Zagazig University
Faculty of Engineering
Computer and Systems Engineering Dept.
AI Min Project Report
year : 2017

Maze Solving

By,
Ahmed Mahmoud Ali
Anas Ahmed Fouad
Menna-Allah Ashraf
Mustafa Kamel El-Sayed

Supervisor,
Dr. Mohamed Amal
Eng.Amr Ahmed Zamel

This document is the report concerning the Laboratory done in the winter semester 2015 by our group. The group was composed of four students who studied Computer and Systems Engineering on the fourth academic year at Zagazig University, Faculty of Engineering.

Zagazig, Sunday, Feb 5, 2017.

Participants (alphabetically ordered):

Name	ID	Task(s)	Signature
Ahmed Mahmoud Ali احمد محمود علي	15	**Coding**	
Anas Ahmed Fouad انس احمد فؤاد	16	**Coding**	
Menna-Allah Ashraf منه الله اشرف	5	**Coding**	
Mustafa Kamel El-Sayed مصطفي كامل السيد	27	**Coding**	

Abstract:

In this project we aimed to solve any Maze using Breads First Search or Depth First Search.

The Program is written in JAVA and the results is represented using GUI instead AND CL interface.

Both are shown when user attempts to solve the maze via the program.

5 button are used by the GUI, first button is “Solve DFS” used to solve using Depth First Search.
Second button is “Solve BFS” used to solve using Depth First Search.
Third button is “Clear” used to clear any traces of previous attempts to solve the maze.
Fourth button is “Exit” used to close the GUI interface and Halt the program.
Fifth button is “Generate Random Maze” used to clear the maze screen and generate a new maze that may or may not be solvable.

1 Introduction

1.1 Introduction

In this project we'll be solving a maze using DFS and BFS algorithms.
The program will be written using Java SE.
IDE used is Netbeans .

2 Proplem formulation

DFS formulation took the shape of Stack, whichever comes last will be explored first.

BFS formulation took the shape of Queue, whichever comes first will be explored first.

3 Proposed Algorithms

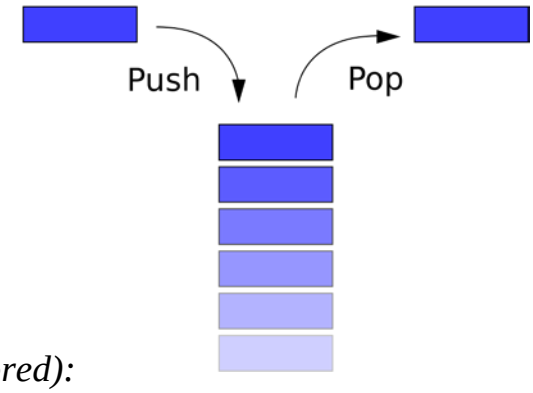
We used Two algorithms to solved the maze problem, solveStack() and solveQueue().

3.1 solveStack()

The algorithm is an implimintaion of Stack with the comands push and pop.

Sudo Code:

```
Insert initial state in stack;
while (sack is not empty):
        Pop;
        If(node is goal):
                Break;
        Mark the node as explored;
        Put all neighbours in next;
        If (neighbour is in maze and not explored):
                Push;
Repeat;
```

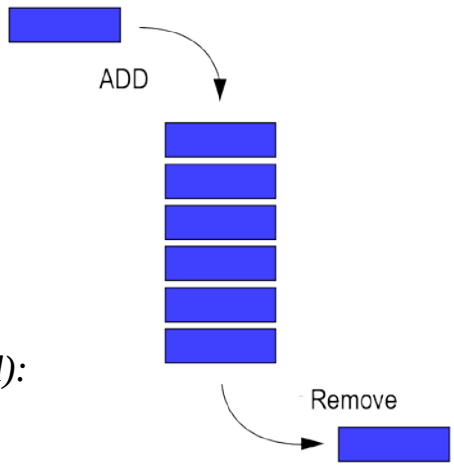


3.2 solveQueue()

The algorithm is an implimintaion of Queue with the commands add and remove.

Sudo Code:

```
Insert initial state in queue;
while (queue is not empty):
        remove first;
        If(node is goal):
                Break;
        Mark the node as explored;
        Put all neighbours in next;
        If (neighbour is in maze and not explored):
                Add to queue;
Repeat;
```



4 results and comparison between algorithms

There are no true difference between the two algorithms, both are types of uninformed search.

