▶ **MPSI – Option Informatique**
Année 2002, Septième TP Caml
Vincent Simonet (`http://cristal.inria.fr/~simonet/`)

Labyrinthes

1 Représentation des labyrinthes

Les labyrinthes auxquels nous nous intéressons aujourd'hui sont défini en partant d'une grille de cases. Chaque case représente soit un *mur* soit un espace *vide* (i.e. l'espace où le joueur peut se déplacer).

Nous représentons en *Caml* un labyrinthe à n lignes et p colonnes par une matrice m de booléens de mêmes dimensions. Les lignes seront numérotées *de bas en haut* et de 0 à $n-1$, les colonnes de gauche à droite et de 0 à $p-1$. Si la case du labyrinthe située à l'intersection de la i^{e} ligne et la j^{e} colonne est un mur, $m.(i).(j)$ sera égal à `false`. S'il s'agit d'un chemin, $m.(i).(j)$ vaudra `true`.

▶ **Question 1** *Définissez la matrice qui correspond au labyrinthe suivant :*

Nous nous intéressons maintenant à l'affichage d'un labyrinthe dans la fenêtre graphique de *Caml*. Nous appelerons le *pas* du tracé la largeur et la hauteur (en pixels) commune des cases du damier représentant le labyrinthe.

▶ **Question 2** *Écrivez une fonction* draw_square *qui prend pour argument l'entier pas, un couple (i, j) désignant les coordonnées* ***dans la matrice*** *d'une case ainsi que la couleur à utiliser. Cette fonction tracera un rectangle plein représentant la case.*

value draw_square : $int \to (int \times int) \to color \to unit$

▶ **Question 3** *Déduisez-en une fonction* draw_maze *qui affiche un labyrinthe. Cette fonction prendra pour arguments l'entier pas et la matrice m.*

value draw_maze : $int \to bool\ vect\ vect \to unit$

On appelle un *chemin* d'un labyrinthe m une liste de couples d'entiers $[(i_0, j_0); \ldots; (i_{k-1}, j_{k-1})]$ telle que :
- Pour tout $0 \leq \ell \leq k-1$, (i_ℓ, j_ℓ) soit les coordonnées d'une case vide de m,
- Pour tout $1 \leq \ell \leq k-1$, les cases de coordonnées $(i_{\ell-1}, j_{\ell-1})$ et (i_ℓ, j_ℓ) soient contigues (i.e. $|i_{\ell-1} - i_\ell| + |j_{\ell-1} - j_\ell| = 1$).

▶ **Question 4** *Écrivez une fonction* draw_path *qui prend pour argument un chemin et l'affiche dans la fenêtre graphique.*

value draw_path : $int \to (int \times int)\ list \to unit$

2 Trouver la sortie !

Un problème intéressant, lorsqu'on s'intéresse à de tels labyrinthes, est de concevoir un algorithme qui permette à partir d'une position de départ (i_d, j_d), de rejoindre une arrivée (i_a, j_a). Nous ne ferons pas d'hypothèse particulière sur le placement de ces cases qui pourront se trouver à n'importe quel emplacement de la grille.
Le but de cette partie est d'implanter un algorithme de parcours *en profondeur* d'un labyrinthe, sans utiliser beaucoup de ressources. Le principe de l'algorithme est le suivant : lorsqu'on arrive sur une case, on commence par la marquer comme vue (pour être sûr de ne jamais s'y aventurer à nouveau) puis on se rend sur l'une de ses voisines. Si à une étape, tous nos voisins sont des murs ou ont déjà été vus, on revient sur nos pas jusqu'à trouver des cases non encore explorées.

2.1 Gestion d'une pile

Une pile est une suite finie de données de même type munie des deux opérations suivantes : *empiler* (*push* en anglais) qui consiste à ajouter une donnée au sommet de la pile et *dépiler* (*pop* en anglais) qui consiste à lire l'élément situé au sommet de la pile et à le retirer. L'idée importante à propos des piles est que l'on retire les éléments dans l'ordre inverse de leur insertion. Nous

