COMPOSITION D'INFORMATIQUE (Option) n°2

Durée : 4h

⋆ ⋆ ⋆

Étude du jeu Ricochet Robots

À travers l'étude d'un jeu de société, ce sujet s'intéresse aux mouvements de robots, qui possèdent des capacités limitées de localisation. Avec le développement de la robotique, plusieurs problèmes de ce type font l'objet de nombreuses recherches : parcours minimum pour examiner une surface donnée, stratégies collectives avec plusieurs robots en interaction proches, nombre de robots nécessaires pour que tous les points d'une surface avec obstacels soient accessibles, etc.
Ce sujet porte sur la résolution de la sitation pratique du jeu « Ricochet Robots » créé par Alex Randolph en 1999. Ce jeu se déroule sur un plateau de 16×16 cases, avec 4 robots et des murs. À chaque mouvement, un des robots se déplace, dans une des quatre directions, jusqu'à ce qu'il rencontre un obstacle (un mur ou un autre robot). Le but est trouver le nombre de coups minimal pour déplacer un robot particulier de son point de départ jusqu'à une case précise du plateau. Un exemple en 8 coups est donné figure 1.

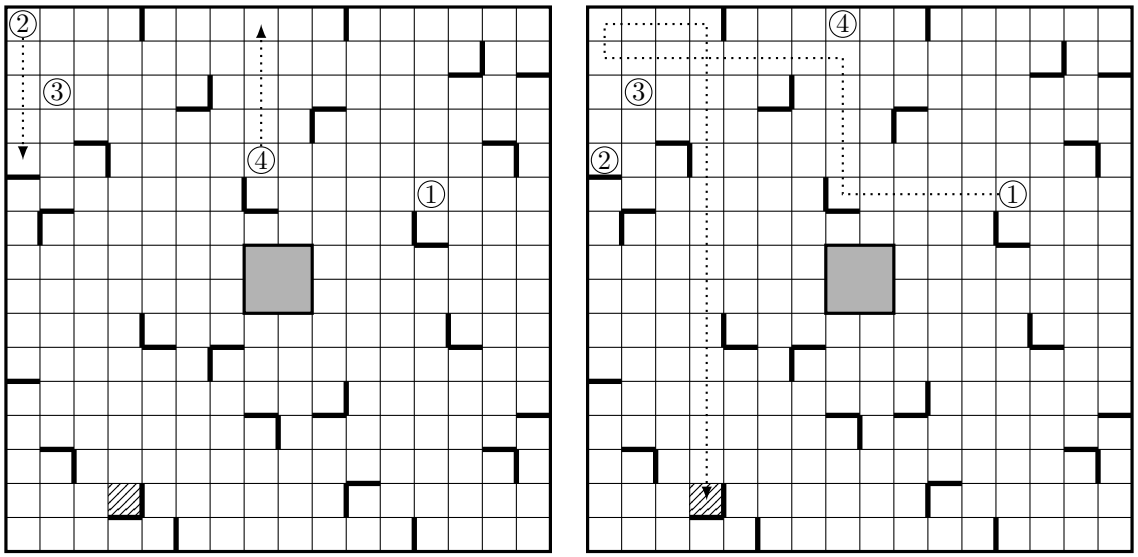

Figure 1 : Le jeu des robots : le but est d'amener le robot 1 sur la case hachurée. À gauche : deux déplacements des robots 2 et 4 ; à droite : six déplacements du robot 1. Le jeu est résolu en 8 mouvements (solution optimale)

On rappelle la définition des fonctions suivantes, disponibles dans la bibliothèque standard de Caml :

- `Array.copy : 'a array -> 'a array` telle que l'appel `Array.copy v` renvoie un nouveau tableau contenant les valeurs contenues dans `v`.
- `Array.make : int -> 'a -> 'a array` telle que l'appel `Array.make n x` renvoie un nouveau tableau de longueur `n` initialisé avec des éléments égaux à `x`.
- `Array.make_matrix : int -> int -> 'a -> 'a array array` telle que l'appel `Array.make_matrix p q x` renvoie une nouvelle matrice à `p` lignes et `q` colonnes initialisée avec des éléments égaux à `x`.

