CONCOURS CENTRALE•SUPÉLEC

Option informatique

MP

4 heures Calculatrices autorisées

2018

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 proche, nombre de robots nécessaires pour que tous les points d'une surface avec obstacles soient accessibles, etc.

Ce sujet porte sur la résolution de la situation pratique du jeu « Ricochet Robots » (*Rasende Roboter* pour la première édition en allemand) 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 autre robot). Le but est de 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.

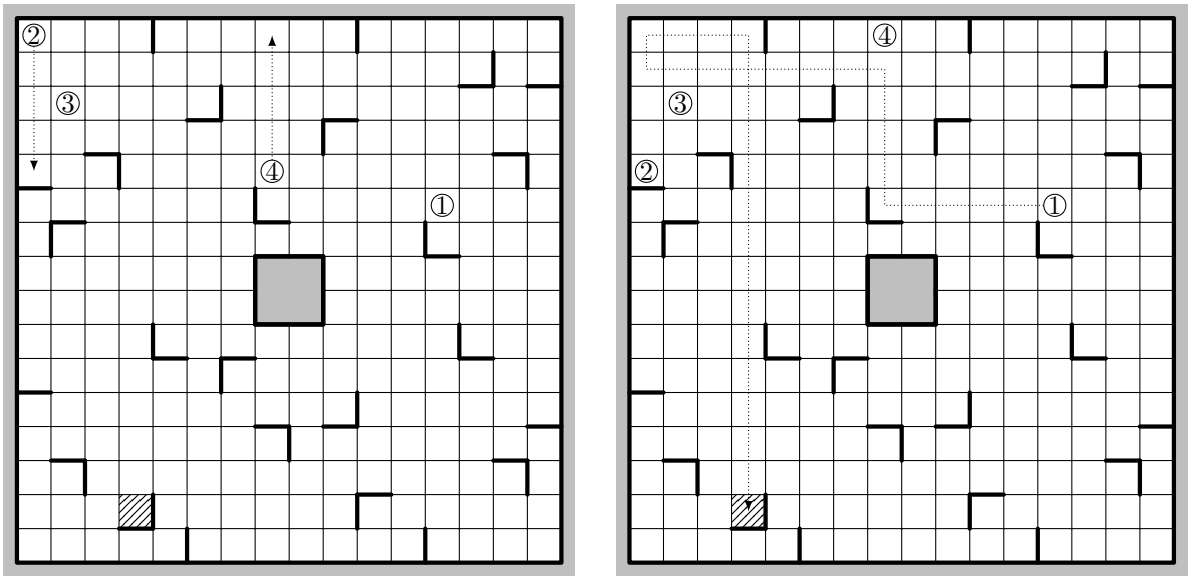

Figure 1 Le jeu des robots : le but est d'amener le robot 1 sur la case hachurée. À gauche : deux déplacements des robots 2 et 4 ; à droite : six déplacements du robot 1. Le jeu est résolu en 8 mouvements (solution optimale)

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

— `copy_vect : 'a vect -> 'a vect` telle que l'appel `copy_vect v` renvoie un nouveau tableau contenant les valeurs contenues dans `v` ;

— `make_vect : int -> 'a -> 'a vect` telle que l'appel `make_vect n x` renvoie un nouveau tableau de longueur `n` initialisé avec des éléments égaux à `x` ;

— `make_matrix : int -> int -> 'a -> 'a vect vect` telle que l'appel `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).

