TP C# 10 - Labyrinthe

1 Rendu

Pour ce TP nous attendons les fichiers suivants :

```
rendu-TPCSXX-login_x.zip
|-- rendu-TPCSXX-login_x/
    |-- AUTHORS
    |-- Bonus
        |-- Maze_bonus.cs
        |-- Cell_bonus.cs
    |-- labyrinthe.sln
    |-- labyrinthe
        |-- Tout sauf bin/ et obj/
```

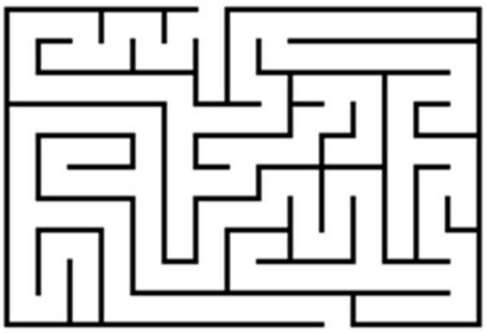
2 Introduction

2.1 Objectifs du TP

Aujourd'hui nous allons jouer avec les classes et les labyrinthes ! Durant ce TP vous allez devoir implémenter un algorithme de génération de labyrinthes ainsi qu'un algorithme de résolution. Pour commencer nous allons implémenter 2 classes permettant de représenter des labyrinthes. Une fois cela fait, nous passerons aux choses sérieuses avec la génération de labyrinthes et leur résolution. Avant de commencer, voici une petite introduction sur les differents labyrinthes.

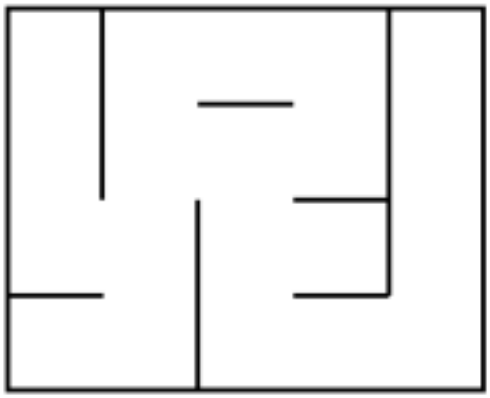
2.2 Les labyrinthes

Un labyrinthe est une zone contenant des murs et possédant une ou plusieurs entrées et une ou plusieurs sorties. Dans notre cas, le labyrinthe sera composé de cases séparées par des murs et possédera une entrée et une sortie. Pour le représenter, on utilisera un tableau à deux dimensions composé de cellules. Les labyrinthes existent sous deux formes, parfaits et imparfaits. Le labyrinthe parfait :



Il ne possède aucun ilots isolés, toutes les cases peuvent être atteintes et il peut être parcourus grâce un simple parcours profondeur.

Le labyrinthe imparfait :



Il possède des ilots isolés qui rendent sa résolution plus compliquée. Pour ce TP nous implémenterons un labyrinthe parfait.

3 Implémenter le labyrinthe

Le labyrinthe sera implémenté sous la forme de deux classes : *Cell* et *Maze*. La classe *Maze* représentera le labyrinthe et contiendra un tableau à deux dimensions. La classe *Cell* représentera les cases du labyrinthe. Nous commencerons par la classe *Cell*, nécessaire pour la classe *Maze*.

3.1 Implémenter la classe Cell

Pour cette classe, il vous faudra compléter les FIXME suivant. Une cellule représente une case du tableau et contient les données suivantes :

— Les cellules voisines et les cellules joignables.
— Une cellule est joignable si elle est voisine et qu'aucun mur ne sépare les cellules.

Ces données faciliteront le travail lors de la génération et la résolution du labyrinthe. Votre classe *Cell* contiendra donc les champs suivants :

— Une liste de cellules voisines : neighbors.
— Une liste de cellules joignables : linked.
— Un booléen pour verifier si la case à déja été visitée.

Prototype :

```
public class Cell
{
    // FIXME: Add attributes : a neighbors list, a linked list, a boolean
    // isVisited.

    // FIXME: Initialization.
    public Cell() { }

    // FIXME: Add a cell to the neighbors list.
    public void addNeighbor(Cell c) { }

    // FIXME: Add a link between two cells and withdraw the wall between them.
    // Throw MazeException if there are not neighbors.
    public void addLink(Cell c) { }

    // FIXME: return true if c is next to the cell.
    public Boolean isNeighbor(Cell c) { }

    // FIXME: return true if c is linked with the cell.
    public Boolean isLinked(Cell c) { }

    // FIXME: return the list of neighboring cells.
    public List<Cell> getNeighbors() { }

    // FIXME: return the list of linked cells.
    public List<Cell> getLinked() { }

    // FIXME: Mix randomly the list of neighbors.
    // Use the random object passed as parameter.
    public void randomizeNeighbors(Random rng) { }
}
```

3.2 Implémenter la classe Maze

Pour cette classe, il vous faudra compléter les FIXME.
La classe *Maze* possèdera en attribut une matrice *cells* qui contiendra les cellules du labyrinthe. Ajoutez aussi la fonction d'affichage *print* du labyrinthe sur la console.