I Déplacement d'un robot dans une grille

On considère pour le moment une grille sans robots du jeu Ricochet Robots. Notons n le nombre de cases par ligne et colonne de la grille (16 dans le jeu originel). *Dans les fonctions demandées, on supposera que n est une variable globale.* On numérote chaque case par un couple (a, b) de $[\![0; n-1]\!]^2$, correspondant à la ligne a et à la colonne b. On numérote également les lignes horizontales et verticales séparant les cases à l'aide d'un entier de $[\![0; n]\!]$, de sorte que la case (a, b) est délimitée par les lignes horizontales a (au dessus) et $a+1$ (en dessous), de même que par les lignes verticales b (à gauche) et $b+1$ (à droite) (figure 2).

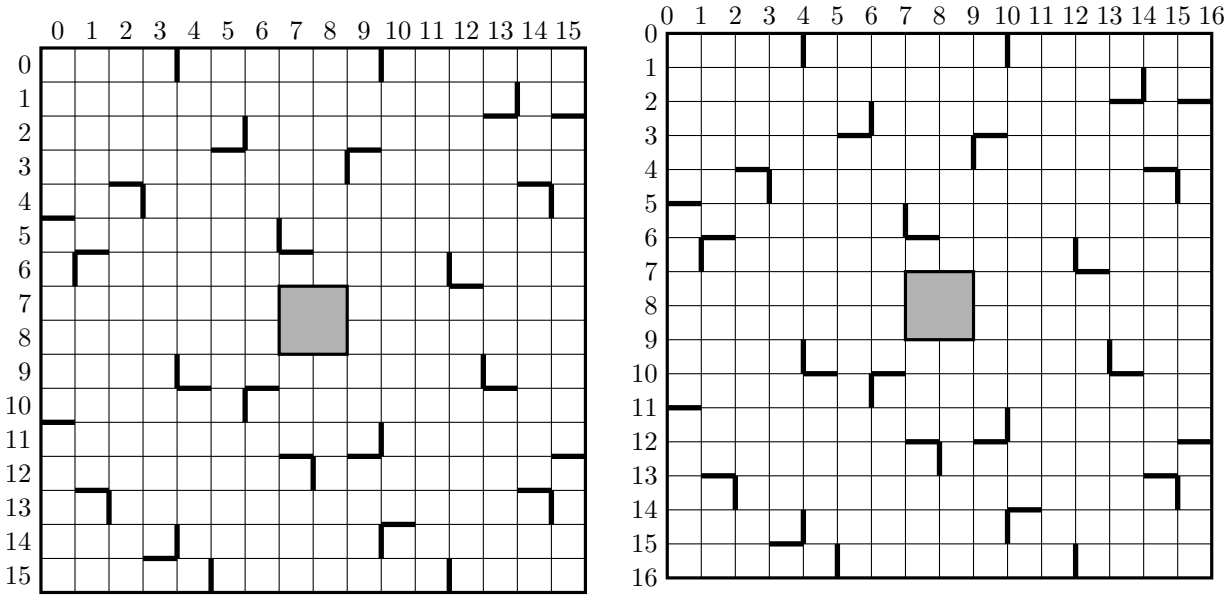

Figure 2 : À gauche : numérotation des cases par ligne/colonne ; à droite : numérotation des lignes horizontales et verticales