There is no rule to govern which of them will execute faster.

Advantages:

DFS and BFS are very handy when trying to solve small problem and don't want too much memory and processing concussion. Also we don't care about the cost to the goal, we just want a solution.

Disadvantages:

When it comes down to cost then both BFS and DFS are not suitable. Both don't put path cost in consideration, they just roam the problem looking for the first legal solution.

Other informed algorithms must be used to find path with lest cost.

5 comments

You should not rely entirely on this documentation as it's just words with no actual implementation, we highly recommend that you ignore this documentation completely and rely on the comments in the code and the video tutorial that we made to further simplify the code. But none the less, the comments alone will be sufficient.

In this project, we learned a lot about GUI in java and enhanced our knowledge of java syntax.

6 Appendix1 (Code with comment)

Main Class

```
package maze;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.net.URL;
import java.util.LinkedList;
import java.util.Random;
import java.util.Stack;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class Maze extends JFrame {
    //assign numbers for every color that will be used, colors are
```

```
defined later (line:349 in paint)
        //block (black square)
        final static int X = 1;
        //free space   (white Square)
        final static int C = 0;
        //initial state
        final static int S = 2;
        //goal
        final static int E = 8;
        // the path
        final static int V = 9;

        //initial state   (i,j)
        final static int START_I = 1, START_J = 1;
        //goal  (i,j)
        final static int END_I = 2, END_J = 9;

        int[][] maze = new int[][]{ // the initial array for the maze
            {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
            {1, 2, 0, 0, 0, 0, 0, 0, 0, 1},
            {1, 0, 0, 0, 1, 0, 1, 1, 0, 8},
            {1, 0, 1, 1, 1, 0, 1, 1, 0, 1},
            {1, 0, 0, 0, 0, 1, 1, 1, 0, 1},
            {1, 1, 1, 1, 0, 1, 1, 1, 0, 1},
            {1, 1, 1, 1, 0, 1, 0, 0, 0, 1},
            {1, 1, 0, 1, 0, 1, 1, 0, 0, 1},
            {1, 1, 0, 0, 0, 0, 0, 0, 0, 1},
            {1, 1, 1, 1, 1, 1, 0, 1, 1, 1}
        };

        // for random array which we will store the randomly generated
array in.
        int[][] arr;

        // Buttons For GUI (still not initialized)
        JButton solveStack;
        JButton solveBFS;
        JButton clear;
        JButton exit;
        JButton genRandom;

        // JLabel and JTextField For GUI (still not initialized)
        JLabel elapsedDfs;
        JTextField textDfs;
        JLabel elapsedBFS;
        JTextField textBFS;

        boolean repaint = false;  //(line:349 in paint) the value will
define what we want to paint on the Jframe, solve the maze or paint the
maze

        // start time
        long startTime;
        //stop time
        long stopTime;
        //calculate the elapsed time
        long duration;
        //time for DFS
        double dfsTime;
        //time for BFS
        double bfsTime;

       // take copy of the original maze, used when we want to remove
(clear) the solution from the JFrame
       int[][] savedMaze = clone();
```

```
        // the maze class constructor, this will be the first thing to be
executed when we creat an object from this class
        public Maze() {

            setTitle("Maze");     //Title For JFrame
            setSize(960, 530);    // Size For JFrame  (width,height)


            URL urlIcon = getClass().getResource("flat-theme-action-maze-
icon.png");    // Path for image for The JFrame
            ImageIcon image = new ImageIcon(urlIcon);
// store the image in variable named image
            setIconImage(image.getImage());
// set The Image For JFrame

            setLocationRelativeTo(null);
// to make JFrame appear in the Middle of the screen
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// To close the app when click on exit or (X)
            setLayout(null);
// to set the position of the components (like JLabel and JTextField,
Buttons) and by hand (we will choose the position by ourselves)

            // initialize objects for Jlabel and JTextField
            elapsedDfs = new JLabel("Elapsed Time :");
            elapsedBFS = new JLabel("Elapsed Time :");
            textDfs = new JTextField();
            textBFS = new JTextField();

            // initialize objects for Buttons
            solveStack = new JButton("Solve DFS");
            solveBFS = new JButton("Solve BFS");
            clear = new JButton("Clear");
            exit = new JButton("Exit");
            genRandom = new JButton("Generate Random Maze");

            // Add The Buttons to JFrame
            add(solveStack);
            add(solveBFS);
            add(clear);
            add(elapsedDfs);
            add(textDfs);
            add(elapsedBFS);
            add(textBFS);
            add(exit);
            add(genRandom);

            // make JFrame visible (it's invisible by default, we don't
know why!!)
            setVisible(true);

