package mazeSolverBySamim;

import java.awt.\*;

import java.awt.event.\*;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Random;

import javax.swing.\*;

public class myMaze {

public static JFrame mazeFrame; // The main form of the program

public static void main(String[] args) {

int width = 500;

int height = 545;

mazeFrame = new JFrame("Maze Solver By Samim");

mazeFrame.setContentPane(new MazePanel(width,height));

mazeFrame.pack();

mazeFrame.setResizable(false);

// the form is located in the center of the screen

Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();

double screenWidth = screenSize.getWidth();

double ScreenHeight = screenSize.getHeight();

int x = ((int)screenWidth-width)/2;

int y = ((int)ScreenHeight-height)/2;

mazeFrame.setLocation(x,y);

mazeFrame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

mazeFrame.setVisible(true);

} // end main()

/\*

\* This class defines the contents of the main form

\* and contains all the functionality of the program.

\*/

public static class MazePanel extends JPanel {

/\*

\* Cell class that represents the cell of the grid

\*/

private class Cell {

int row; // the row number of the cell(row 0 is the top)

int col; // the column number of the cell (Column 0 is the left)

int g; // the value of the function g of A\* algorithm

int h; // the value of the function h of A\* algorithm

int f; // the value of the function h of A\* algorithm

Cell prev; /\* Each state corresponds to a cell

and each state has a predecessor which

is stored in this variable \*/

public Cell(int row, int col){

this.row = row;

this.col = col;

}

} // end nested class Cell

/\*

\* When the user presses a button performs the corresponding functionality

\*/

private class ActionHandler implements ActionListener {

@Override

public void actionPerformed(ActionEvent evt) {

String cmd = evt.getActionCommand();

if (cmd.equals("Clear")) {

fillGrid();

dfs.setEnabled(true);

bfs.setEnabled(true);

aStar.setEnabled(true);

}

else if (cmd.equals("Solve") && !endOfSearch) {

searching = true;

dfs.setEnabled(false);

bfs.setEnabled(false);

aStar.setEnabled(false);

timer.setDelay(80);

timer.start();

}

}

} // end nested class ActionHandler

/\*

\* The class that is responsible for the animation

\*/

private class RepaintAction implements ActionListener {

@Override

public void actionPerformed(ActionEvent evt) {

expandNode();

if (found) {

timer.stop();

endOfSearch = true;

plotRoute();

}

repaint();

}

} // end nested class RepaintAction

// Class that creates a random maze

private class MyMaze {

private int dimensionX, dimensionY; // dimension of maze

private int gridDimensionX, gridDimensionY; // dimension of output grid

private char[][] mazeGrid; // output grid

private Cell[][] cells; // 2d array of Cells

private Random random = new Random(); // The random object

// constructor

public MyMaze(int xDimension, int yDimension) {

dimensionX = xDimension;

dimensionY = yDimension;

gridDimensionX = xDimension \* 2 + 1;

gridDimensionY = yDimension \* 2 + 1;

mazeGrid = new char[gridDimensionX][gridDimensionY];

init();

generateMaze();

}

private void init() {

// create cells

cells = new Cell[dimensionX][dimensionY];

for (int x = 0; x < dimensionX; x++) {

for (int y = 0; y < dimensionY; y++) {

cells[x][y] = new Cell(x, y, false); // create cell

}

}

}

// inner class to represent a cell

private class Cell {

int x, y; // coordinates

// cells this cell is connected to

ArrayList<Cell> neighbors = new ArrayList<>();

// Impossible cell

boolean wall = true;

// if true, has yet to be used in generation

boolean open = true;

// construct Cell at x, y

Cell(int x, int y) {

this(x, y, true);

}

// construct Cell at x, y and with whether it isWall

Cell(int x, int y, boolean isWall) {

this.x = x;

this.y = y;

this.wall = isWall;

}

// add a neighbor to this cell, and this cell as a neighbor to the other

void addNeighbor(Cell other) {

if (!this.neighbors.contains(other)) { // avoid duplicates

this.neighbors.add(other);

}

if (!other.neighbors.contains(this)) { // avoid duplicates

other.neighbors.add(this);

}

}

// used in updateGrid()

boolean isCellBelowNeighbor() {

return this.neighbors.contains(new Cell(this.x, this.y + 1));

}

// used in updateGrid()

boolean isCellRightNeighbor() {

return this.neighbors.contains(new Cell(this.x + 1, this.y));

}

// useful Cell equivalence

@Override

public boolean equals(Object other) {

if (!(other instanceof Cell)) return false;