Pour représenter en Caml la grille avec ses obstacles, on se donne deux tableaux (vecteurs) de taille n. Le premier contient les obstacles verticaux sur chacune des lignes, le second contient les obstacles horizontaux sur chacune des colonnes. Un obstacle est donné par le numéro de la ligne (verticale ou horizontale) auquel il appartient. Les obstacles sur une ligne (ou colonne) sont donnés sous la forme d'un tableau ordonné dans l'ordre croissant. Par exemple, la représentation du plateau de la figure 1 est donnée figure 3.

```
let ob_li = [| [|0; 4; 10; 16|]; [|0; 14; 16|]; [|0; 6; 16|]; [|0; 9; 16|];
  [|0; 3; 15; 16|]; [|0; 7; 16|]; [|0; 1; 12; 16|]; [|0; 7; 9; 16|];
  [|0; 7; 9; 16|]; [|0; 4; 13; 16|]; [|0; 6; 16|]; [|0; 10; 16|];
  [|0; 8; 16|]; [|0; 2; 15; 16|]; [|0; 4; 10; 16|]; [|0; 5; 12; 16|] |];;

let ob_co = [| [|0; 5; 11; 16|]; [|0; 6; 13; 16|]; [|0; 4; 16|]; [|0; 15; 16|];
  [|0; 10; 16|]; [|0; 3; 16|]; [|0; 10; 16|]; [|0; 6; 7; 9; 12; 16|];
  [|0; 7; 9; 16|]; [|0; 3; 12; 16|]; [|0; 14; 16|]; [|0; 16|]; [|0; 7; 16|];
  [|0; 2; 10; 16|]; [|0; 4; 13; 16|]; [|0; 2; 12; 16|] |];;
```

Figure 3 : Exemple de représentation en Caml

Notez que les bordures de la grille sont considérées comme des obstacles. Ainsi, les entiers 0 et N sont présents dans les tableaux associés à chaque ligne/colonne.

Question 1 Écrire une fonction `dichotomie : int -> int array -> int` telle que si `t` est un tableau d'entiers strictement croissants et `a` un élément supérieur ou égal au premier élément du tableau et strictement inférieur au dernier, alors `dichotomie a t` renvoie l'unique indice `i` tel que `t.(i)` $\leq$ `a` $<$ `t.(i + 1)`. La fonction doit avoir une complexité logarithmique en la taille du tableau.

On considère un robot positionné en (a, b), avec $0 \leq a, b < n$. Il peut se déplacer dans les quatre directions cardinales ouest/est/nord/sud.

Question 2 Écrire une fonction `dpt_grille : int * int -> (int * int) array` fournissant les 4 cases atteintes par les déplacements en questions, sous forme d'un tableau à 4 éléments (ouest/est/nord/sud). Si le robot ne peut pas bouger dans une direction donnée (car il est contre un obstacle), on considèrera que le résultat du déplacement dans cette direction est la case (a, b) elle-même. Les deux tableaux `ob_li` et `ob_co` sont des variables globales.

Question 3 Écrire une fonction `mat_dpt : unit -> (int * int) array array` produisant une matrice `m` telle que `m.(a).(b)` contienne le vecteur des déplacements possibles pour un robot depuis la case (a, b), et ce pour tous $0 \leq a, b < n$. Donner la complexité de cette fonction.

On cherche maintenant à intégrer les positions d'autres robots dans le déplacement d'un robot. On utilise la fonction précédente pour créer une matrice `m` que l'on considèrera comme globale.

Question 4 Écrire une fonction `modif : (int * int) array -> int * int -> int * int -> unit` telle que si `t` est le tableau de taille 4 donnant les déplacements ouest/est/nord/sud d'un robot placé en `(a, b)` dans la grille ne contenant pas d'autre robots, et `(c, d)` la position d'un autre robot, alors `modif t (a, b) (c, d)` modifie si nécessaire le tableau `t` en prenant en compte le robot en `(c, d)`.

On s'intéresse maintenant au déplacement d'un robot situé en (a, b) dans la grille, avec d'autres robots éventuellement présents, dont les positions sont stockées dans une liste.

Question 5 Déduire des questions précédentes une fonction de signature :

```
dpt_robot : int * int -> (int * int) list -> (int * int) array
```

telle que `dpt_robot (a, b) q` donne les déplacements ouest/est/nord/sud d'un robot situé en (a, b) dans la grille, les positions des autres robots étant stockées dans la liste `q`. On ne modifiera pas la matrice `m` : on souhaite une copie modifiée de `m.(a).(b)`.

Question 6 Si on suppose que la solution optimale demande au plus k mouvements, une solution possible pour résoudre le jeu Ricochet Robots consiste à générer toutes les suites possibles de k déplacements. Avec 4 robots en tout, estimer la complexité d'une telle approche (on utilisera la notation O).

La suite du problème a pour objet de proposer une solution plus efficace pour la résolution du jeu Ricochet Robots.