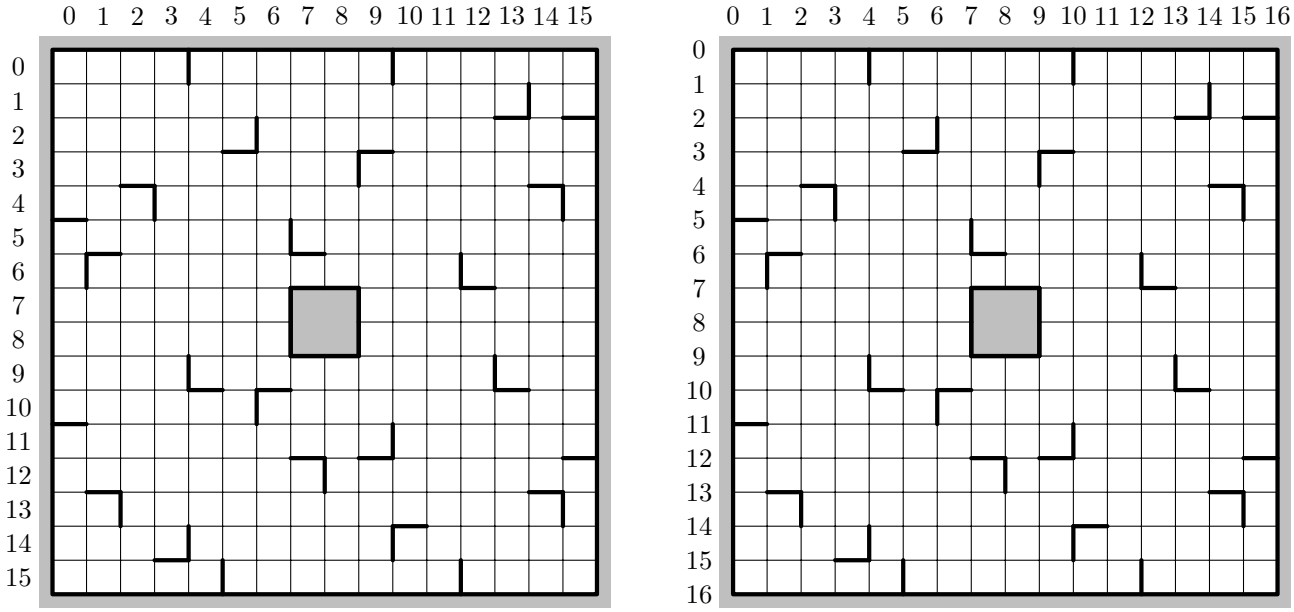

Figure 2 À gauche : numérotation des cases par ligne/colonne ; à droite : numérotation des lignes horizontales et verticales

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 obstacles_lignes = [| [|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 obstacles_colonnes = [| [|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.

Q 1. Écrire une fonction `dichotomie a t` de signature `int -> int vect -> 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, la fonction renvoie l'unique indice `i` tel que $\texttt{t.(i)} \leqslant \texttt{a} < \texttt{t.(i+1)}$. La fonction doit avoir une complexité logarithmique en la taille du tableau.

On considère un robot positionné en (a, b), avec $0 \leqslant a, b < N$. Il peut se déplacer dans les quatre directions cardinales ouest/est/nord/sud représentées sur la figure 4.

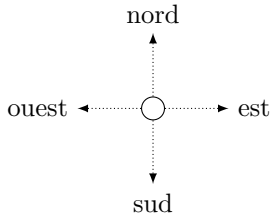

Figure 4 Déplacements cardinaux

Q 2. Écrire une fonction `deplacements_grille (a,b)` de signature `int * int -> (int * int) vect` fournissant les 4 cases atteintes par les déplacements en question, sous forme d'un tableau à 4 éléments

(ouest/est/nord/sud). Si le robot ne peux pas bouger dans une direction donnée (car il est contre un obstacle), on considérera que le résultat du déplacement dans cette direction est la case (a, b) elle-même. Les deux tableaux `obstacles_lignes` et `obstacles_colonnes` sont des variables globales.

Q 3. Écrire une fonction `matrice_deplacements ()`, de type `unit -> (int * int) vect vect vect` 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 $0 \leqslant a, b < N$. Donner la complexité de création de la matrice.

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 `mat_deplacements` que l'on considérera comme globale.

