Preamble

As before. Submit the `HW4.java`.

Context

The HW adopts the following plan:

1. Solution of labyrinths by backtracking
2. Generation of labyrinths by backtracking in recursive version
3. Generation of labyrinths by backtracking in iterative version
4. Generation of uniformly distributed mazes with Wilson's algorithm

Code generation

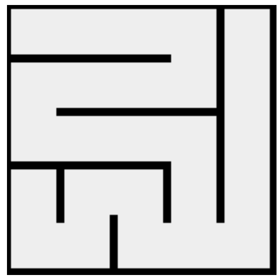
The code you just downloaded is organized into several classes. Here are their main functions:

- The class `Maze` models a maze. It contains the 2-dimensional array `grid` containing elements of type `Cell`. The size of the array is `heightxwidth`.
  - The method `getCell(i, j)` allows to access the box with the coordinates (`i`, `j`).
  - The method `getFirstCell()` returns the first box, that is, the box with coordinates $(0, 0)$.
- The class `Cell` models a square in a maze. Each square therefore has four possible neighboring squares: north, east, south, and west. A wall can block the path to each of these neighboring squares.
  - The method `getNeighbors(boolean ignoreWalls)` returns a list of neighboring cells. If the argument `ignoreWalls` is `true`, then all neighboring cells are returned. If the argument `ignoreWalls` is `false`, then only cells to which a passage exists are returned.
  - The method `breakWall(Cell c)` creates a passage from the current square to the square `c` given as an argument. It therefore removes a wall.
  - A mark can be placed on a square with the method `setMarked`. The existence of a mark can be tested with the method `isMarked`.

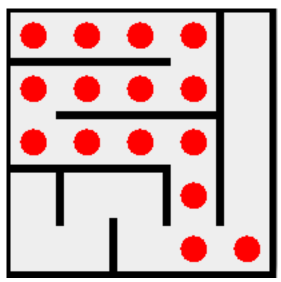
1 Solution of a maze by backtracking

In this first question, we want to find a path in a given maze, going from the northwest corner to the southeast corner, that is, from the cell with coordinates $(0, 0)$ to the exit, in our case the one with coordinates $(\texttt{height} - 1, \texttt{width} - 1)$. The `isExit` method in the `Cell` class can be useful: it returns `true` if and only if the current cell is the exit. The path must be indicated with marks on the cells. More precisely, a call to the `isMarked` method must return `true` for all cells on the path and `false` for all cells that are not on the path.

For example, in the 5×5 maze



the single path from northwest to southeast is



where a marked box is distinguished by a red circle.

Complete the method `searchPath()` of the class `ExtendedCell`, so that:

(1) it returns `true` if there exists between this and the exited cell a path not passing through any marked cell, and marks the cells of such a path;

(2) it returns `false` otherwise, and in this case, it does not modify any mark.

To find a path from the entrance to the exit in the maze, it will be enough to call the method on the Cell instance representing the $(0, 0)$ box. This is what is done in the tests.

You may notice that the method starts with the call `maze.slow()`. This is used to slow down the pathfinding animation so that it is observable to the naked eye.

Test with the `Test1.java`.

2 Maze generation by recursive backtracking

In the rest of the assignment, we will work on maze generation . As with the mazes in Question 1, they must be perfect, that is, they must have the property that there is a unique path between each pair of squares. Initially, all the squares are isolated (i.e., there are walls in all four directions).

Recursive backtracking for generation has the same general structure as pathfinding. One difference is that it is not necessary to mark any cells. Another difference is the condition for the recursive call: we want to continue only if the neighbor is isolated. Otherwise, it would lead to creating a cycle and the maze would not be perfect.

To avoid always ending up in the same maze, we want to run through the directions in random order. To do this, we can use the static method `shuffle` of the auxiliary class `Collections` which takes a list and randomly permutes it.

So we have the following simple structure for our algorithm in the `generateRec` method of the class `ExtendedCell`:

- For all adjacent squares in random order: if the neighbor is isolated, create a connection (by removing the wall with `breakWall`) and make a recursive call with the neighbor.

Test your method with the `Test2.java`.

3 Maze generation by iterative backtracking

We will now translate the recursive generation algorithm into an iterative version. This means that we exchange the call stack for a linked list that contains the coordinates of the cells we have started processing.

An iterative approach can have several advantages: less space on the call stack and more control over the order in which cells are visited. The call stack always chooses the most recent element. With an explicit list, this behavior can be changed.

