// imported libraries

import java.awt.Graphics;

import java.awt.Color;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.awt.event.ItemEvent;

import java.awt.event.ItemListener;

import java.awt.event.MouseAdapter;

import java.awt.event.MouseEvent;

import java.awt.event.MouseListener;

import java.awt.event.MouseMotionListener;

import java.util.ArrayList;

import java.util.Random;

import javax.swing.JFrame;

import javax.swing.BorderFactory;

import javax.swing.ImageIcon;

import javax.swing.JButton;

import javax.swing.JLabel;

import javax.swing.JOptionPane;

import javax.swing.JPanel;

import javax.swing.JSlider;

import javax.swing.border.Border;

import javax.swing.border.EtchedBorder;

import javax.swing.event.ChangeEvent;

import javax.swing.event.ChangeListener;

import javax.swing.JComboBox;

public class PathFinding {

//FRAME

JFrame frame;

//GENERAL VARIABLES

private int cells = 20;

private int delay = 30;

private double dense = .5;

private double density = (cells\*cells)\*.5;

private int startx = -1;

private int starty = -1;

private int finishx = -1;

private int finishy = -1;

private int tool = 0;

private int checks = 0;

private int length = 0;

private int curAlg = 0;

private int WIDTH = 850;

private final int HEIGHT = 650;

private final int MSIZE = 600;

private int CSIZE = MSIZE/cells;

//UTIL ARRAYS

private String[] algorithms = {"Dijkstra","A\*"};

private String[] tools = {"Start","Finish","Wall", "Eraser"};

//BOOLEANS

private boolean solving = false;

//UTIL

Node[][] map;

Algorithm Alg = new Algorithm();

Random r = new Random();

//SLIDERS

JSlider size = new JSlider(1,5,2);

JSlider speed = new JSlider(0,500,delay);

JSlider obstacles = new JSlider(1,100,50);

//LABELS

JLabel algL = new JLabel("Algorithms");

JLabel toolL = new JLabel("Toolbox");

JLabel sizeL = new JLabel("Size:");

JLabel cellsL = new JLabel(cells+"x"+cells);

JLabel delayL = new JLabel("Delay:");

JLabel msL = new JLabel(delay+"ms");

JLabel obstacleL = new JLabel("Dens:");

JLabel densityL = new JLabel(obstacles.getValue()+"%");

JLabel checkL = new JLabel("Checks: "+checks);

JLabel lengthL = new JLabel("Path Length: "+length);

//BUTTONS

JButton searchB = new JButton("Start Search");

JButton resetB = new JButton("Reset");

JButton genMapB = new JButton("Generate Map");

JButton clearMapB = new JButton("Clear Map");

JButton creditB = new JButton("Credit");

//DROP DOWN

JComboBox algorithmsBx = new JComboBox(algorithms);

JComboBox toolBx = new JComboBox(tools);

//PANELS

JPanel toolP = new JPanel();

//CANVAS

Map canvas;

//BORDER

Border loweredetched = BorderFactory.createEtchedBorder(EtchedBorder.LOWERED);

public static void main(String[] args) { //MAIN METHOD

new PathFinding();

}

public PathFinding() { //CONSTRUCTOR

clearMap();

initialize();

}

public void generateMap() { //GENERATE MAP

clearMap(); //CREATE CLEAR MAP TO START

for(int i = 0; i < density; i++) {

Node current;

do {

int x = r.nextInt(cells);

int y = r.nextInt(cells);

current = map[x][y]; //FIND A RANDOM NODE IN THE GRID

} while(current.getType()==2); //IF IT IS ALREADY A WALL, FIND A NEW ONE

current.setType(2); //SET NODE TO BE A WALL

}

}

public void clearMap() { //CLEAR MAP

finishx = -1; //RESET THE START AND FINISH

finishy = -1;

startx = -1;

starty = -1;

map = new Node[cells][cells]; //CREATE NEW MAP OF NODES

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

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

map[x][y] = new Node(3,x,y); //SET ALL NODES TO EMPTY

}

}

reset(); //RESET SOME VARIABLES

}

public void resetMap() { //RESET MAP

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

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

Node current = map[x][y];

if(current.getType() == 4 || current.getType() == 5) //CHECK TO SEE IF CURRENT NODE IS EITHER CHECKED OR FINAL PATH

map[x][y] = new Node(3,x,y); //RESET IT TO AN EMPTY NODE

}

}

if(startx > -1 && starty > -1) { //RESET THE START AND FINISH

map[startx][starty] = new Node(0,startx,starty);

map[startx][starty].setHops(0);