Q 4. Écrire une fonction `modif t (a,b) (c,d)` de signature

```
(int * int) vect -> 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'autres robots, et (c, d) la position d'un autre robot, alors la fonction 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.

Q 5. Déduire des questions précédentes une fonction `deplacements_robots (a,b) q` de signature

```
int * int -> (int * int) list -> (int * int) vect
```

donnant 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 `mat_deplacements` : on souhaite une copie modifiée de `mat_deplacements.(a).(b)`.

Q 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 Quelques fonctions utilitaires

II.A – Une fonction de tri

Q 7. Écrire une fonction `insertion x q` de signature `'a -> 'a list -> 'a list` prenant en entrée un élément `x` et une liste `q` triée dans l'ordre croissant, et renvoyant une liste triée dans l'ordre croissant, constituée des éléments de `q` et `x`.

Q 8. En déduire une fonction `tri_insertion q` de signature `'a list -> 'a list` permettant de trier une liste dans l'ordre croissant.

Q 9. Rappeler la complexité de ce tri dans le pire et le meilleur cas. Que peut-on dire de la complexité si dans la liste `q`, tous les éléments excepté peut-être un sont dans l'ordre croissant ?

II.B – Quelques fonctions sur les listes

Q 10. Écrire une fonction `mem1 x q` de signature `'a -> ('a * 'b) list -> bool` testant l'appartenance d'un couple dont le premier élément est `x` dans la liste `q`.

Q 11. Écrire une fonction `assoc x q` de signature `'a -> ('a * 'b) list -> 'b` renvoyant, s'il existe, l'élément `y` du premier couple `(x,y)` appartenant à la liste `q`.

II.C – Implantation d'une structure de file

On rappelle que l'on peut facilement implanter une structure de file à l'aide de deux listes : une des listes est utilisée pour rajouter des éléments, l'autre pour enlever des éléments. On définit ainsi le type

```
type 'a file = {mutable entree: 'a list; mutable sortie: 'a list};;
```

Lorsqu'on veut retirer un élément de la file alors que la deuxième liste est vide, on remplace celle-ci par la première, renversée.

On pourra utiliser les fonctions suivantes, qui permettent de manipuler une file ainsi définie :

`creer_file_vide : unit -> 'a file`	crée une file vide
`est_vide_file : 'a file -> bool`	teste si une file est vide
`enfiler : 'a file -> 'a -> unit`	ajoute un élément à une file
`defiler : 'a file -> 'a`	supprime l'élément en tête de file et le renvoie

On pourra supposer dans la suite que ces fonctions sont écrites de sorte que toute suite de p opérations `enfiler`/`defiler` à partir d'une file vide (ne produisant pas d'erreur) se fait en complexité $O(p)$.

III 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 $h_w : K \to [\![0, w-1]\!]$ 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.

III.A – Une famille de fonctions 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 $(a_i, b_i)_{0 \leqslant i < p}$ avec $0 \leqslant a_i, b_i < N-1$ est donnée par :

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

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.

Q 12. Écrire une fonction récursive `hachage_liste w q` de signature `int -> (int * int) list -> int` calculant la quantité précédente.

III.B – Tables de hachage de largeur fixée

Dans cette sous-section, on fixe une largeur de hachage w. Un bon choix pour w serait par exemple un nombre premier ni trop petit, ni trop grand, comme 997. Pour les listes, on considérerait alors la fonction de hachage h_{997} donnée en `Caml` par `hachage_liste 997`. On définit en toute généralité le type suivant :

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

III.B.1) Implantation de la structure de dictionnaire

Q 13. Écrire une fonction `creer_table h w` de signature `('a -> int) -> int -> ('a, 'b) table_hachage` telle que `creer_table h w` renvoie une nouvelle table de hachage vide de largeur `w` munie de la fonction de hachage `h`.

Q 14. Écrire une fonction `recherche t k` de signature `('a, 'b) table_hachage -> 'a -> bool` renvoyant un booléen indiquant si la clé k est présente dans la table t. On pourra utiliser les fonctions de la partie II.

Q 15. Écrire une fonction `element t k` de signature `('a, 'b) table_hachage -> 'a -> 'b` renvoyant l'élément e associé à la clé k dans la table t, si cette clé est bien présente dans la table.

Q 16. Écrire une fonction `ajout t k e` de signature `('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 clé est déjà présente.

Q 17. Écrire enfin une fonction `suppression t k` de signature `('a, 'b) table_hachage -> 'a -> unit` supprimant l'entrée de la clé k dans la table t. On n'effectuera aucun changement si la clé n'est pas présente.

III.B.2) Étude de la complexité de la recherche d'un élément

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 donc ici l'hypothèse de *hachage uniforme simple* : pour une clé donnée, la probabilité d'être hachée dans l'alvéole i est $1/w$, indépendante des autres clés. On note n le nombre de clés stockées dans la table et on appelle $\alpha = n/w$ le *facteur de remplissage* de la table. On suppose de plus, que le calcul du haché d'une clé se fait en temps constant.

Q 18. 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 $O(1+\alpha)$.

Q 19. On prend au hasard une clé présente dans la table ; toutes les clés sont équiprobables. Montrer qu'alors la recherche de la clé se fait en $O(1+\alpha)$, en moyenne sur toutes les clés présentes.

III.C – Tables de hachage dynamique

Les deux questions précédentes montrent 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 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). Par exemple, pour les listes de couples de $[\![0, N-1]\!]^2$, la fonction `hachage_liste` précédemment écrite fournit une telle famille. On définit en toute généralité le type suivant :