II Tables de hachage

Dans l'optique de résoudre le problème du jeu des robots, nous allons travailler sur un graphe dont les sommets seront étiquetés par les positions des robots. Le nombre de sommets possibles étant élevé, il est nécessaire d'utiliser une structure de données adaptée pour travailler sur ce graphe. Nous allons donc réaliser une structure de dictionnaire permettant, en particulier, de tester facilement si un sommet a déjà été vu ou non et d'associer un sommet à chaque sommet découvert.
Une structure de dictionnaire est un ensemble de couples (clé, élément), les clés (nécessairement distinctes) appartenant à un même ensemble K, les éléments à un ensemble E. La structure doit garantir les opérations suivantes :

- recherche d'un élément connaissant sa clé ;
- ajout d'un couple (clé, élément) ;
- suppression d'un couple connaissant sa clé.

Une structure de dictionnaire peut-être réalisée à l'aide d'une table de hachage. Cette table est implantée dans un tableau de w listes (appelées **alvéoles**) de couples (clé, élément). Ce tableau est organisé de façon à ce que la liste d'indice i contienne tous les couples (k, e) tels que $h_w(k) = i$ où $h_w : K \leftarrow [\![0; w-1]\!]$ s'appelle **fonction de hachage**. On appelle w la **largeur** de la table de hachage et $h_w(k)$ le **haché** de la clé k.
Ainsi pour rechercher ou supprimer l'élément de clé k, on commence par calculer son haché qui détermine l'alvéole adéquate et on est alors ramené à une action sur la liste correspondante. De même pour ajouter un nouvel élément au dictionnaire on l'ajoute à l'alvéole indiquée par le haché de sa clé.

II.A Une famille de fonction h_w

Nous commençons par nous doter d'une famille de fonctions h_w, pour les listes de couples de $[\![0; n-1]\!]^2$. Un hachage naturel d'une liste comportant les couples $(a_i, b_i)_{i \in [\![0;p-1]\!]}$ avec $0 \leq a_i, b_i < n$ est donnée par :

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

Autrement dit, on évalue le polynôme dont les coefficients sont donnés par les a_i et b_i en n et on ne considère que le reste dans la division euclidienne par w. On rappelle qu'on a supposé que n est une variable globale.

Question 7 Écrire une fonction récursive `h : int -> (int * int) list -> int` telle que `h w q` calcule la quantité précédente.

II.B Complexité pour une largeur fixe

Nous étudions ici la complexité de la recherche d'une clé dans une table de hachage. Dans le pire cas, toutes les clés sont hachées vers la même alvéole, ainsi la complexité de la recherche d'une clé dans une table de hachage n'est pas meilleure que la recherche dans une liste. Cependant, si la fonction de hachage h_w est bien choisie, on peut espérer que les clés vont se répartir de façon apparemment aléatoire dans les alvéoles, ce qui donnera une complexité bien meilleure.
Nous faisons ici l'hypothèse de **hachage uniforme simple** : pour une clé donnée, la probabilité d'être hachée dans l'alvéole i est $\frac{1}{w}$, indépendante des autres clés. On note N le nombre de clés stockées dans la table et on appelle $\alpha = \frac{N}{w}$ le **facteur de remplissage** de la table. On suppose de plus que le calcul du haché d'une clé se fait en temps constant.
Enfin, on suppose que la complexité de recherche d'une clé dans une alvéole est linéaire en la taille de la liste présente dans l'alvéole.

Question 8 On se donne une clé k non présente dans la table. Montrer que l'espérance de la complexité de la recherche de k dans la table est un $O(1+\alpha)$.

Question 9 On prend au hasard une clé présente dans la table, les clés étant équiprobables. Montrer que la recherche de la clé se fait en $O(1+\alpha)$, en moyenne sur toutes les clés présentes.