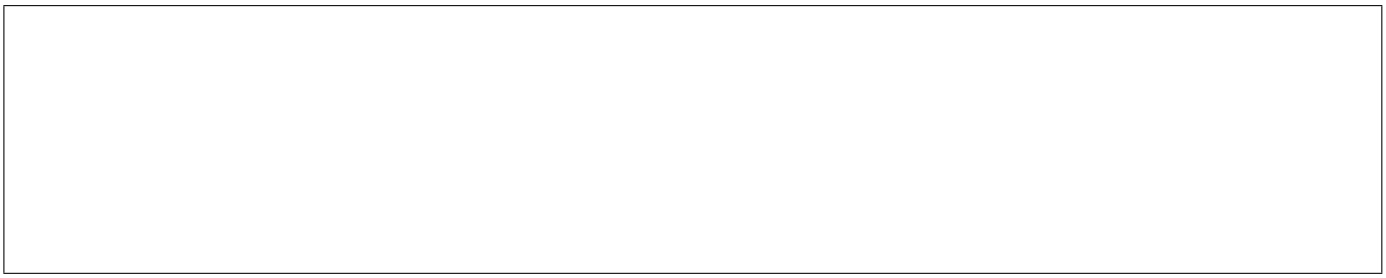

Fig. 1: Exemple de parcours

représentons une pile en *Caml* par une référence sur une liste.

▶ **Question 5 (Push)** *Écrivez une fonction* push *qui prend pour arguments une pile p et un élément x et qui empile x sur p.*

value push : *'a list ref → 'a → unit*

▶ **Question 6 (Pop)** *Écrivez une fonction* pop *prenant pour argument une pile p et dépile l'élément situé au sommet de p. Votre fonction lèvera une exception si la pile est vide.*

value pop : *'a list ref → 'a*

2.2 Petites fonctions intermédiaires

Dans la suite, on dit que deux cases sont voisines si et seulement si elles ont un côté commun. Une case qui n'est pas située sur le bord d'un labyrinthe admet donc exactement quatre voisines.

▶ **Question 7** *Écrivez une fonction* blocked *telle que* blocked m (i , j) *retourne un booléen indiquant si les quatres voisins de la case (i, j) sont des murs. (On pourra supposer que la case (i, j) n'est pas située sur un bord du labyrinthe.)*

value blocked : *bool vect vect → (int × int) → bool*

▶ **Question 8** *Écrivez une fonction* next *telle que* next laby (i , j) *retourne les coordonnées d'une case voisine de (i, j) qui ne soit pas un mur (On pourra supposer que la case (i, j) a effectivement un voisin qui n'est pas un mur et n'est pas située sur un bord du labyrinthe.)*

value next : *bool vect vect → (int × int) → (int × int)*

2.3 Parcours en profondeur

Nous allons écrire une fonction parcours qui prend pour arguments la matrice m, les coordonnées des points de départ (i_d, j_d) et d'arrivée (i_a, j_a) et trouve un chemin entre ces deux cases. Pour mener à bien notre parcours, nous aurons besoin de définir localement :

- Une référence *pos* sur un couple d'entiers indiquant la case où l'on se trouve.
- Une pile p – notre fil d'Ariane – contenant le chemin qui nous a mené de (i_d, j_d) à $!pos$.
- Une copie de la matrice m (vous pourrez utiliser la fonction copy_matrix pour la créer).

Lors du parcours, vous « murerez » les cases visitées en modifiant le booléen de la matrice correspondant. La fonction de parcours pourra alors procéder de la manière suivante : tant que la case courante n'est pas (i_a, j_a), on regarde si elle possède une voisine accessible. Si c'est le cas on y avance, sinon on revient sur ses pas. Un exemple de tel parcours est donné en bas de cette page.

▶ **Question 9** *Écrivez une fonction* walk *qui étant données deux cases (i_d, j_d) et (i_a, j_a), retourne un chemin joignant ces deux points (sous forme d'une liste). Votre fonction lèvera une exception si un tel chemin n'existe pas.*

value walk : *bool vect vect → (int × int) → (int × int) → (int × int) list*

▶ **Question 10** *Modifiez votre fonction* walk *de manière à ce qu'elle affiche le chemin sur la fenêtre graphique au fur et à mesure de sa « découverte ».*

▶ **Question 11** *Voyez-vous des simplifications possibles de l'algorithme si on fait l'une des hypothèses suivantes :* « le labyrinthe ne comporte pas de cycle » *ou* « le départ et l'arrivée sont situés sur le bord ».

3 Génération de labyrinthes

Nous pouvons maintenant nous intéresser à un problème plus délicat : celui de la génération « aléatoire » de labyrinthes. Il n'existe pas pour ce problème de solution « canonique ». Voici une possibilité permettant d'engendrer des grilles carrées de taille $2^k + 1$:

- On découpe la grille en quatre zones carrées de même taille.
- Dans trois des quatre murs de séparation (choisis aléatoirement), on creuse un trou à une position impaire.
- On recommence récursivement pour chacun des quatre carrés.

