TP 2 : Piles et Labyrinthe

1 Quelques manipulations rapides de piles

Récupérer l'implémentation de la structure de pile sur mon site web. Celle-ci est différente de celle du cours, et basée sur le type `Stack` de Ocaml, qui est une structure abstraite de pile déja implémentée. Les fonctions sont les suivantes :

```
creer_pile : unit -> 'a Stack.t
pile_vide : 'a Stack.t -> bool
sommet : 'a Stack.t -> 'a
depiler : 'a Stack.t -> 'a
empiler : 'a Stack.t -> 'a -> unit
```

(Le type `Stack.t` est le type pile de Ocaml). On remarquera que les types sont les mêmes que pour les piles du cours, à l'exception de la création qui ne prend aucun élément en paramètre (il s'agit donc de piles non bornées).

Exercice 1. Écrire une fonction `pile_entier n` de type `int -> int pile` qui crée une pile contenant les entiers de 0 à $n-1$, $n-1$ étant au sommet.

```
# let p = pile_entier 5 ;;
val p : int Stack.t = <abstr>
# sommet p ;;
- : int = 4
```

Exercice 2. Écrire une fonction `afficher p`, de type `int pile -> unit` qui affiche le contenu d'une pile d'entiers à l'écran en commençant par le sommet, la pile devant rester inchangée en fin de fonction. Indication : Il est nécessaire d'utiliser une deuxième pile (temporaire), dans laquelle vous « viderez » la première en affichant les éléments au passage, puis vous remettrez les éléments dedans. `print_int` permet d'afficher un entier, `print_string " "` vous permettra de mettre des espaces).

```
# let p=pile_entier 5 in afficher p ; sommet p ;;
4 3 2 1 0 - : int = 4
```

Exercice 3. Écrire une fonction `sommet_au_fond p` de type `'a pile -> unit` qui place l'élément en haut de la pile `p` (supposée non vide) tout au fond. On fera usage d'une autre pile.

```
# let p=pile_entier 5 in sommet_au_fond p ; afficher p ;;
3 2 1 0 4 - : unit = ()
```

Remarque : si on a besoin d'effectuer des opérations comme `sommet_au_fond` sur une pile, c'est que la structure de données utilisée est mal choisie...

2 Réalisation d'un labyrinthe parfait

Sur la grille discrète $[\![0, n-1]\!] \times [\![0, n-1]\!]$, on considère les n^2 nœuds $\{(i, j) \mid 0 \leq i, j \leq n-1\}$. Deux nœuds de la grille sont voisins si ils sont à distance exactement 1. Un chemin simple entre deux nœuds c et c' est une suite de nœuds tous distincts et successivement voisins $c = c_1, \ldots, c_k, \ldots, c_\ell = c'$, d'extrémités c et c'.

Un labyrinthe parfait est le résultat du tracé de segments entre nœuds voisins, tel que, pour toute paire de nœuds de la grille, il existe un et un seul chemin simple entre ces nœuds.

Voir les figures 1 et 2 pour des exemples de labyrinthes parfaits de dimension 5×5 et 18×18. Pour le second, on a marqué l'unique chemin simple allant du coin inférieur gauche (0, 0) au coin supérieur droit (17, 17).

À la différence des livres de jeux pour enfants, cheminer dans ces labyrinthes consiste à suivre les « lignes-chemins » plutôt qu'à se déplacer entre des « lignes-parois ».

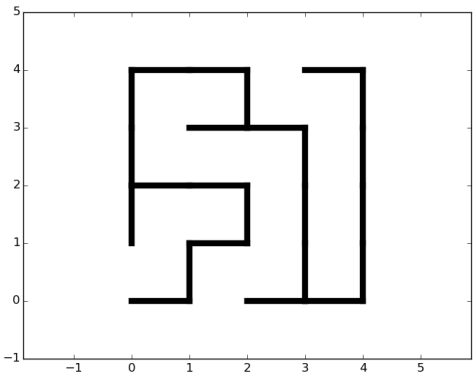

FIGURE 1: Un labyrinthe parfait de taille 5×5.