II.C Tables de hachage dynamique

Les deux questions précédentes montrer que l'on peut assurer une complexité moyenne constante pour la recherche dans une table de hachage, sous réserve que le facteur de remplissage α soit borné. Il en va de même des opérations d'insertion et de suppression, pour peu que les clés à ajouter/supprimer vérifient des hypothèses d'indépendance. Bien souvent, et cela va être le cas dans notre problème, on ne sait pas à l'avance quel sera le nombre de clés à stocker dans la table, et on préfère ne pas surestimer ce nombre pour garder un espace mémoire linéaire en le nombre de clés stockées. Ainsi, il est utile de faire varier la largeur w de la table de hachage : si le facteur de remplissage devient trop important, on réarrange la table sur une largeur plus grande (de même, on peut réduire la largeur de la table lorsque le facteur de remplissage devient trop petit). On parle alors de tables de hachage dynamiques pour ces tables à largeur variable.
À une table de hachage dynamique est associée une famille de fonctions de hachage (h_w), comme la fonction `h` écrite précédemment. On définit en toute généralité le type suivant :

```
type ('a, 'b) table = {
   h: int -> 'a -> int;
   mutable taille: int;
   mutable data: ('a * 'b) list array;
   mutable w: int};;
```

Le champ `taille` permet ici de contenir le nombre de clés présentes dans la table.

Question 10 Écrire une fonction `creer_table h` qui crée une table de hachage dynamique initialement vide, avec la famille de fonctions de hachage `h` et la largeur initiale 1.

Question 11 Écrire une fonction `recherche : ('a, 'b) table -> 'a -> 'b` renvoyant l'élément e associé à la clé k dans la table t. La fonction écrira un message d'erreur le cas échéant.

On supposera écrite de manière similaire une fonction `mem : ('a, 'b) table -> 'a -> bool` qui teste la présence d'une clé dans la table.

Question 12 Écrire une fonction `rearrange t w2` qui prend en entrée une table de hachage dynamique et une nouvelle largeur de hachage `w2` et qui réarrange la table sur une largeur `w2`. On montrera que la complexité est en $O(N+w+w_2)$ où N est le nombre de clés présentes, w l'ancienne largeur et w_2 la nouvelle.

Une stratégie heuristique simple pour garantir que le facteur de remplissage reste borné, tout en garantissant une bonne répartition des clés dans le cas des listes de couples à valeurs dans $[\![0; n-1]\!]$ avec $n = 16$, est d'utiliser les puissances de 3 comme largeurs de hachage. Après l'ajout d'un élément à la table, si celle-ci est de taille strictement supérieure à trois fois sa largeur w, on la réarrange sur une largeur $w' = 3w$.

Question 13 Écrire une fonction `ajout t k e` qui ajoute le couple (k, e) à la table de hachage (si la clé k n'est pas présente), en réarrangeant si nécessaire la table, en suivant le principe ci-dessus.

III Résolution du jeu des robots

Un graphe orienté est un couple $G = (V, E)$ où V est un ensemble de **sommets** (*vertices* en anglais), généralement représenté par $[\![0; |V| - 1]\!]$, et $E \subset V \times V$ est un ensemble **d'arêtes** (*edges* en anglais). Pour $u, v \in V$, on appelle **chemin** de u à v une suite de sommets $u = v_0, v_1, \ldots, v_k = v$ telle que pour $i \in [\![0; k-1]\!]$, $(v_i, v_{i+1}) \in E$. k est appelé la **longueur** du chemin. La **distance** entre deux sommets u et v est la plus courte longueur d'un chemin de u à v.
Une représentation graphique consiste à représenter un point dans le plan pour chaque sommet, et une flèche d'un sommet v_1 vers un sommet v_2 si $(v_1, v_2) \in E$.

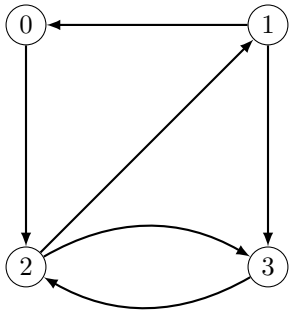

Figure 4 : Une représentation graphique du graphe $G = (\{0, 1, 2, 3\}, \{(0, 2), (1, 0), (1, 3), (2, 1), (2, 3), (3, 2)\})$

III.A Parcours en largeur : étude théorique

On se donne un sommet $v_0 \in V$ et on considère l'algorithme ci-dessous (parcours en largeur) :