            // set the positions of the components on the JFrame
(x,y,width,height). here we chose the position by hand, this is why we sat
the set Layout to null
            solveStack.setBounds(500, 50, 100, 40);
            solveBFS.setBounds(630, 50, 100, 40);
            clear.setBounds(760, 50, 100, 40);
            exit.setBounds(760, 115, 100, 40);
            elapsedDfs.setBounds(500, 100, 100, 40);
            genRandom.setBounds(500, 180, 170, 40);
            elapsedBFS.setBounds(630, 100, 100, 40);
            textDfs.setBounds(500, 130, 100, 25);
            textBFS.setBounds(630, 130, 100, 25);

            // what happen when click on Generate Random Maze Button
            genRandom.addActionListener(new ActionListener() {
```

```
                @Override
                public void actionPerformed(ActionEvent e) {

                    int x[][] = GenerateArray();  // generate random array
and store in x
                    repaint = true;//(line:349 in paint) the value will
define what we want to paint on the Jframe, solve the maze or paint the
maze (in this case "paint the maze")
                    restore(x);   // make the array maze equal the array x
|| to show the array x on screen
                    repaint();   // repaint the maze on the JFrame
                }
            });

            // what happen when click on Exit Button
            exit.addActionListener(new ActionListener() {

                @Override
                public void actionPerformed(ActionEvent e) {
                    System.exit(0);        //Close The App
                }
            });

            // what happen when click on Clear Button
            clear.addActionListener(new ActionListener() {

                @Override
                public void actionPerformed(ActionEvent e) {

                    if (arr == null) {       // happens if the random
array (resulted when we press random button) is empty (we didn't generate a
random array)
                        repaint = true;      // set to repaint maze layout
(not maze solution)
                        restore(savedMaze);  // restore the maze to the
original, we use a clone() method (line:298 ) to make a copy
                        repaint();           // repaint the maze
(savedMaze) on the JFrame
                    } else {                 // happens if a random array
was generated
                        repaint = true;      // set to repaint maze layout
(not maze solution)
                        restore(arr);        // to make the array arr
equal to the array maze || to show the array arr on the screen
                        repaint();           // repaint the maze on the
JFrame
                    }

                    textBFS.setText("");       // clear the text in
JTextField, the one indicating the elapsed time of BFS
                    textDfs.setText("");       // clear the text in
JTextField, the one indicating the elapsed time of DFS

                }
            });

            // what happen when click on Solve DFS Button
            solveStack.addActionListener(new ActionListener() {

                @Override
                public void actionPerformed(ActionEvent e) {
                    if (arr == null) {       // happens if the random
array (resulted when we press random button) is empty (we didn't generate a
random array)
                        restore(savedMaze);  // restore the maze to the
original
```

```
                        repaint = false;     // set to repaint maze
solution (not maze layout)
                        solveStack();        // call method solveStack()
which solves the maze using DFS
                        repaint();           // repatin the maze on the
JFrame
                    } else {                 // happens if a random array
was generated

                        restore(arr);        // to make the array arr
equal to the array maze || to show the array arr on the screen
                        repaint = false;     // set to repaint maze
solution (not maze layout)
                        solveStack();        // call method solveStack()
which solves the maze using DFS
                        repaint();           // repatin the maze on the
JFrame

                    }

                }
            });

            // what happen when click on Solve BFS Button
            solveBFS.addActionListener(new ActionListener() {

                @Override
                public void actionPerformed(ActionEvent e) {

                    if (arr == null) {       // happens if the random
array (resulted when we press random button) is empty (we didn't generate a
random array)
                        restore(savedMaze);  // restore the maze to the
original
                        repaint = false;     // set to repaint maze
solution (not maze layout)
                        solveQueue();        // call method solveQueue()
which solves the maze using BFS
                        repaint();           // repatin the maze on the
JFrame
                    } else {                 // happens if a random array
was generated
                        restore(arr);       // to make the array arr equal