}

if(finishx > -1 && finishy > -1)

map[finishx][finishy] = new Node(1,finishx,finishy);

reset(); //RESET SOME VARIABLES

}

private void initialize() { //INITIALIZE THE GUI ELEMENTS

frame = new JFrame();

frame.setVisible(true);

frame.setResizable(false);

frame.setSize(WIDTH,HEIGHT);

frame.setTitle("Path Finding");

frame.setLocationRelativeTo(null);

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

frame.getContentPane().setLayout(null);

toolP.setBorder(BorderFactory.createTitledBorder(loweredetched,"Controls"));

int space = 25;

int buff = 45;

toolP.setLayout(null);

toolP.setBounds(10,10,210,600);

searchB.setBounds(40,space, 120, 25);

toolP.add(searchB);

space+=buff;

resetB.setBounds(40,space,120,25);

toolP.add(resetB);

space+=buff;

genMapB.setBounds(40,space, 120, 25);

toolP.add(genMapB);

space+=buff;

clearMapB.setBounds(40,space, 120, 25);

toolP.add(clearMapB);

space+=40;

algL.setBounds(40,space,120,25);

toolP.add(algL);

space+=25;

algorithmsBx.setBounds(40,space, 120, 25);

toolP.add(algorithmsBx);

space+=40;

toolL.setBounds(40,space,120,25);

toolP.add(toolL);

space+=25;

toolBx.setBounds(40,space,120,25);

toolP.add(toolBx);

space+=buff;

sizeL.setBounds(15,space,40,25);

toolP.add(sizeL);

size.setMajorTickSpacing(10);

size.setBounds(50,space,100,25);

toolP.add(size);

cellsL.setBounds(160,space,40,25);

toolP.add(cellsL);

space+=buff;

delayL.setBounds(15,space,50,25);

toolP.add(delayL);

speed.setMajorTickSpacing(5);

speed.setBounds(50,space,100,25);

toolP.add(speed);

msL.setBounds(160,space,40,25);

toolP.add(msL);

space+=buff;

obstacleL.setBounds(15,space,100,25);

toolP.add(obstacleL);

obstacles.setMajorTickSpacing(5);

obstacles.setBounds(50,space,100,25);

toolP.add(obstacles);

densityL.setBounds(160,space,100,25);

toolP.add(densityL);

space+=buff;

checkL.setBounds(15,space,100,25);

toolP.add(checkL);

space+=buff;

lengthL.setBounds(15,space,100,25);

toolP.add(lengthL);

space+=buff;

creditB.setBounds(40, space, 120, 25);

toolP.add(creditB);

frame.getContentPane().add(toolP);

canvas = new Map();

canvas.setBounds(230, 10, MSIZE+1, MSIZE+1);

frame.getContentPane().add(canvas);

searchB.addActionListener(new ActionListener() { //ACTION LISTENERS

@Override

public void actionPerformed(ActionEvent e) {

reset();

if((startx > -1 && starty > -1) && (finishx > -1 && finishy > -1))

solving = true;

}

});

resetB.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

resetMap();

Update();

}

});

genMapB.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

generateMap();

Update();

}

});

clearMapB.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

clearMap();

Update();

}

});

algorithmsBx.addItemListener(new ItemListener() {

@Override

public void itemStateChanged(ItemEvent e) {

curAlg = algorithmsBx.getSelectedIndex();

Update();

}

});

toolBx.addItemListener(new ItemListener() {

@Override

public void itemStateChanged(ItemEvent e) {

tool = toolBx.getSelectedIndex();

}

});

size.addChangeListener(new ChangeListener() {

@Override

public void stateChanged(ChangeEvent e) {

cells = size.getValue()\*10;

clearMap();

reset();

