TP9 : Labyrinthes

Paul Melotti (paul.melotti@ens.fr)*

6 et 13 mars 2014

L'objectif de ce TP est de découvrir comment les notions de parcours en profondeur et en largeur que vous avez vues peuvent être utilisées dans le cas des labyrinthes, et d'exploiter des petits lutins.

Un labyrinthe sera représenté de la façon suivante : c'est une matrice d'entiers, qui valent 0 sur les zones dégagées et 1 sur les murs. **On suppose que tous les coefficients du bord de la matrice sont des 1.** Une seule case de la matrice contient l'entier 2, c'est la sortie. On a le droit de se déplacer dans les quatre directions, mais pas en diagonale. Exemple :

```
let laby = [|
  [|1;1;1;1;1;1;1;1;1;1|];
  [|1;0;1;0;0;1;2;1;1|];
  [|1;0;1;0;1;0;0;0;1|];
  [|1;0;0;0;0;0;1;0;1|];
  [|1;1;1;1;1;1;1;1;1|]
|];;
```

Question 1. Écrire une fonction `affiche : int vect vect -> unit` qui affiche un labyrinthe : on représentera les murs par des `#`, les zones dégagées par des espaces vides et la sortie par un `S`. Exemple :

```
affiche laby;;
#########
# #  #S##
# # #   #
#     # #
#########
```

1 Parcours en profondeur

On lance un petit lutin dans le labyrinthe à la case i_0, j_0 et on le charge de trouver la sortie. On lui donne un petit pot de peinture, comme ça il peut marquer les cases déjà rencontrées. Naturellement, notre lutin va effectuer un parcours en profondeur :

```
PP(case) =
   SI c'est la sortie, c'est gagné.
   SI la case n'est pas marquée,
      marquer la case,
      pour toutes les cases voisines,
         PP(voisine).
```

Question 2. Pourquoi faut-il marquer les cases déjà rencontrées ?

En Caml, on marquera les cases rencontrées en mettant un 3 dans la matrice. Comme on ne veut pas modifier notre labyrinthe initial, on a besoin de la fonction suivante :

Question 3. Écrire une fonction `copie_matrice : int vect vect -> int vect vect` qui renvoie une copie de la matrice donnée en argument.

*N'HÉSITEZ à m'écrire pour toute question sur l'informatique, les TIPE, les concours... Je reste à votre disposition.

Question 4. Écrire une fonction `estsoluble : int vect vect -> int -> int -> bool` qui prend en entrée un labyrinthe et la case d'où on part et dit si on peut atteindre la sortie. Utilisez pour cela un parcours en profondeur.

On a maintenant un petit lutin qui sait nous dire si on peut sortir du labyrinthe. Le problème, c'est qu'il ne nous dit pas quel chemin il faut prendre. Il serait intéressant d'écrire sur chaque case visitée le chemin allant du départ à cette case, sous la forme d'une `(int * int) list`.

Question 5. Reprendre la fonction de la question 4 pour qu'elle retienne aussi les chemins. Elle renverra un `bool * (int * int) list` : le booléen dit si on peut sortir, et la liste nous donne un chemin pour sortir (et ce que vous voulez lorsqu'on ne peut pas sortir).

2 Parcours en largeur

On voudrait maintenant connaître le plus court chemin de l'entrée à la sortie. Pour cela, au lieu de s'enfoncer le plus possible dans les chemins comme avec un parcours en profondeur, on va plutôt regarder d'abord les cases qui sont à distance 1, puis celles qui sont à distance 2, etc. C'est l'idée du parcours en largeur. Pour cela, on va utiliser une file d'attente (structure FIFO) pour stocker les cases à explorer (et on a donc besoin d'un lutin plus intelligent).

Question 6. Écrire rapidement des fonctions de manipulation de files d'attente : enfilage, défilage, et test de file vide.

On utilise la fonction d'exploration suivante : la file d'attente `f` est initialisée à `[(i0,j0)]` et la case (i_0, j_0) est initialement marquée, et on lance le parcours en largeur :

```
TANT QUE f non vide,
   Défiler x de f,
   Pour tout voisin y de x,
      SI y non marqué
         enfiler y,
         .........,
         marquer y.
FIN TANT QUE
```

La partie laissée en petits points correspond à une certaine action que vous devrez trouver vous-même.

Question 7. Se convaincre que cet algorithme (le parcours en largeur) explore bien les cases par distance à l'origine croissante.

Conseil : essayer sur un exemple.

Question 8. Écrire une fonction `voisins : int -> int -> int vect vect -> (int * int) list` qui étant donné `i`, `j` et un labyrinthe renvoie la liste des voisins de la case (i, j) qui ne sont pas marqués et ne sont pas des murs.

Question 9. Écrire une fonction qui fait comme à la question 5, mais cette fois-ci renvoie un chemin de longueur minimale lorsque le labyrinthe est résoluble.