to the array maze || to show the array arr on the screen
                        repaint = false;    // set to repaint maze
solution (not maze layout)
                        solveQueue();       // call method solveQueue()
which solves the maze using BFS
                        repaint();          // repatin the maze on the
JFrame
                    }

                }
            });

        }

        // get size of the maze
        public int Size() {
            return maze.length;
        }

        // Print the Maze to CL
        public void Print() {
            for (int i = 0; i < Size(); i++) {      //go to every row
                for (int J = 0; J < Size(); J++) {  //go to every element
```

```
in the row
                    System.out.print(maze[i][J]);   //print the element
                    System.out.print(' ');          //print space
                }
                System.out.println();               // go to new line
            }
        }

        //return true if cell is within maze
        public boolean isInMaze(int i, int j) {  //parameters are the
position (i and j) of the cell

            if (i >= 0 && i < Size() && j >= 0 && j < Size()) {
                return true;
            } else {
                return false;
            }
        }

        public boolean isInMaze(MazePos pos) {   //overloaded of
isInMaze(int i, int j) , parameter is the node itself
            return isInMaze(pos.i(), pos.j());   //extract the position of
the cell (i and j) and call the first method isInMaze(int i, int j)
        }

        // to mark the node in the array with a certain value; ex: if
explored mark with value 9 (Green)
        public int mark(int i, int j, int value) {
            assert (isInMaze(i, j));  // it is used for test. if the
condition is false it will throw an error named AssertionError.
            int temp = maze[i][j];    // store the original value in temp
            maze[i][j] = value;       // put the value from the parameter
in maze cell with corresponding i,j
            return temp;              // return original value
        }

        public int mark(MazePos pos, int value) {   //overloaded of
mark(int i, int j, int value) , parameter is the node itself and the value
we want to insert
            return mark(pos.i(), pos.j(), value);   //extract the position
of the cell (i and j) and call the first method mark(int i, int j, int
value)
        }

        // return true if the node equal to v=9 (Green, Explored)
        public boolean isMarked(int i, int j) {
            assert (isInMaze(i, j));
            return (maze[i][j] == V);

        }

        public boolean isMarked(MazePos pos) {   //overloaded of
isMarked(int i, int j) , parameter is the node itself
            return isMarked(pos.i(), pos.j());   //extract the position of
the cell (i and j) and call the first method isMarked(int i, int j)
        }

        // return true if the node is equal to 0 (White, Unexplored)
        public boolean isClear(int i, int j) {
            assert (isInMaze(i, j));
            return (maze[i][j] != X && maze[i][j] != V);

        }

        public boolean isClear(MazePos pos) {   //overloaded of
isClear(int i, int j) , parameter is the node itself
```

```
            return isClear(pos.i(), pos.j());   //extract the position of
the cell (i and j) and call the first method isClear(int i, int j)
        }

        // to make sure if it is reach the goal (Goal Test)
        public boolean isFinal(int i, int j) {

            return (i == Maze.END_I && j == Maze.END_J);
        }

        public boolean isFinal(MazePos pos) {  //overloaded of isFinal(int
i, int j) , parameter is the node itself
            return isFinal(pos.i(), pos.j());  //extract the position of
the cell (i and j) and call the first method isFinal(int i, int j)
        }

        // make Copy from the original maze
        public int[][] clone() { //used to create savedMaze[][] we already
discussed its use before
            int[][] mazeCopy = new int[Size()][Size()];
            for (int i = 0; i < Size(); i++) {
                for (int j = 0; j < Size(); j++) {
                    mazeCopy[i][j] = maze[i][j];
                }
            }
            return mazeCopy;
        }


        // to restore the maze to the initial state
        public void restore(int[][] savedMazed) {
            for (int i = 0; i < Size(); i++) {
                for (int j = 0; j < Size(); j++) {
                    maze[i][j] = savedMazed[i][j];
                }
            }

            maze[1][1] = 2;  // the start point
            maze[2][9] = 8;  // the goal
        }

        //generate random maze whith values 0 and 1
        public int[][] GenerateArray() {
            arr = new int[10][10];
            Random rnd = new Random();
            int min = 0;
            int high = 1;

            for (int i = 0; i < 10; i++) {
                for (int j = 0; j < 10; j++) {
                    int n = rnd.nextInt((high - min) + 1) + min;
                    arr[i][j] = n;

                }
            }
              arr[0][1] = 0;arr[1][0] = 0;arr[2][1] = 0;arr[1][2] =
0;//make sure all paths from initial state are legal moves (white block)