Update();

}

});

speed.addChangeListener(new ChangeListener() {

@Override

public void stateChanged(ChangeEvent e) {

delay = speed.getValue();

Update();

}

});

obstacles.addChangeListener(new ChangeListener() {

@Override

public void stateChanged(ChangeEvent e) {

dense = (double)obstacles.getValue()/100;

Update();

}

});

creditB.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e)

{

JOptionPane.showMessageDialog(frame, " Path Finding\n"

+ " Build Date: May 10, 2023 ", "Credit", JOptionPane.PLAIN\_MESSAGE, new ImageIcon(""));

}

});

startSearch(); //START STATE

}

public void startSearch() { //START STATE

if(solving) {

switch(curAlg) {

case 0:

Alg.Dijkstra();

break;

case 1:

Alg.AStar();

break;

}

}

pause(); //PAUSE STATE

}

public void pause() { //PAUSE STATE

int i = 0;

while(!solving) {

i++;

if(i > 500)

i = 0;

try {

Thread.sleep(1);

} catch(Exception e) {}

}

startSearch(); //START STATE

}

public void Update() { //UPDATE ELEMENTS OF THE GUI

density = (cells\*cells)\*dense;

CSIZE = MSIZE/cells;

canvas.repaint();

cellsL.setText(cells+"x"+cells);

msL.setText(delay+"ms");

lengthL.setText("Path Length: "+length);

densityL.setText(obstacles.getValue()+"%");

checkL.setText("Checks: "+checks);

}

public void reset() { //RESET METHOD

solving = false;

length = 0;

checks = 0;

}

public void delay() { //DELAY METHOD

try {

Thread.sleep(delay);

} catch(Exception e) {}

}

class Map extends JPanel implements MouseListener, MouseMotionListener{ //MAP CLASS

public Map() {

addMouseListener(this);

addMouseMotionListener(this);

}

public void paintComponent(Graphics g) { //REPAINT

super.paintComponent(g);

for(int x = 0; x < cells; x++) { //PAINT EACH NODE IN THE GRID

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

switch(map[x][y].getType()) {

case 0:

g.setColor(Color.GREEN);

break;

case 1:

g.setColor(Color.RED);

break;

case 2:

g.setColor(Color.BLACK);

break;

case 3:

g.setColor(Color.WHITE);

break;

case 4:

g.setColor(Color.CYAN);

break;

case 5:

g.setColor(Color.YELLOW);

break;

}

g.fillRect(x\*CSIZE,y\*CSIZE,CSIZE,CSIZE);

g.setColor(Color.BLACK);

g.drawRect(x\*CSIZE,y\*CSIZE,CSIZE,CSIZE);

}

}

}

@Override

public void mouseDragged(MouseEvent e) {

try {

int x = e.getX()/CSIZE;

int y = e.getY()/CSIZE;

Node current = map[x][y];

if((tool == 2 || tool == 3) && (current.getType() != 0 && current.getType() != 1))

current.setType(tool);

Update();

} catch(Exception z) {}

}

@Override // indicates that child class method is over-writing its parent class method

public void mouseMoved(MouseEvent e) {}

@Override

public void mouseClicked(MouseEvent e) {}

@Override

public void mouseEntered(MouseEvent e) {}

@Override

public void mouseExited(MouseEvent e) {}

@Override

public void mousePressed(MouseEvent e) {

resetMap(); //RESET THE MAP WHENEVER CLICKED

try {

int x = e.getX()/CSIZE; //GET THE X AND Y OF THE MOUSE CLICK IN RELATION TO THE SIZE OF THE GRID

int y = e.getY()/CSIZE;

Node current = map[x][y];

switch(tool ) {

case 0: { //START NODE

if(current.getType()!=2) { //IF NOT WALL

if(startx > -1 && starty > -1) { //IF START EXISTS SET IT TO EMPTY

map[startx][starty].setType(3);

map[startx][starty].setHops(-1);

}

current.setHops(0);

startx = x; //SET THE START X AND Y

starty = y;

current.setType(0); //SET THE NODE CLICKED TO BE START

}

break;