Prototype :

```
public class Maze
{
    // FIXME: Add attributes : list of cells, ...

    // throw MazeException if the size of the array is 0 or less
    // Add the cells and complete bonds between them.
    public Maze (int width, int height) { }

    // FIXME: Print the maze into the console.
    // You are free to print it the way you like.
    public void print() { }
}
```

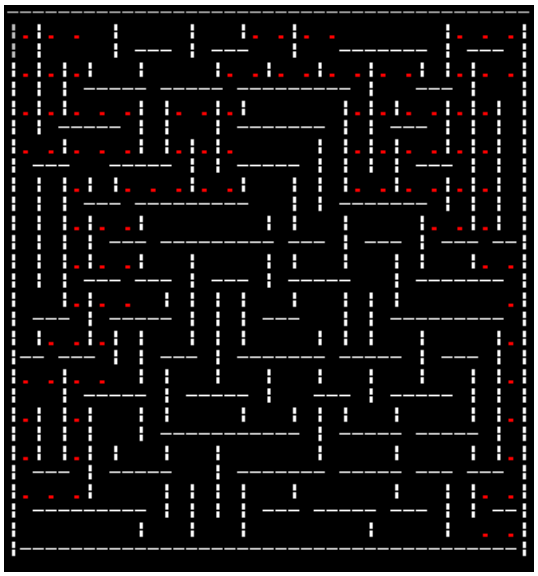


Figure 1 – Your maze may look like this with the solution

4 Génération du Labyrinthe

Il existe diverse méthodes pour générer un labyrinthe [1]. Pour ce Tp nous utiliserons la méthode par exploration exhaustive. Cette méthode est simple : On commence par mettre des murs entre toutes les cases adjacentes. Ensuite on choisit une case aléatoire et on parcourt ses voisines non visitées (Toujours de manière aléatoire pour obtenir différents labyrinthes) et ceci de manière recursive. À chaque saut, on marque la case comme visitée et on supprime le murs entre les deux cases. Pour savoir les cellules déjà visitées, on utilise la variable isVisited.

Voici la marche a suivre pour cet algorithme :

```
Marquer toutes les cellules comme non visitées.
Se placer sur une cellule aléatoire C et la marquer.
```

1. Voir `http://en.wikipedia.org/wiki/Maze_generation_algorithm`

```
Ensuite appliquer cet algorithme :
Marquer C comme visitée.
Tant que toutes les cellules voisines ne sont pas visitées faire :
    Choisir une cellule voisine non visitée aléatoirement.(C2)
    Supprimer le mur entre les cellules.
    Recommencer recursivement sur C2.
Fin tant que
```

Prototype :

```
public class Maze {
    // FIXME: Make a perfect maze with the Recursive backtracker algorithm
    public void generate() { }

    // FIXME
    void generate_rec(Cell c) { }
}
```

5 Résolution du Labyrinthe

Maintenant que nous avons généré un labyrinthe, nous allons voir comment le résoudre. Pour cela nous allons le parcourir jusqu'à trouver la case finale. Si vous vous trouvez sur la case finale, il faut alors renvoyer vrai. Ce booléen va alors remonter les appels recursifs et construire le chemin jusqu'a la sortie. La case de départ se trouve en haut à gauche et l'arrivée en bas à droite. Si toutes les voisines d'une cellule renvoie faux alors elle ne fait pas partie du chemin jusqu'à la case finale. Il faut bien sûr marquer les cellules visitées comme lors de la génération.

Cela donne l'algorithme suivant :

```
Marquer toutes les cellules comme non visitées.
Se placer sur l'entrée.
Tant que toutes les cellules joignables ne sont pas visitées faire :
    Choisir une cellule joignable aléatoirement.
    Marquer la cellule comme visitée.
    Si la cellule est l'arrivée :
        retourner Vrai
    Sinon Si toutes les cellules voisines sont visitées :
        retourner Faux
    Sinon
        Si le resultat de l'appel recursif est Faux :
            Ne rien faire.
        Sinon
            Ajouter la cellule au chemin.
        Fin si
    Fin si
Fin tant que
Retourner le chemin complet.
```

Profitez en pour implémenter une fonction qui affiche le labyrinthe avec sa solution.

Prototype :

```
class Maze
{
    public List<Cell> solve()
    {
        // FIXME: Solve the maze and return the path as a list.
        // Call solve_rec.
    }

    bool solve(Cell cur, Cell end, List<Cell> path)
    {
        // FIXME: Mark the cell as visited.
        // Visit randomly unvisited sons until you have reached the end of the
        // maze.
        // Return true if the cell is the final cell.
        // Else add the node if its belongs to the path.
        // Recursive call.
        // Return false.
    }

    // FIXME: print the maze with the path.
    public void print(List<Cell> path) { }
}
```

6 Interface

Dans votre *main*, ajoutez des instructions pour permettre à l'utilisateur de choisir la taille de son labyrinthe, en pensant à vérifier que la taille donnée est correcte. Ajoutez au moins une option pour générer le labyrinthe et une pour le résoudre.

7 Bonus

Les bonus sont libres mais voici quelques exemples :
— Générer et résoudre des labyrinthe imparfaits. (Vous pouvez regarder du côté des algorithmes dijkstra et A*)
— Utiliser la classe Bitmap pour créer une image de votre labyrinthe.
— Utiliser XNA pour afficher le labyrinthe.

It's dangerous to code alone !