Cell otherCell = (Cell) other;

return (this.x == otherCell.x && this.y == otherCell.y);

}

}

// generate from upper left

private void generateMaze() {

generateMaze(0, 0);

}

// generate the maze from coordinates x, y

private void generateMaze(int x, int y) {

generateMaze(getCell(x, y)); // generate from Cell

}

private void generateMaze(Cell startAt) {

// don't generate from cell not there

if (startAt == null) return;

startAt.open = false; // indicate cell closed for generation

ArrayList<Cell> cellsList = new ArrayList<>();

cellsList.add(startAt);

while (!cellsList.isEmpty()) {

Cell cell;

if (random.nextInt(10)==0)

cell = cellsList.remove(random.nextInt(cellsList.size()));

else cell = cellsList.remove(cellsList.size() - 1);

// for collection

ArrayList<Cell> neighbors = new ArrayList<>();

// cells that could potentially be neighbors

Cell[] potentialNeighbors = new Cell[]{

getCell(cell.x + 1, cell.y),

getCell(cell.x, cell.y + 1),

getCell(cell.x - 1, cell.y),

getCell(cell.x, cell.y - 1)

};

for (Cell other : potentialNeighbors) {

// skip if outside, is a wall or is not opened

if (other==null || other.wall || !other.open) continue;

neighbors.add(other);

}

if (neighbors.isEmpty()) continue;

// get random cell

Cell selected = neighbors.get(random.nextInt(neighbors.size()));

// add as neighbor

selected.open = false; // indicate cell closed for generation

cell.addNeighbor(selected);

cellsList.add(cell);

cellsList.add(selected);

}

updateGrid();

}

// used to get a Cell at x, y; returns null out of bounds

public Cell getCell(int x, int y) {

try {

return cells[x][y];

} catch (ArrayIndexOutOfBoundsException e) { // catch out of bounds

return null;

}

}

// draw the maze

public void updateGrid() {

char backChar = '\*', wallChar = 'X', cellChar = ' ';

// fill background

for (int x = 0; x < gridDimensionX; x ++) {

for (int y = 0; y < gridDimensionY; y ++) {

mazeGrid[x][y] = backChar;

}

}

// build walls

for (int x = 0; x < gridDimensionX; x ++) {

for (int y = 0; y < gridDimensionY; y ++) {

if (x % 2 == 0 || y % 2 == 0)

mazeGrid[x][y] = wallChar;

}

}

// make meaningful representation

for (int x = 0; x < dimensionX; x++) {

for (int y = 0; y < dimensionY; y++) {

Cell current = getCell(x, y);

int gridX = x \* 2 + 1, gridY = y \* 2 + 1;

mazeGrid[gridX][gridY] = cellChar;

if (current.isCellBelowNeighbor()) {

mazeGrid[gridX][gridY + 1] = cellChar;

}

if (current.isCellRightNeighbor()) {

mazeGrid[gridX + 1][gridY] = cellChar;

}

}

}

// creates a clean grid ...

searching = false;

endOfSearch = false;

fillGrid();

// created by the maze construction algorithm

for (int x = 0; x < gridDimensionX; x++) {

for (int y = 0; y < gridDimensionY; y++) {

if (mazeGrid[x][y] == wallChar && grid[x][y] != ROBOT && grid[x][y] != TARGET){

grid[x][y] = OBST;

}

}

}

}

} // end nested class MyMaze

//Constants of class MazePanel

private final static int

EMPTY = 0, // empty cell

OBST = 1, // cell with obstacle

ROBOT = 2, // the position of the robot

TARGET = 3, // the position of the target

FRONTIER = 4, // cells that form the frontier (OPEN SET)

CLOSED = 5, // cells that form the CLOSED SET

ROUTE = 6; // cells that form the robot-to-target path

// Variables of class MazePanel

int rows = 41, // the number of rows of the grid

columns = 41, // the number of columns of the grid

squareSize = 500/rows; // the cell size in pixels

ArrayList<Cell> openSet = new ArrayList();// the OPEN SET

ArrayList<Cell> closedSet = new ArrayList();// the CLOSED SET

Cell robotStart; // the initial position of the robot

Cell targetPos; // the position of the target

// buttons for selecting the algorithm

JRadioButton dfs, bfs, aStar;

int[][] grid; // the grid

boolean found; // flag that the goal was found

boolean searching; // flag that the search is in progress

boolean endOfSearch; // flag that the search came to an end

int delay; // time delay of animation (in msec)

// the object that controls the animation

RepaintAction action = new RepaintAction();

// the Timer which governs the execution speed of the animation

Timer timer;