```
type ('a,'b) table_dyn = {
   hache: int -> 'a -> int;
   mutable taille: int;
   mutable donnees: ('a * 'b) list vect;
   mutable largeur: int};;
```

On notera trois différences par rapport au type précédent :

— la fonction `hache` possède un paramètre supplémentaire qui est la largeur de hachage, elle correspond maintenant à la famille de fonctions de hachage (h_w) ;

— on a rendu les champs `donnees` et `largeur` modifiables ;

— un champ `taille` (modifiable) est rajouté, il doit à tout moment contenir le nombre de clés présentes dans la table.

Q 20. Écrire une fonction `creer_table_dyn h` permettant de créer une table de hachage dynamique initialement vide, avec la famille de fonctions de hachage `h` et la largeur initiale 1.

On admet avoir écrit deux fonctions `recherche_dyn t k` et `element_dyn t k`, variantes des fonctions `recherche` et `element` précédentes, basées sur le même principe. On va maintenant développer une stratégie pour maintenir à tout moment un facteur de remplissage borné.

Q 21. Écrire une fonction `rearrange_dyn t w2` prenant en entrée une table de hachage dynamique et une nouvelle largeur de hachage `w2`, qui réarrange la table sur une largeur `w2`. En supposant que le calcul des valeurs de hachage se fasse en temps constant, la complexité doit être en $O(n+w+w_2)$ où n est le nombre de clés présentes dans la table (sa taille), w est l'ancienne largeur de la table, 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 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 $w' = 3w$.

Q 22. Écrire une fonction `ajout_dyn t k e` ajoutant 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.

Dans l'hypothèse que chaque ajout se fait en temps $O(1+\alpha)$, où α est le facteur de remplissage de la table, on peut montrer qu'une série de p ajouts dans une table initialement vide prend un temps $O(p)$.

On pourrait écrire de même une fonction de suppression dynamique, de sorte de maintenir un facteur de remplissage de la table borné, et qu'une série de p opérations licites d'insertion/suppression dans la table prenne un temps $O(p)$.

IV Résolution du jeu des robots

IV.A – 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 = {robot: int * int; autres_robots: (int * int) list};;
```

Pour chaque sommet, on impose que la liste `autres_robots` soit triée dans l'ordre croissant en suivant l'ordre lexicographique (l'ordre naturel pour les couples en `Caml`). Cet ordre est défini par $(a, b) \leqslant (a', b')$ si $a < a'$ ou si $a = a'$ et $b \leqslant b'$. Par exemple, `Caml` évalue l'expression `(2,3) < (3,0)` en `true`.

