);
/** ajoute les éléments de l'ensemble e2 à l'ensemble e1 */
void union(ensemble *e1, ensemble e2);
/** renvoie une configuration de e s'il est non vide en la supprimant de e,
* NULL sinon */
configuration *un_element(ensemble *e);
/** libère la mémoire associée à l'ensemble e */
void liberation(ensemble e);
```

L'objectif de cette partie est d'étudier différents algorithmes pour trouver des solutions minimales en nombre de coups pour le jeu *Lights Out*.
La section 2 introduit la notion de jeu à un joueur et un premier algorithme pour trouver une solution optimale. La section 3 propose un algorithme plus efficace.

2 Jeu à un joueur, parcours en largeur (BFS : *Breadth-First Search*)

Un jeu à un joueur est la donnée d'un ensemble non vide E, d'un élément $e_0 \in E$, d'une fonction $s : E \to \mathscr{P}(E)$, où $\mathscr{P}(E)$ désigne l'ensemble des parties de E, et d'un sous-ensemble F de E. L'ensemble E représente les configurations possibles du jeu. L'élément e_0 est la configuration initiale. Pour une configuration e, l'ensemble $s(e)$ représente toutes les configurations atteignables en un coup à partir de e. Enfin, F est l'ensemble des configurations gagnantes du jeu. On dit qu'une configuration e_p est à la *profondeur* p s'il existe une séquence finie de $p+1$ configurations

$$e_0\ e_1\ \ldots\ e_p$$

avec $e_{i+1} \in s(e_i)$ pour tout $0 \leq i < p$. Si par ailleurs $e_p \in F$, une telle séquence est appelée une *solution* du jeu, de profondeur p. Une solution *optimale* est une solution de profondeur minimale. On notera qu'un même configuration peut être à plusieurs profondeurs différentes.
Voici un exemple de jeu :

$$\begin{aligned} E &= \mathbb{N}^* \\ e_0 &= 1 \\ s(n) &= \{2n, n+1\} \end{aligned} \tag{1}$$

Question 15 : Donner une solution optimale pour le jeu (1) lorsque F = {42} (sans avoir à justifier son optimalité).

Parcours en largeur. Pour chercher une solution optimale pour un jeu quelconque, on peut utiliser un parcours en largeur. Un pseudo-code pour un tel parcours est donné par l'algorithme 1.

Algorithme 1 BFS : Parcours en largeur d'un jeu d'ensemble de configurations E et de fonction de succession $s : E \to \mathscr{P}(E)$

entrée : configuration initiale $e_0 \in E$, ensemble de configurations gagnantes $F \subseteq E$
sortie : teste l'existence d'une solution *via* un parcours en largeur des configurations du jeu

```
1 : A ← {e_0}
2 : tant que A ≠ ∅ faire
3 :     B ← ∅
4 :     pour tout x ∈ A faire
5 :         si x ∈ F alors
6 :             renvoyer vrai
7 :         B ← B ∪ s(x)
8 :     A ← B
9 : renvoyer faux
```

Question 16 : Montrer que le parcours en largeur renvoie vrai si et seulement si une configuration gagnante est atteignable à partie de e_0 (c'est-à-dire que l'algorithme 1 possède une correction totale).

Question 17 : On se place dans le cas particulier du jeu (1) pour un ensemble F arbitraire pour lequel le parcours en largeur donné par l'algorithme 1 termine. Montrer alors que la complexité en temps et en espace est exponentielle en la profondeur p de la solution trouvée. On demande de montrer que la complexité est bornée à la fois inférieurement et supérieurement par deux fonctions exponentielles en p.

Question 18 : On suppose qu'on dispose d'une fonction `ensemble s(configuration c)` qui donne toutes les configuration atteignables en un coup à partir de `c`. Écrire une fonction

```
int bfs(configuration e0, ensemble F)
```

qui effectue un parcours en largeur à partir de la configuration initiale `e0` et renvoie la profondeur de la première solution trouvée dans `F`. Lorsqu'il n'y a pas de solution, le comportement de cette fonction pourra être arbitraire.

Question 19 : Montrer que la fonction `bfs` renvoie toujours une profondeur optimale lorsqu'une solution existe.

3 Parcours en profondeur (DFS : *Depth-First Search*)

Comme on vient de le montrer, l'algorithme BFS permet de trouver une solution optimale mais il peut consommer un espace important pour cela, comme illustré dans le cas particulier du jeu (1) qui nécessite un espace exponentiel. On peut y remédier, on utilise un parcours en profondeur. L'algorithme 2 contient le pseudo-code d'une fonction DFS effectuant un parcours en profondeur à partir d'une configuration e de profondeur p, sans dépasser une profondeur maximale m donnée.

Algorithme 2 DFS : Parcours en profondeur, limité par une profondeur maximale, d'un jeu d'ensemble de configurations E, d'ensemble de configurations gagnantes $F \subseteq E$ et de fonction de succession $s : E \to \mathscr{P}(E)$

entrée : configuration $e \in E$, entiers p et m
sortie : teste l'existence d'une solution à partir de e de profondeur p, *via* un parcours en profondeur des configurations du jeu, sans dépasser la profondeur maximale m

```
1 : si p > m alors
2 :     renvoyer faux
3 : si e ∈ F alors
4 :     renvoyer vrai
5 : pour tout x ∈ s(e) faire
6 :     si DFS(x, p + 1, m) est vrai alors
7 :         renvoyer vrai
8 : renvoyer faux
```

Question 20 : Montrer que $\mathrm{DFS}(e_0, 0, m)$ renvoie vrai si et seulement si une solution de profondeur inférieure ou égale à m existe.

Recherche itérée en profondeur. Pour trouver une solution optimale, une idée simple consiste à effectuer un parcours en profondeur avec $m = 0$, puis avec $m = 1$, puis avec $m = 2$, etc., jusqu'à ce que DFS($e_0, 0, m$) renvoie vrai.

Question 21 : Écrire une fonction

```
int ids(configuration e0, ensemble F);
```

qui effectue une recherche itérée en profondeur et renvoie la profondeur d'une solution optimale. Lorsqu'il n'y a pas de solution, cette fonction ne termine pas.

Question 22 : Montrer que la fonction `ids` renvoie toujours une profondeur optimale lorsqu'une solution existe.

Question 23 : Comparer les complexités en temps et en espace du parcours en largeur et de la recherche itérée en profondeur dans les deux cas particuliers suivants :

1. il y a exactement une configuration à chaque profondeur p ;
2. il y a exactement 2^p configurations à la profondeur p.

On demande de justifier les complexités qui seront données.