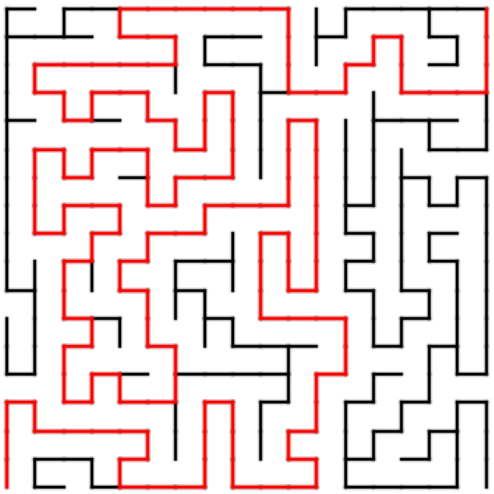

FIGURE 2: Un labyrinthe parfait de taille 18×18, l'unique chemin de la case $(0, 0)$ à la case $(17, 17)$ marqué.

La construction d'un tel labyrinthe consiste à visiter les nœuds de la grille à l'aide d'une structure de pile. On se donne :

— une grille `tab_visites` de dimension $n \times n$ de booléens pour identifier les nœuds déjà visités. La structure est une matrice (tableau de tableaux) en Caml dont on rappelle l'utilisation :

`Array.make_matrix n m x`	crée une matrice à n lignes et m colonnes contenant uniquement des `x`.
`Array.length m`	renvoie le nombre de lignes de `m`.
`m.(i)`	si `m` est une matrice, `m.(i)` est le tableau constituant la i-ème ligne de `m` (les lignes sont indexées à partir de 0).
`m.(i).(j)`	le coefficient en case (i, j) de `m` (lignes et colonnes sont indexées à partir de 0).
`m.(i).(j) <- x`	permet de stocker `x` en case (i, j) de `m`.

— une pile `a_traiter`, initialement vide, contenant les nœuds à traiter sous forme de couples.

On part d'un premier nœud, qu'on supposera être $(0, 0)$, que l'on empile sur `a_traiter` et que l'on marque dans `tab_visites` (on positionne donc `tab_visites.(0).(0)` à `true`, les autres cases contenant `false`). L'algorithme utilisé ensuite est l'algorithme 1.

Algorithme 1 : Construction d'un labyrinthe parfait

tant que *la pile* `a_traiter` *est non vide* **faire**
  Dépiler le « nœud-sommet » c de la pile ;
  Identifier les nœuds voisins de c non encore visités;
  si *il y a au moins un nœud voisin de* c *non encore visité* **alors**
    en choisir un aléatoirement : s;
    tracer le chemin (c, s);
    empiler c sur `a_traiter`;
    empiler s sur `a_traiter`;
    marquer s dans `tab_visites`

2.1 Le programme principal

Le script est récupérable dans l'annexe, voici des explications.
On commence par fixer l'entier n, ainsi que la grille de booléens :

```
let n=50 (* ou une autre valeur *) ;;
tab_visites = Array.make_matrix n n false ;;
tab_visites.(0).(0) <- true ;;
```

Ensuite, on ouvre un graphique, et on se donne un fonction `trace c1 c2` prenant en entrée deux couples d'entiers et traçant le segment d'extrémités `c1` et `c2` (en noir).

```
#load "graphics.cma" ;;
open Graphics ;;

let s=string_of_int (10*n+10) in
let t= " "^s^"x"^s in
open_graph t ;;

let trace c1 c2=
  set_color black ;
  let x1, y1 = c1 and x2, y2 = c2 in
  moveto (10+10*x1) (10+10*y1) ;
  lineto (10+10*x2) (10+10*y2)
;;
```

2.2 Des fonctions auxiliaires

Les fonctions suivantes aident à mettre en oeuvre l'algorithme. Attention, une difficulté est de ne pas sortir de la grille. On rappelle que la variable `n` donnant la taille de la grille est une variable globale, de même que la grille de booléens `tab_visites`.

Question 1. Écrire une fonction `est_dans_grille (x,y)` prenant en entrée un couple d'entiers et renvoyant un booléen indiquant si celui-ci est dans la grille (coordonnées dans $[\![0, n-1]\!]$).

Question 2. Écrire une fonction `visiter (x,y)` qui prend en entrée un couple d'entiers, **supposés représenter une case de la grille.** La fonction positionne la case de `tab_visites` indexée par `(x,y)`.

