# TP C# 10 - Maze

## 1 Report

For this subject we expects following files :

```
report-TPCSXX-login_x.zip
|-- report-TPCSXX-login_x/
    |-- AUTHORS
    |-- Bonus
        |-- Maze_bonus.cs
        |-- Cell_bonus.cs
    |-- maze.sln
    |-- maze
        |-- Everything but bin/ and obj/
```

## 2 Introduction

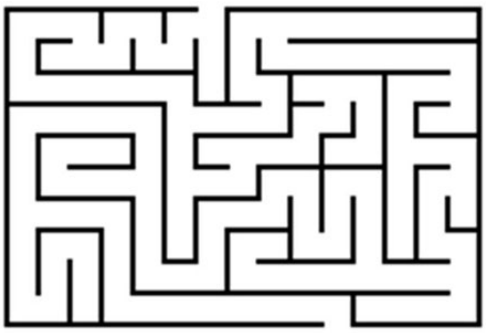
### 2.1 Goals

Today we will play with classes and mazes ! In this subject you will have to implement an algorithm that generate mazes and one that solves them. To implement a maze, we need two classes that will represent the maze. Once it's done we will start the hard part with the maze generation and resolution. Before starting here is a little reminder about maze

### 2.2 The mazes

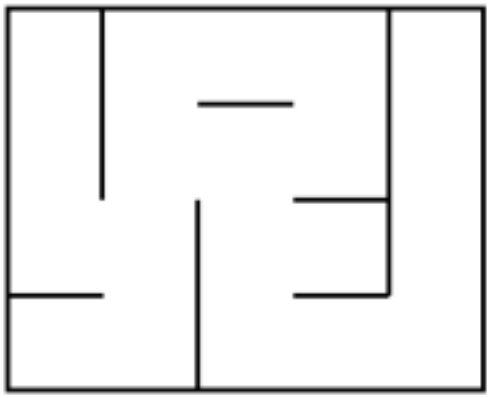
A maze is a surface that contains a wall and with one or more starting and ending points. In our case, the maze will have one start and one end and will be represented by a square delineated by walls. To represent that, we will use a two dimensional board filled with cells. Mazes have two types, perfect and imperfect.

The perfect maze :



It has no isolated wall and every square can be reached. You can go through by a simple depth run.

The imperfect maze :



It has an isolated wall that makes the run a little more complicated. For this subject we will implement a perfect maze.

# 3 Implementing the maze

*Maze* class will represent the maze and contains a two dimensional board. *Cell* class will represent a maze's cell. We will start with the *Cell* class, as we need it for the *Maze* class.

## 3.1 Implement the Cell class

For this class, you must complete the FIXME. The array is composed by cells and contains the following data :

— Neighboring cells and linked cells.

— A cell is linked if it is next to another and no walls are present between them.

These data will simplify the problem. So, your class will contain :

— A list of neighbors.

— A list of linked cells.

— And a boolean to know if the cell has been visited.

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 Implement the Maze class

For this class, you must complete the FIXME part. The *Maze* class has as attributes : the *cells* array variable which contains the cells. Add also the function *print* for the maze.

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() { }
}
```

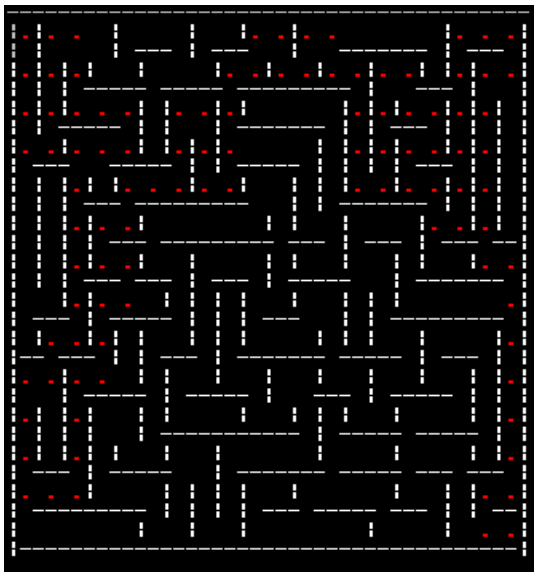


FIGURE 1 – Your maze may look like this with the solution

## 4 Generate the maze

There are lots of ways to generate a maze[1]. For this subject we will use the Recursive backtracker way. This method is simple : First, we put a wall between every neighbor cell. Then we choose a random cell and we move randomly on all of the unvisited neighbor cells and this on a recursive way. At each jump, we put the cell as visited and delete the wall between the two cells. To know if a cell is already visited we use the isVisited variable.
Here is the procedure you need to follow

```
Mark all the cell a unvisited
Choose a random cell and mark it.

Then follow this algorithm :
Mark C as visited
```

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

```
While all the neighbor cell are unvisited do :
    Choose a random neighbor cell(C2)
    Delete the wall between cell.
    Recursively start on C2
End while
```

Prototype :

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

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

## 5 Solving the maze

Now that we have a maze, let's see how to solve it. For this we are going to go through it since we found the end square, then we will return true. Thanks to recursivity, this result will go back to the first call of the solve function and build the way to the end. The starting square is in the top left corner and the ending in the bottom right corner. If every neighbor of a cell returns false, then it's not part of the solution. Of course, you need to mark every visited cell to avoid loop.

This gives us the following procedure :

```
Mark every cell as unvisited.
Go on the first cell.
While every contactable cell is not visited do
    Choose a random contactable cell.
    Mark it as visited.
    If it's the end :
        return True
    Else If every neighbor cell is visited :
        return False
    Else
        If the result of the recursive call is false :
            Do nothing
        Else
            Add the cell to the path.
        End If
    End If
End while
Return the complete path
```

When you're in the *Maze* class, add a function that displays the solved maze in console mode.

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

In your *main* add some instructions to allow the user to choose his maze size and remember to check if the given size is correct. Add at least two options, one for generating the maze and one to solve it.

# 7 Bonus

Bonuses are free but here are some examples :
— Generate and solve imperfect mazes. (You should take a look at dijkstra and A* algorithms)
— Use Bitmap class to create a picture of your maze.
— Use XNA to show your maze and make a character walk through it.

**It's dangerous to code alone !**