              arr[1][9] = 0;arr[2][8] = 0;arr[3][9] = 0;
//make sure all paths to goal are legal moves (white block)
     //        for(int i=0; i<10;i++){
     //           for(int j=0;j<10;j++){
     //               System.out.print(arr[i][j]);
     //               System.out.print(' ');
     //            }
     //           System.out.println("");
     //          }
```

```
            return arr;
        }

        //draw the maze on the JFrame
        @Override
        public void paint(Graphics g) {
            super.paint(g);
            g.translate(70, 70);      //move the maze to begin at 70 from
x and 70 from y

            // draw the maze
            if (repaint == true) {  // what to do if the repaint was set
to true (draw the maze as a problem without the solution)
                for (int row = 0; row < maze.length; row++) {
                    for (int col = 0; col < maze[0].length; col++) {
                        Color color;
                        switch (maze[row][col]) {
                            case 1:
                                color = Color.darkGray;       // block
(black)
                                break;
                            case 8:
                                color = Color.RED;          // goal (red)
                                break;
                            case 2:
                                color = Color.YELLOW;      //initial state
(yellow)
                                break;
                            //   case '.' : color=Color.ORANGE; break;
                            default:
                                color = Color.WHITE;        // white free
space 0  (white)
                        }
                        g.setColor(color);
                        g.fillRect(40 * col, 40 * row, 40, 40);    // fill
rectangular with color
                        g.setColor(Color.BLUE);                    //the
border rectangle color
                        g.drawRect(40 * col, 40 * row, 40, 40);    //draw
rectangular with color

                    }
                }
            }

            if (repaint == false) {   // what to do if the repaint was set
to false (draw the solution for the maze)
                for (int row = 0; row < maze.length; row++) {
                    for (int col = 0; col < maze[0].length; col++) {
                        Color color;
                        switch (maze[row][col]) {
                            case 1:
                                color = Color.darkGray;     // block
(black)
                                break;
                            case 8:
                                color = Color.RED;         // goal  (red)
                                break;
                            case 2:
                                color = Color.YELLOW;      //initial state
(yellow)
                                break;
                            case 9:
                                color = Color.green;   // the path from
the initial state to the goal
                                break;
```

```
                            default:
                                color = Color.WHITE;   // white free space
0  (white)
                        }
                        g.setColor(color);
                        g.fillRect(40 * col, 40 * row, 40, 40);  // fill
rectangular with color
                        g.setColor(Color.BLUE);                  //the
border rectangle color
                        g.drawRect(40 * col, 40 * row, 40, 40);  //draw
rectangular with color

                    }

                }

            }

        }

        public static void main(String[] args) {  // the main program

            SwingUtilities.invokeLater(new Runnable() {  // run the
program through Swing (the entire program is run by GUI)
                                                         // we chose
invokelater it won't make much difference if we chose invokeAndWait since
the operation done by the first button will be done in a very short time
                @Override                                // you can read
more here:
https://docs.oracle.com/javase/tutorial/uiswing/concurrency/initial.html
                public void run() {
                    Maze maze = new Maze();              // we create new
class which will invoke the constructor

                }
            });

        }

        public void solveStack() { //DFS correspond to Stack
            // start of the time
            startTime = System.nanoTime();

            //create stack of MazPos (MazPos (the node) is what we will be
pushing and popping from the stack)
            Stack<MazePos> stack = new Stack<MazePos>();

            //insert the start node
            stack.push(new MazePos(START_I, START_I));

            MazePos crt;   //current node
            MazePos next;  //next node
            while (!stack.empty()) {//while stack not empty

                //get current position by popping from stack
                crt = stack.pop();
                if (isFinal(crt)) { //if the goal is reached then exit, no
need for further exploration.

                    break;
                }

                //mark the current position as explored