Question 3. Écrire une fonction `a_ete_visitee (x,y)` prenant en entrée un couple d'entiers et retournant un booléen indiquant si la case c a déja été visitée. **On considérera une case hors de la grille comme déja visitée, et on ne fera pas d'hypothèses sur les entiers x et y, qui peuvent être négatifs ou supérieurs à n.**

```
# a_ete_visitee (-1,0) ;;
- : bool = true
# a_ete_visitee (n-1,n-1) ;;
- : bool = false
```

Question 4. En déduire une fonction `voisins_non_visites (x,y)` prenant en entrée un couple d'entiers (qu'on suppose être une case de la grille, première et seconde composante sont entre 0 et $n-1$), et renvoyant une (petite) pile contenant les voisins de c non encore visités (ce sont *forcément* des cases de la grille) [1]

```
# let p = voisins_non_visites (0,0) ;;
val p : (int * int) Stack.t = <abstr>
# depiler p
- : int * int = (0, 1)
# depiler p ;;
- : int * int = (1, 0)
# pile_vide p ;;
- : bool = true
```

Question 5. Écrire une fonction `k_eme p k` prenant en entrée une pile `p` et un entier $k \geq 0$ et retournant le k-ème élément de la pile à partir du sommet (indexés à partir de 0). On suppose que la pile a au moins $k+1$ éléments, et on s'autorise à « détruire » la pile.

```
# let p=pile_entier 5 in k_eme p 1 ;;
- : int = 3
```

1. On verra plus tard la structure de liste chaînée... qui évite quand même de s'embêter à manipuler des piles tout le temps !

Question 6. La fonction `Random.int: int -> int` prend en entrée un entier $k > 0$ et retourne aléatoirement un entier de l'intervalle $[\![0, k-1]\!]$. Écrire une fonction `tirage_sort p` prenant en entrée une pile `p` supposée non vide, et renvoyant un élément aléatoire de cette pile. (On utilisera ici la fonction `Stack.size` qui donne le nombre d'éléments présents dans la pile passée en paramètre).

2.3 Poursuite du programme principal

Question 7. Écrire un script permettant de tracer un labyrinthe parfait (il s'agit essentiellement d'écrire l'algorithme 1 en Caml).

Question 8. Estimer la complexité du script en fonction de n.

2.4 Chemins dans le labyrinthe

Cette section est plus délicate, et théorique.

Question 9. Montrer qu'une fois le labyrinthe créé, il existe bien un chemin entre deux couples de la grille (raisonner par l'absurde).

Question 10. Montrer de plus qu'un chemin simple entre deux points de la grille est unique.

Question 11. On suppose que dans l'implémentation de l'algorithme, on a en plus utilisé une matrice *des prédecesseurs* permettant pour chaque nœud s (excepté $(0,0)$) de connaître le nœud c ayant permis la découverte. Expliquer sans coder comment utiliser cette matrice pour obtenir en plus la liste (sous forme de pile, par exemple) des sommets appartenant à l'unique chemin du nœud $(0,0)$ au nœud $(n-1, n-1)$.

Question 12. `set_color red` permet de fixer la couleur du tracé en rouge. Réécrire une version de la fonction `trace` pour tracer en rouge ce chemin.

Question 13. Reprendre l'algorithme de tracé du labyrinthe pour calculer en même temps la matrice des prédécesseurs, puis tracer en rouge l'unique chemin simple de la case $(0,0)$ à la case $(n-1, n-1)$.

3 S'il vous reste du temps

Question 14. On considère le type date suivant :

```
type date = {jour: int ; mois: int ; annee: int} ;;
```

On considère que le jour est un entier entre 1 et 31, le mois un entier entre 1 et 12, et l'année un entier positif.

1. Écrire une fonction permettant de tester si c'est un jour valide. On pourra prendre en compte les années bissextiles [2]

```
# est_correcte {jour = 25; mois = 12; annee=2000} ;;
- : bool = true
# est_correcte {jour = 29; mois = 2; annee=2000} ;;
- : bool = true
# est_correcte {jour = 29; mois = 2; annee=1900} ;;
- : bool = false
# est_correcte {jour = 31; mois = 3; annee=1900} ;;
- : bool = true
```

2. Écrire une fonction permettant de calculer la date du lendemain.

```
# lendemain {jour = 31; mois = 12; annee=2017} ;;
- : date = {jour = 1; mois = 1; annee = 2018}
# lendemain {jour = 29; mois = 2; annee=2000} ;;
- : date = {jour = 1; mois = 3; annee = 2000}
# lendemain {jour = 21; mois = 2; annee=2018} ;;
- : date = {jour = 22; mois = 2; annee = 2018}
```

2. Une année est bissextile si elle est divisible par 4, sauf si elle est divisible par 100 auquel cas elle doit aussi être divisible par 400 !