▶ **Question 12** *Écrivez une fonction* make_maze *prenant pour argument un entier k et retournant un labyrinthe carré de taille $2^k + 1$. Vous pourrez écrire une sous-fonction récursive prenant pour arguments trois entiers i_0, j_0 et m et dont la finalité est de remplir la portion carrée de la grille d'angle inférieur gauche (i_0, j_0) et de côté $2^m - 1$ cases.*

▶ **MPSI – Option Informatique**
Année 2002, Septième TP Caml
Vincent Simonet (`http://cristal.inria.fr/~simonet/`)

Labyrinthes

Un corrigé

▶ Question 2

```
let draw_square step (i, j) color =
  set_color color;
  fill_rect (i * step) (j * step) step step
;;
```

▶ Question 3

```
let draw_maze step m =
  let n = vect_length m in
  let p = vect_length m.(0) in
  for i = 0 to n - 1 do
    for j = 0 to p - 1 do
      draw_square step (i, j)
        (if m.(i).(j) then yellow else blue)
    done
  done
;;
```

▶ Question 4

```
let rec draw_path step = function
    [] -> ()
  | t :: q ->
     draw_square step t green;
     draw_path step q
;;
```

▶ Question 5

```
let push p x =
  p := x :: !p
;;
```

▶ Question 6

```
let pop p =
  match !p with
    [] -> invalid_arg "pop"
  | hd :: tl ->
    p := tl;
    hd
;;
```

▶ Question 7

```
let blocked m (i, j) =
  not (m.(i-1).(j) or m.(i+1).(j)
    or m.(i).(j-1) or m.(i).(j+1))
;;
```

▶ Question 8

```
let next m (i, j) =
  if m.(i-1).(j) then (i-1, j)
  else if m.(i+1).(j) then (i+1, j)
  else if m.(i).(j-1) then (i, j-1)
  else (i, j+1)
;;
```

▶ Question 9

```
let walk m start arrival =
  let pos = ref start in
  let stack = ref [] in
  let m' = copy_matrix m in
  m'.(fst !pos).(snd !pos) <- false;
  while !pos <> arrival do
    if not (blocked m' !pos) then begin
      push stack !pos;
      pos := next m' !pos;
      m'.(fst !pos).(snd !pos) <- false
    end
    else pos := pop stack
  done;
  rev (arrival :: !stack)
;;
```

▶ Question 10

```
let wait () =
  let _ = sys__system_command "sleep 0.3" in
  ()
;;

let drawing_walk step m start arrival =
  let pos = ref start in
  let stack = ref [] in
  let m' = copy_matrix m in
  clear_graph ();
  draw_maze step m';
  m'.(fst !pos).(snd !pos) <- false;
  draw_square step !pos green;
  while !pos <> arrival do
    wait ();
    if not (blocked m' !pos) then begin
      flush stdout;
      push stack !pos;
      pos := next m' !pos;
      draw_square step !pos green;
      m'.(fst !pos).(snd !pos) <- false
    end
    else begin
      draw_square step !pos yellow;
      pos := pop stack
    end
  done;
  rev (arrival :: !stack)
;;
```

▶ Question 12

```
let rec prefix ** x = function
    0 -> 1
  | n -> x * (x ** (n - 1))
;;

let make_laby k =

  let n = 2 ** k + 1 in
  let laby = make_matrix n n true in

  let rec def (i0, j0) m =
    if m >= 2 then begin
      let m2 = 2 ** m - 1 in

      (* On trace la croix centrale *)
      for i = i0 to i0 + m2 - 1 do
        laby.(i).(j0 + m2/2) <- false
      done;
      for j = j0 to j0 + m2 - 1 do
        laby.(i0 + m2/2).(j) <- false
      done;

      (* On fait trois trous dans cette croix *)
      let a = random__int 4 in
      let r () = (random__int (m2/4 + 1)) * 2 in
      if a <> 0 then laby.(i0 + m2/2).(j0 + r()) <- true;
      if a <> 1 then laby.(i0 + m2/2).(j0 + m2/2 + 1 + r()) <- true;
      if a <> 2 then laby.(i0 + r()).(j0 + m2/2) <- true;
      if a <> 3 then laby.(i0 + m2/2 + 1 + r()).(j0 + m2/2) <- true;

      (* Appels recursifs dans les sous-carres *)
      def (i0, j0) (m - 1);
      def (i0 + m2/2 + 1, j0) (m - 1);
      def (i0, j0 + m2/2 + 1) (m - 1);
      def (i0 + m2/2 + 1, j0 + m2/2 + 1) (m - 1);
    end
  in

  def (1,1) k;

  (* Trace du bord *)
  for x = 0 to n - 1 do
    laby.(x).(0) <- false; laby.(x).(n-1) <- false;
    laby.(0).(x) <- false; laby.(n-1).(x) <- false
  done;

  laby
;;
```