                mark(crt, V);

                //push its neighbours in the stack
```

```
                next = crt.north();    // go up from the current node
                if (isInMaze(next) && isClear(next)) {  //isClear() method
is used to implement Graph Search (in tree we can reExplore nodes, could
get stuck in infinite loop which happened to up in previous builds)
                    stack.push(next);
                }
                next = crt.east();    //go right from the current node
                if (isInMaze(next) && isClear(next)) {
                    stack.push(next);
                }
                next = crt.west();    //go left from the current node
                if (isInMaze(next) && isClear(next)) {
                    stack.push(next);
                }
                next = crt.south();  // go down from the current node
                if (isInMaze(next) && isClear(next)) {
                    stack.push(next);
                }
            }

            if (!stack.empty()) {              //you exited before the
stack was emptied, meaning that you found the solution and exited.
                 stopTime = System.nanoTime();
                JOptionPane.showMessageDialog(rootPane, "You Got it");

            } else {                           //the stack is empty,
meaning that you went through all nodes and didn't find the solution.
                JOptionPane.showMessageDialog(rootPane, "You Are stuck in
the maze");
            }

            System.out.println("\nFind Goal By DFS : ");
            Print();
            // stop time

            duration = stopTime - startTime;    //calculate the elapsed
time

            dfsTime =  (double)duration / 1000000;   //convert to ms
            System.out.println(String.format("Time %1.3f ms", dfsTime));

            textDfs.setText(String.format("%1.3f ms", dfsTime));

        }

        public void solveQueue() { //BFS correspond to Queue.
            //start the timer
            startTime = System.nanoTime();

            //create LinkedList of MazPos (MazPos (the node) is what we
will be adding and removing from the LinkedList)
            LinkedList<MazePos> list = new LinkedList<MazePos>();

            // add initial node to the list
            list.add(new MazePos(START_I, START_J));

            MazePos crt, next;
            while (!list.isEmpty()) {

                //get current position
                crt = list.removeFirst();

                // to be sure if it reach the goal
                if (isFinal(crt)) { //if the goal is reached then exit, no
need for further exploration.
                    break;
```

```
                }

                //mark the current position as explored
                mark(crt, V);

                //add its neighbors in the queue
                next = crt.north();    //move up
                if (isInMaze(next) && isClear(next)) { //isClear()
function is used to implement Graph Search
                    list.add(next);
                }
                next = crt.east();    //move right
                if (isInMaze(next) && isClear(next)) {
                    list.add(next);
                }
                 next = crt.west();    //move left
                if (isInMaze(next) && isClear(next)) {
                    list.add(next);
                }
                next = crt.south();   //move down
                if (isInMaze(next) && isClear(next)) {
                    list.add(next);
                }

            }

            if (!list.isEmpty()) {              //you exited before the
list was emptied, meaning that you found the solution and exited.
                stopTime = System.nanoTime();    //stop the timer
                JOptionPane.showMessageDialog(rootPane, "You Got it");
            } else {                            //the list is empty,
meaning that you went through all nodes and didn't find the solution.
                JOptionPane.showMessageDialog(rootPane, "You Are stuck in
the maze");
            }

            System.out.println("\nFind Goal By BFS : ");
            Print();

            duration = stopTime - startTime;      //calculate the elapsed
time

            bfsTime = (double) duration / 1000000;     //convert to ms
            System.out.println(String.format("Time %1.3f ms", bfsTime));

            textBFS.setText(String.format("%1.3f ms", bfsTime));


        }

    }
```

Secondary class

```
package maze;
public class MazePos {

    int i,j;                    // cell position

    public MazePos(int i, int j){
     this.i=i;
     this.j=j;
    };
    public int i() { return i;}   // get i (for overloaded methods)

    public int j() { return j;}   // get j (for overloaded methods)

    public void Print(){
       System.out.println("(" + i + "," + j + ")");  //print the
position
    }

    // go up
    public MazePos north(){
      return new MazePos(i-1,j);
    }

    //go down
    public MazePos south(){
        return new MazePos(i+1 , j);
    }

    //go right
    public MazePos east(){
        return new MazePos(i,j+1);
    }

    //go left
    public MazePos west(){
      return new MazePos(i,j-1);
    }

}
```