Les arcs dans le graphe (orienté) sont définis naturellement : un sommet s est relié à un sommet s' si on peut passer de s à s' par un mouvement licite d'un des robots.

Q 23. Avec p robots en tout sur un plateau de taille $N \times N$, quel est le nombre possible de sommets ? Donner le nombre exact pour $p = 4$ et $N = 16$.

Q 24. Écrire une fonction `sommets_accessibles s` de type `sommet -> sommet list` prenant en entrée un sommet et renvoyant la liste des sommets accessibles via `s` à partir du déplacement d'un des robots. S'il y a p robots en tout, la fonction renverra une liste de $4p$ sommets, certains sommets pouvant être égaux au sommet `s` : ils correspondent au mouvement d'un robot dans une direction où il est bloqué.

IV.B – Parcours en largeur : étude théorique

On se donne un graphe $G = (S, A)$ orienté, S dénote l'ensemble des sommets et A l'ensemble des arcs. On se donne un sommet $s_0 \in S$ du graphe et on considère l'algorithme 1 (parcours en largeur).

Entrées : un arbre $G = (S, A)$, orienté, un sommet de départ s_0
Sortie : un tableau de booléens, un tableau de prédecesseurs
$F \leftarrow$ creer_file_vide() ;
Enfiler s_0 dans F ;
$b_{s_0} \leftarrow$ **vrai** ;
$b_s \leftarrow$ **faux** pour tout $s \in S$; (* un tableau de booléens pour chaque sommet, tous faux *)
$\pi_s \leftarrow s$ pour tout $s \in S$; (* un tableau de prédecesseurs pour chaque sommet, initialement $\pi[s] = s$ *)
tant que F est non vide **faire**
    $s \leftarrow$ defiler(F) ;
    pour tout s' voisin de s tel que $b_{s'}$ est **faux faire**
        $b_{s'} \leftarrow$ **vrai** ; $\pi_{s'} \leftarrow s$; enfiler s' dans F ;
    fin pour
fin tant que
renvoyer b, π

Algorithme 1 Parcours en largeur

Q 25. Montrer que l'algorithme termine.

Q 26. Montrer que l'algorithme visite tous les sommets s du graphe pour lesquels il existe un chemin de s_0 à s.

Q 27. Pour un sommet s visité par l'algorithme (c'est-à-dire tel que b_s soit `Vrai` à la fin de l'algorithme), expliquer à partir de π comment retrouver un chemin de s_0 à s.

Q 28. Montrer que ce chemin est un plus court chemin de s_0 à s.

Q 29. On suppose que les voisins sont implantés par liste d'adjacence, la complexité est linéaire en le nombre de voisins pour les parcourir. Les opérations de file et les opérations sur les tableaux π et b s'effectuent en temps constant, donner la complexité de l'algorithme en fonction de $|S|$ et $|A|$.

Un parcours en largeur du graphe associé au jeu permet donc de trouver une solution qui nécessite le minimum de déplacements des robots. La difficulté dans l'implantation de cet algorithme réside dans le grand nombre de sommets du graphe. Pour pallier cette difficulté, on remplace les tableaux b et π par un dictionnaire implanté dans une table de hachage dynamique. Les clés et les éléments du dictionnaires sont tous les deux des sommets du graphe tels que l'élément associé à la clé s soit π_s. On remplace ainsi le test de $b_{s'}$ par l'existence de la clé s' dans le dictionnaire.

• • • FIN • • •