Ce qu'il faut retenir de cette étude : parcours en profondeur = pile, parcours en largeur = file. Le parcours en largeur est utile dans des problèmes de plus court chemin.

Question 10. - Facultative Essayez de générer des labyrinthes aléatoires : chaque case est un mur avec probabilité p et un couloir avec probabilité $1-p$. Les cases au bord doivent rester des murs, et vous pouvez mettre une ou des sortie-s un peu où vous voulez. Quelques fonctions pour faire de l'aléatoire : `random__int n` renvoie un entier de $[0;n[$, `random__float f` renvoie un flottant de $[0;f[$.

On peut se poser de nombreuses questions sur les labyrinthes aléatoires, par exemple : quelle est la probabilité de relier le haut du labyrinthe au bas ? Évaluez cette probabilité en faisant des essais sur de grands labyrinthes (de taille 100*100), pour différents p.

Vous venez d'étudier quelques questions relatives à la *percolation par sites* sur le réseau $\mathbb{Z}^2$.

3 Un dernier pour la route

Voici quelques compléments sur des TP passés. Libre à vous de faire ce qui vous inspire.

3.1 Listes d'adjacence et tri topologique

On a appris à représenter un graphe orienté G par sa matrice d'adjacence M, c'est-à-dire une matrice telle que $M_{ij} = 1$ si (i,j) est une arête, 0 sinon.

On peut utiliser une autre représentation : les listes d'adjacence. Pour chaque sommet i du graphe, on fait la liste l_i des sommets j tels que (i,j) est une arête. Puis on met toutes ces listes d'adjacence dans un tableau $[|l_0;...;l_{n-1}|]$. Le graphe est alors représenté par un `int list vect`.

Question 11. Écrire des fonctions qui permettent de passer de l'une à l'autre de ces deux représentations d'un graphe.

Soit G un graphe orienté acyclique connexe, de sommets $[0, n-1]$. Le tri topolgique de G consiste à renuméroter les sommets de manière à ce que toute arête (i,j) vérifie $i < j$. On propose l'algorithme suivant :

On crée une liste initialement vide, et on effectue un parcours en profondeur du graphe. Lorsqu'on a complètement fini l'exploration d'un sommet, on l'ajoute en tête de liste.

Question 12. Justifier que cet algorithme convient.

Question 13. Le programmer avec des listes d'adjacence.

Ce tri est utilisé lorsqu'on veut organiser un planning de tâches à réaliser qui sont liées par des contraintes du type "A doit être fait avant B". Par exemple, modéliser et résoudre le problème suivant : on veut s'habiller correctement le matin avec les vêtements : cravate, pantalon, ceinture, chaussettes, montre, chaussures, chemise, veste, caleçon.

3.2 Écriture sous forme infixe

Par rapport aux écritures préfixe et suffixe, l'écriture infixe est plus naturelle pour l'utilisateur. Par contre elle est plus compliquée à manipuler informatiquement à cause des problèmes de parenthésages.

Comme au TP7, partie 2, on va se limiter à des symboles fonctionnels d'arité 0, 1 ou 2. On rappelle le type utilisé pour manier des expressions :

```
type expr = C of string | U of string*expr | B of string*expr*expr.
```

L'écriture bien parenthésée d'une expression t, ou écriture infixe de t, est définie inductivement par :

$$\mathrm{Inf}(t) = \begin{cases} t & \text{si } t \text{ est constant} \\ f\ \mathrm{Inf}(t_1) & \text{si } t = f(t_1) \\ \left(\ \mathrm{Inf}(t_1)\ f\ \mathrm{Inf}(t_2)\ \right) & \text{si } t = f(t_1, t_2) \end{cases}$$

Question 14. Écrire une fonction `ecritinf : expr -> unit` qui imprime l'écriture infixe d'une expression.

Question 15. Écrire une fonction `creeliste : expr -> (int * string) list` qui prend une expression et renvoie la liste de couples associée (c'est la liste des couples arité, nom du symbole, écrits dans l'ordre de l'écriture infixe ; on mettra -1 dans la case arité pour une parenthèse).

Question 16. Écrire une fonction `arbre_de_liste : (int * string) list -> expr` qui reconstruit une expression à partir de sa liste de couples.

Question 17. On souhaite transformer une chaîne de caractères du style `"( ( exp x + y ) * z )"` en son arbre syntaxique. On suppose que le seul symbole d'arité 1 est `exp`, que les seuls symboles d'arité 2 sont `+` et `*` (les autres symboles sont donc d'arité 0), et que les symboles sont séparés par des espaces. Écrire une fonction qui transforme ce genre de chaîne en liste de couples, puis en arbre syntaxique.

3.3 Autre chose

Combien y a-t-il de positions finales nulles (*i.e.* sans gagnant) au Tic-Tac-Toe 3*3*3 ?