// The creator of the panel

public MazePanel(int width, int height) {

setLayout(null);

setBorder(BorderFactory.createMatteBorder(2,2,2,2,Color.black));

setPreferredSize( new Dimension(width,height) );

grid = new int[rows][columns];

JButton mazeButton = new JButton("Maze");

mazeButton.addActionListener(new ActionHandler());

mazeButton.setBackground(Color.yellow);

mazeButton.setToolTipText

("Creates a random maze");

mazeButton.addActionListener(new java.awt.event.ActionListener() {

@Override

public void actionPerformed(java.awt.event.ActionEvent evt) {

mazeButtonActionPerformed(evt);

}

});

JButton clearButton = new JButton("Clear");

clearButton.addActionListener(new ActionHandler());

clearButton.setBackground(Color.yellow);

clearButton.setToolTipText

("Clears the grid");

JButton animationButton = new JButton("Solve");

animationButton.addActionListener(new ActionHandler());

animationButton.setBackground(Color.yellow);

animationButton.setToolTipText

("Solves the maze");

/\* ButtonGroup that synchronizes the three RadioButtons

choosing the algorithm, so that only one

can be selected anytime \*/

ButtonGroup algoGroup = new ButtonGroup();

dfs = new JRadioButton("DFS");

dfs.setToolTipText("Depth First Search algorithm");

algoGroup.add(dfs);

dfs.addActionListener(new ActionHandler());

bfs = new JRadioButton("BFS");

bfs.setToolTipText("Breadth First Search algorithm");

algoGroup.add(bfs);

bfs.addActionListener(new ActionHandler());

aStar = new JRadioButton("A\*");

aStar.setToolTipText("A\* algorithm");

algoGroup.add(aStar);

aStar.addActionListener(new ActionHandler());

dfs.setSelected(true); // DFS is initially selected

// the contents of the panel

add(mazeButton);

add(clearButton);

add(animationButton);

add(dfs);

add(bfs);

add(aStar);

// sizes and positions

mazeButton.setBounds(10, 510, 80, 25);

clearButton.setBounds(425, 510, 80, 25);

animationButton.setBounds(330, 510, 80, 25);

dfs.setBounds(120, 510, 70, 25);

bfs.setBounds(190, 510, 70, 25);

aStar.setBounds(260, 510, 70, 25);

// the timer

timer = new Timer(delay, action);

// Here is the first step of the algorithms

fillGrid();

} // end constructor

// Function executed if the user presses the button "Maze"

private void mazeButtonActionPerformed(java.awt.event.ActionEvent evt) {

initializeGrid(true);

} // end mazeButtonActionPerformed()

// Creates a new clean grid or a new maze

private void initializeGrid(Boolean makeMaze) {

squareSize = 500/(rows > columns ? rows : columns);

// the maze must have an odd number of rows and columns

if (makeMaze && rows % 2 == 0) {

rows -= 1;

}

if (makeMaze && columns % 2 == 0) {

columns -= 1;

}

grid = new int[rows][columns];

robotStart = new Cell(rows-28,1);

targetPos = new Cell(27,columns-2);

dfs.setEnabled(true);

dfs.setSelected(true);

bfs.setEnabled(true);

aStar.setEnabled(true);

if (makeMaze) {

MyMaze maze = new MyMaze(rows/2,columns/2);

} else {

fillGrid();

}

} // end initializeGrid()

// Expands a node and creates his successors

private void expandNode(){

// The handling of the four algorithms

Cell current;

if (dfs.isSelected() || bfs.isSelected()) {

// Here is the 3rd step of the algorithms DFS and BFS

// 3. Remove the first state, Si, from OPEN SET ...

current = openSet.remove(0);

} else {

// Here is the 3rd step of the algorithms A\*

// 3. Remove the first state, Si, from OPEN SET,

current = openSet.remove(0);

}

// ... and add it to CLOSED SET.

closedSet.add(0,current);

// Update the color of the cell

grid[current.row][current.col] = CLOSED;

// If the selected node is the target ...

if (current.row == targetPos.row && current.col == targetPos.col) {

// ... then terminate etc

Cell last = targetPos;

last.prev = current.prev;

closedSet.add(last);

found = true;

return;

}

// Count nodes that have been expanded.

/\* Here is the 4rd step of the algorithms

4. Create the successors of Si, based on actions

that can be implemented on Si.

Each successor has a pointer to the Si, as its predecessor.