Entrées : Un graphe $G = (V, E)$ orienté, un sommet de départ v_0
Sorties : Un tableau de booléens, un tableau de prédécesseurs
$F \leftarrow$ File vide
Enfiler v_0 dans F
$b[v] \leftarrow$ **faux** pour tout $v \in V$ (tableau de booléens)
$b[v_0] \leftarrow$ **vrai**
$p[v] \leftarrow v$ pour tout $v \in V$(tableau de prédécesseurs)
Tant que F est non vide **Faire**
    $v \leftarrow$ Défiler F
    Pour tout v' voisin de v tel que $b[v']$ est **faux Faire**
        $b[v'] \leftarrow$ **vrai**
        $p[v'] \leftarrow v$
        Enfiler v' dans F
Renvoyer b, p

Question 14 Montrer que l'algorithme termine.

Question 15 Montrer qu'après l'exécution de l'algorithme, $b[v]$ vaut **vrai** (on dit que v est **visité**) si et seulement s'il existe un chemin de v_0 à v dans le graphe.

Question 16 Pour un sommet v visité par l'algorithme, expliquer comment retrouver un chemin de v_0 à v.

Question 17 Pour $v \in V$, on note d_v la distance de v_0 à v, ℓ_v la longueur du chemin de v_0 à v donné par l'algorithme et n_v le rang d'insertion du sommet dans la file d'attente (le sommet v_0 est de rang 0). Prouver la propriété suivante par induction :

1. $d_v = \ell_v$.
2. Pour tout sommet u, $d_u < d_v \Rightarrow n_u < n_v$.

Cette propriété permet de déduire que le chemin trouvé par l'algorithme de parcours en largeur est un plus court chemin.

Un graphe sera implémenté par une structure dite de **liste d'adjacence** :

```
type graphe = int list array;;
```

Un graphe sera donc un tableau `g` de taille $|V|$ tel que `g.(i)` contient la liste des numéros des sommets voisins de `i` (par ordre croissant, par exemple).

```
let g = [| [2]; [0; 3]; [1; 3]; [2] |];;
```

Figure 5 : Une représentation Caml du graphe de la figure 4

Question 18 Déterminer la complexité du de l'algorithme de parcours en largeur en fonction de $|V|$ et $|E|$, en supposant que les opérations sur les files et sur les tableaux b et p se font en temps constant.

On suppose disposer d'un type `'a file` et des fonctions classiques associées :

- `creer_file : unit -> 'a file`
- `vide : 'a file -> bool`
- `enfiler : 'a file -> 'a -> unit`
- `defiler : 'a file -> 'a`

Question 19 Écrire une fonction `parcours : graphe -> int -> bool array * int array` qui effectue le parcours en largeur d'un graphe en partant d'un sommet donné.

III.B Graphe orienté associé au jeu des robots

La résolution du jeu des robots peut se faire en traduisant le problème sous forme d'un graphe dans lequel chaque sommet représente une position des robots sur le plateau de jeu. On distingue le « robot principal » (celui que l'on veut amener sur une case donnée) et les autres robots. Ainsi, un sommet est représenté par le type suivant :

```
type sommet = {r: int * int; q: (int * int) list};;
```

Pour chaque sommet, on impose que la liste `q` soit triée dans l'ordre lexicographique croissant (qui est l'ordre naturel des couples en Caml).
Les arêtes dans le graphe orienté sont définies naturellement : un sommet u est relié à un sommet v si on peut passer de u à v par un mouvement licite d'un des robots.

Question 20 Avec p robots en tout sur un plateau de taille $n \times n$, quel est le nombre possible de sommets ? Donner le nombre exacte pour $p = 4$ et $n = 16$.

Question 21 Écrire une fonction `accessibles : sommet -> sommet list` prenant en entrée un sommet et renvoyant la liste des sommets accessibles via un sommet `v` donné en argument à partir du déplacement d'un des robots (principal ou non). S'il y a p robots en tout, la fonction renverra une liste de $4p$ sommets, certains sommets pouvant être égaux au sommet `v` : ils correspondent au mouvement d'un robot dans une direction où il est bloqué.

⋆ ⋆ ⋆