The class `Bag` (implemented in the `Util.java` file, which you don't need to look at) offers several ways to choose the next element (of type `Cell`) from a list. The four different ways available are: `NEWEST`(we choose the newest element), `OLDEST`(for the oldest element), `MIDDLE`(the element in the middle of the list) and `RANDOM`(a random element from the list). For each of the four options, the method `peek()` returns the element selected according to the choice. The method `pop()` removes this element. Both methods assume a non-empty list. To test if the list is indeed empty, we can use the method `isEmpty()`. The method `add(Cell c)` allows adding a new element `c` to the end of the list. We also specify that the `peek()` and methods `pop()` are synchronized. In particular, if a cell is selected in the list with `peek()` and other cells are added to the list, the next call to `pop()` removes the previously selected cell.

To implement the iterative algorithm in the method `generateIter` of the class `Maze`, we can follow the following process. The algorithm starts with a list that contains only the element $(0, 0)$, then enters a loop, like this:

(1) If the list is not empty, select an item from the list using the selection method. Otherwise, finish.

(2) For the first isolated neighboring box in random order: create a passage, add the neighbor to the list and go to step (1).

(3) If you have exhausted all directions around the current square, remove that square from the list and return to step (1).

You may notice that a choice of `NEWEST` returns to the recursive version. A choice of `RANDOM` produces quite interesting mazes too, which is not the case for the choices `MIDDLE` or `OLDEST`.

Test your method with the `Test3.java`.

4 Uniform generation: Wilson's algorithm

We have just implemented random maze generation algorithms. The shape of the mazes generated by iterative backtracking strongly depended on the choice of the method for selecting the next cell to be processed. In particular, the generated mazes are not uniformly distributed across the set of all perfect mazes of the given size.

In this exercise we will therefore implement a uniform generation algorithm. The generation method chosen is called Wilson's algorithm . It proceeds as follows:

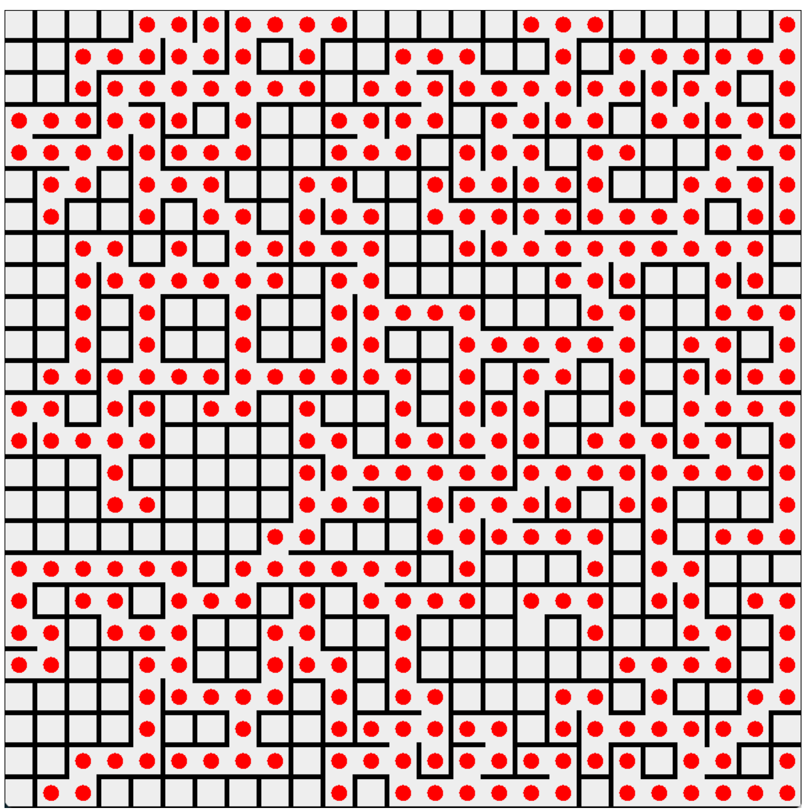
(1) Choose a box uniformly at random and mark it.

(2) As long as there is an unmarked square, choose an unmarked square uniformly at random, and

   (i) generate a uniform random walk from the chosen square until a marked square is encountered (ignore walls to generate the walk). Remove loops from the walk: only keep the last exit from each square.

   (ii) mark all the boxes and create the connections (by deleting the walls with `breakWall`) along the generated step.

Here is the algorithm in mid-execution:



Implement Wilson's algorithm by extending the method `generateWilson()` of the class `Maze`. To draw random uniform boxes successively, simply put all the boxes in a linked list and shuffle it. We can add to the class `Cell` a field `Cell next`, to keep track of the direction taken by the random walk the last time it leaves a box.

Test your method with the `Test4.java`.