In the case of DFS and BFS algorithms, successors should not

belong neither to the OPEN SET nor the CLOSED SET.\*/

ArrayList<Cell> succesors;

succesors = createSuccesors(current, false);

// Here is the 5th step of the algorithms

// 5. For each successor of Si, ...

for (Cell cell: succesors){

// ... if we are running DFS ...

if (dfs.isSelected()) {

// ... add the successor at the beginning of the list OPEN SET

openSet.add(0, cell);

// Update the color of the cell

grid[cell.row][cell.col] = FRONTIER;

// ... if we are runnig BFS ...

} else if (bfs.isSelected()){

// ... add the successor at the end of the list OPEN SET

openSet.add(cell);

// Update the color of the cell

grid[cell.row][cell.col] = FRONTIER;

// ... if we are running A\* algorithm (step 5 of A\* algorithm) ...

} else if (aStar.isSelected()){

// ... calculate the value f(Sj) ...

int dxg = current.col-cell.col;

int dyg = current.row-cell.row;

int dxh = targetPos.col-cell.col;

int dyh = targetPos.row-cell.row;

// without diagonal movements

// calculate Manhattan distances

cell.g = current.g+Math.abs(dxg)+Math.abs(dyg);

cell.h = Math.abs(dxh)+Math.abs(dyh);

}

cell.f = cell.g+cell.h;

// ... If Sj is neither in the OPEN SET nor in the CLOSED SET states ...

int openIndex = isInList(openSet,cell);

int closedIndex = isInList(closedSet,cell);

if (openIndex == -1 && closedIndex == -1) {

// ... then add Sj in the OPEN SET ...

// ... evaluated as f(Sj)

openSet.add(cell);

// Update the color of the cell

grid[cell.row][cell.col] = FRONTIER;

// Else ...

} else {

// ... if already belongs to the OPEN SET, then ...

if (openIndex > -1){

// ... compare the new value assessment with the old one.

if (openSet.get(openIndex).f <= cell.f) {

// ... then eject the new node with state Sj.

} else {

// ... remove the element (Sj, old) from the list

// to which it belongs ...

openSet.remove(openIndex);

// ... and add the item (Sj, new) to the OPEN SET.

openSet.add(cell);

// Update the color of the cell

grid[cell.row][cell.col] = FRONTIER;

}

// ... if already belongs to the CLOSED SET, then ...

} else {

// ... compare the new value assessment with the old one.

if (closedSet.get(closedIndex).f <= cell.f) {

// ... then eject the new node with state Sj.

} else {

// ... remove the element (Sj, old) from the list

// to which it belongs ...

closedSet.remove(closedIndex);

// ... and add the item (Sj, new) to the OPEN SET.

openSet.add(cell);

// Update the color of the cell

grid[cell.row][cell.col] = FRONTIER;

}

}

}

}

} //end expandNode()

// Creates the successors of a state/cell

private ArrayList<Cell> createSuccesors(Cell current, boolean makeConnected){

int r = current.row;

int c = current.col;

// creates an empty list for the successors of the current cell.

ArrayList<Cell> temp = new ArrayList<>();

// If not at the topmost limit of the grid

// and the up-side cell is not an obstacle ...

if (r > 0 && grid[r-1][c] != OBST &&

// ... and (only in the case of not running the A\* )

// not already belongs neither to the OPEN SET nor to the CLOSED SET ...

((aStar.isSelected()) ? true :

isInList(openSet,new Cell(r-1,c)) == -1 &&

isInList(closedSet,new Cell(r-1,c)) == -1)) {

Cell cell = new Cell(r-1,c);

// ... update the pointer of the up-side cell so it points the current one ...

cell.prev = current;

// ... and add the up-side cell to the successors of the current one.

temp.add(cell);

}

// If not at the rightmost limit of the grid

// and the right-side cell is not an obstacle ...

if (c < columns-1 && grid[r][c+1] != OBST &&

// ... and (only in the case of not running the A\*)

// not already belongs neither to the OPEN SET nor to the CLOSED SET ...

((aStar.isSelected())? true :

isInList(openSet,new Cell(r,c+1)) == -1 &&

isInList(closedSet,new Cell(r,c+1)) == -1)) {

Cell cell = new Cell(r,c+1);

// ... update the pointer of the right-side cell so it points the current one ...

cell.prev = current;

// ... and add the right-side cell to the successors of the current one.

temp.add(cell);

}

// If not at the lowermost limit of the grid

// and the down-side cell is not an obstacle ...

if (r < rows-1 && grid[r+1][c] != OBST &&

// ... and (only in the case of not running the A\*)

// not already belongs neither to the OPEN SET nor to the CLOSED SET ...

