Algorithme de Lee : Résolution d'un labyrinthe

Ensimag 1A - Préparation au Projet C

Présentation

Le but de ce problème est de trouver le chemin le plus court entre 2 points dans un labyrinthe. On définit un labyrinthe comme étant un tableau de dimensions $h \times w$, dans lequel certaines cases sont occupées. On peut passer d'une case à une des 4 cases voisines ssi cette dernière n'est pas occupée. Le labyrinthe définit aussi un point de départ et un point d'arrivée. Les labyrinthes à traiter sont décrits dans un fichier texte. Un exemple d'un tel labyrinthe est donné figure 1.

```
+------------------------------+        +------------------------------+
| # ## #    # #          #   A #|        | # ## #    # #          #  .A #|
| # #  # #   # # #   # # ## ## |        | # #  # #   # # #   # # ##.## |
|##   ##    #    ##       #    |        |##   ##    #    ##       #..  |
|   ##  D# # #     ###     #   |        |   ##  D# # #     ###     #.  |
|  # # #  #   ##   # #   #     |        |  # # #. #   ##   # #   #...  |
| #      #    #   ##       ####|        | #    ..# ...#...##......####|
| ###   ### #   #         #  # |        | ###  .###.#...#....     #  # |
|   ##       #   ####  #      #|        |   ## ..... #   ####  #      #|
|#    #    # #  ## ##          |        |#    #    # #  ## ##          |
|   #    #   # ## #   ##  #  # |        |   #    #   # ## #   ##  #  # |
+------------------------------+        +------------------------------+
```

FIGURE 1 – Exemple d'un labyrinthe au format texte et sa solution

Les objectifs de ce sujet, du point de vue du langage C, sont les suivants :
— Précédence des opérateurs C ;
— Manipulation des pointeurs ;
— Manipulation et analyse de fichiers.

Algorithme employé

L'algorithme employé ici pour résoudre ce problème procède de manière itérative, en recherchant à chaque itération les cases atteignables en $n + 1$ pas à partir de celles atteignables en n pas.

Soit C l'ensemble des cases non occupées. Pour $c \in C$, (c, i) est dans l'ensemble Att ssi la case c est atteignable en i pas depuis la case de départ. D est la case de départ, et A la case d'arrivée. Enfin, $V(c)$ désigne les cases voisines de la case c.

Avec ces notations, l'algorithme est le suivant :
$Att = (D, 0)$
$n \leftarrow 0$
while $((A, n) \notin Att)$
  for all c such that $(c, n) \in Att$
    for all $c' \in V(c) \cap C$
      if $(c', i) \notin Att \; \forall i \leq n + 1$
        $Att = Att \cup (c', n + 1)$
  $n \leftarrow n + 1$

Dans l'implémentation, on pourra utiliser un "tableau" dont la valeur de chaque case représente son état (case occupée, case de départ, case d'arrivée, case pas encore atteinte, ou case atteinte en n pas).

L'algorithme ci-dessus nécessite de faire le parcours de tout le tableau à chaque itération. Cela est loin d'être le plus efficace, mais a le mérite d'être simple.

Par ailleurs, il est à noter que cet algorithme ne fait pas le tracé du chemin. Pour ce faire, il faut partir de l'arrivée et remonter jusqu'à la case de départ en trouvant à chaque itération une case atteignable avec un nombre de pas égal au nombre de pas de la case actuelle moins un. Plusieurs chemins optimaux peuvent exister : dans ce cas nous vous demandons de choisir simplement l'un de ces chemins.

Travail demandé

Nous vous proposons un découpage en trois modules :

— `maze.h`, qui définit la structure de données représentant un labyrinthe. Vous choisirez comment stocker le tableau de valeurs, qui peut être par exemple un tableau unidimensionel de $h \times w$ cases, ou bien un tableau de h pointeurs vers des tableau de w cases (voir `maze.h`). Le tableau unidimensionel est plus simple à allouer et libérer, et potentiellement plus efficace, tandis que le tableau 2D simplifie l'identification des voisins d'une case donnée.

— `lee.h`, dans lequel l'algorithme de lee est implémenté.

— `mazeio.h` pour gérer les entrées–sorties. Une des difficultés du travail demandé est d'arriver à construire le tableau correspondant au labyrinthe à partir d'un fichier (`create_maze_from_file`). En particulier, allouer un tableau de taille correcte nécessite deux parcours du fichier. A la lecture du fichier, vous vérifierez qu'il définit bien un labyrinthe valide (rectangulaire, un seul départ, une seule arrivée, bordure correcte, tous les caractères sont valides,...). Vous devrez utiliser les fonctions de la libc pour la lecture des fichiers, en particulier `fopen` et `fgetc`. La fonction `print_maze` peut être appelée sur un labyrinthe avant ou après l'appel à `solve_maze` et/ou à `build_path`, ou même dans le corps de ces fonctions pour vous aider à débugger : selon les cas, on pourra vouloir afficher les nombres de pas pour atteindre les cases (debug), ou simplement afficher la structure du labyrinthe et éventuellement le chemin optimal trouvé. A cette fin, on lui transmet le booléen `values`.

Trois exemples de labyrinthes sont fournis.

Extension possible

Comme mentionné, l'algorithme proposé doit parcourir tout le tableau à chaque itération, afin de trouver les cases atteignables en n pas pour identifier celles atteignables en $n + 1$ pas. Nous vous proposons d'optimiser cet algorithme : il s'agit de construire à chaque iteration la liste des cases atteignables en $n + 1$ pas, puis d'utiliser cette liste à l'itération suivante. Cette optimisation sera implémentée dans `solve_maze`, et ne nécessite aucun changement des interfaces proposées (*.h). Pensez bien à libérer toute mémoire allouée (valgrind power !).