((aStar.isSelected()) ? true :

isInList(openSet,new Cell(r+1,c)) == -1 &&

isInList(closedSet,new Cell(r+1,c)) == -1)) {

Cell cell = new Cell(r+1,c);

// ... update the pointer of the down-side cell so it points the current one ...

cell.prev = current;

// ... and add the down-side cell to the successors of the current one.

temp.add(cell);

}

// If not at the leftmost limit of the grid

// and the left-side cell is not an obstacle ...

if (c > 0 && grid[r][c-1] != OBST &&

// ... and (only in the case of not running the A\*)

// not already belongs neither to the OPEN SET nor to the CLOSED SET ...

((aStar.isSelected()) ? true :

isInList(openSet,new Cell(r,c-1)) == -1 &&

isInList(closedSet,new Cell(r,c-1)) == -1)) {

Cell cell = new Cell(r,c-1);

// ... update the pointer of the left-side cell so it points the current one ...

cell.prev = current;

// ... and add the left-side cell to the successors of the current one.

temp.add(cell);

}

if (dfs.isSelected()){

Collections.reverse(temp);

}

return temp;

} // end createSuccesors()

// Returns the index of the cell 'current' in the list 'list'

private int isInList(ArrayList<Cell> list, Cell current){

int index = -1;

for (int i = 0 ; i < list.size(); i++) {

if (current.row == list.get(i).row && current.col == list.get(i).col) {

index = i;

break;

}

}

return index;

} // end isInList()

/\*

Calculates the path from the target to the initial position

of the robot, counts the corresponding steps

and measures the distance traveled.

\*/

private void plotRoute(){

searching = false;

endOfSearch = true;

int index = isInList(closedSet,targetPos);

Cell cur = closedSet.get(index);

grid[cur.row][cur.col]= TARGET;

do {

cur = cur.prev;

grid[cur.row][cur.col] = ROUTE;

} while (!(cur.row == robotStart.row && cur.col == robotStart.col));

grid[robotStart.row][robotStart.col]=ROBOT;

} // end plotRoute()

/\*

\* Gives initial values ​​for the cells in the grid.

\* With the first click on button 'Clear' clears the data

\* of any search was performed (Frontier, Closed Set, Route)

\* and leaves intact the obstacles and the robot and target positions

\* in order to be able to run another algorithm

\* with the same data.

\* With the second click removes any obstacles also.

\*/

private void fillGrid() {

if (searching || endOfSearch){

for (int r = 0; r < rows; r++) {

for (int c = 0; c < columns; c++) {

if (grid[r][c] == FRONTIER || grid[r][c] == CLOSED || grid[r][c] == ROUTE) {

grid[r][c] = EMPTY;

}

if (grid[r][c] == ROBOT){

robotStart = new Cell(r,c);

}

if (grid[r][c] == TARGET){

targetPos = new Cell(r,c);

}

}

}

searching = false;

} else {

for (int r = 0; r < rows; r++) {

for (int c = 0; c < columns; c++) {

grid[r][c] = EMPTY;

}

}

robotStart = new Cell(rows-39,2);

targetPos = new Cell(37,columns-5);

}

if (aStar.isSelected()){

robotStart.g = 0;

robotStart.h = 0;

robotStart.f = 0;

}

found = false;

searching = false;

endOfSearch = false;

openSet.removeAll(openSet);

openSet.add(robotStart);

closedSet.removeAll(closedSet);

grid[targetPos.row][targetPos.col] = TARGET;

grid[robotStart.row][robotStart.col] = ROBOT;

timer.stop();

repaint();

} // end fillGrid()

@Override

public void paintComponent(Graphics g) {

super.paintComponent(g); // Fills the background color.

g.setColor(Color.black);

g.fillRect(10, 10, columns\*squareSize+1, rows\*squareSize+1);

for (int r = 0; r < rows; r++) {

for (int c = 0; c < columns; c++) {

if (grid[r][c] == EMPTY) {

g.setColor(Color.WHITE);

} else if (grid[r][c] == ROBOT) {

g.setColor(Color.red);

} else if (grid[r][c] == TARGET) {

g.setColor(Color.GREEN);

} else if (grid[r][c] == OBST) {

g.setColor(Color.black);

} else if (grid[r][c] == FRONTIER) {

g.setColor(Color.BLUE);

} else if (grid[r][c] == CLOSED) {

g.setColor(Color.cyan);

} else if (grid[r][c] == ROUTE) {

g.setColor(Color.orange);

}

g.fillRect(11 + c\*squareSize, 11 + r\*squareSize, squareSize - 1, squareSize - 1);

}

}

}

}

}//The end