}

case 1: { //FINISH NODE

if(current.getType()!=2) { //IF NOT WALL

if(finishx > -1 && finishy > -1) //IF FINISH EXISTS SET IT TO EMPTY

map[finishx][finishy].setType(3);

finishx = x; //SET THE FINISH X AND Y

finishy = y;

current.setType(1); //SET THE NODE CLICKED TO BE FINISH

}

break;

}

default:

if(current.getType() != 0 && current.getType() != 1)

current.setType(tool);

break;

}

Update();

} catch(Exception z) {} //EXCEPTION HANDLER

}

@Override

public void mouseReleased(MouseEvent e) {}

}

class Algorithm { //ALGORITHM CLASS

//DIJKSTRA WORKS BY PROPAGATING OUTWARDS UNTIL IT FINDS THE FINISH AND THEN WORKING ITS WAY BACK TO GET THE PATH

//IT USES A PRIORITY QUE TO KEEP TRACK OF NODES THAT IT NEEDS TO EXPLORE

//EACH NODE IN THE PRIORITY QUE IS EXPLORED AND ALL OF ITS NEIGHBORS ARE ADDED TO THE QUE

//ONCE A NODE IS EXLPORED IT IS DELETED FROM THE QUE

//AN ARRAYLIST IS USED TO REPRESENT THE PRIORITY QUE

//A SEPERATE ARRAYLIST IS RETURNED FROM A METHOD THAT EXPLORES A NODES NEIGHBORS

//THIS ARRAYLIST CONTAINS ALL THE NODES THAT WERE EXPLORED, IT IS THEN ADDED TO THE QUE

//A HOPS VARIABLE IN EACH NODE REPRESENTS THE NUMBER OF NODES TRAVELED FROM THE START

public void Dijkstra() {

ArrayList<Node> priority = new ArrayList<Node>(); //CREATE A PRIORITY QUE

priority.add(map[startx][starty]); //ADD THE START TO THE QUE

while(solving) {

if(priority.size() <= 0) { //IF THE QUE IS 0 THEN NO PATH CAN BE FOUND

solving = false;

break;

}

int hops = priority.get(0).getHops()+1; //INCREMENT THE HOPS VARIABLE

ArrayList<Node> explored = exploreNeighbors(priority.get(0), hops); //CREATE AN ARRAYLIST OF NODES THAT WERE EXPLORED

if(explored.size() > 0) {

priority.remove(0); //REMOVE THE NODE FROM THE QUE

priority.addAll(explored); //ADD ALL THE NEW NODES TO THE QUE

Update();

delay();

} else { //IF NO NODES WERE EXPLORED THEN JUST REMOVE THE NODE FROM THE QUE

priority.remove(0);

}

}

}

//A STAR WORKS ESSENTIALLY THE SAME AS DIJKSTRA CREATING A PRIORITY QUE AND PROPAGATING OUTWARDS UNTIL IT FINDS THE END

//HOWEVER ASTAR BUILDS IN A HEURISTIC OF DISTANCE FROM ANY NODE TO THE FINISH

//THIS MEANS THAT NODES THAT ARE CLOSER TO THE FINISH WILL BE EXPLORED FIRST

//THIS HEURISTIC IS BUILT IN BY SORTING THE QUE ACCORDING TO HOPS PLUS DISTANCE UNTIL THE FINISH

public void AStar() {

ArrayList<Node> priority = new ArrayList<Node>();

priority.add(map[startx][starty]);

while(solving) {

if(priority.size() <= 0) {

solving = false;

break;

}

int hops = priority.get(0).getHops()+1;

ArrayList<Node> explored = exploreNeighbors(priority.get(0),hops);

if(explored.size() > 0) {

priority.remove(0);

priority.addAll(explored);

Update();

delay();

} else {

priority.remove(0);

}

sortQue(priority); //SORT THE PRIORITY QUE

}

}

public ArrayList<Node> sortQue(ArrayList<Node> sort) { //SORT PRIORITY QUE

int c = 0;

while(c < sort.size()) {

int sm = c;

for(int i = c+1; i < sort.size(); i++) {

if(sort.get(i).getEuclidDist()+sort.get(i).getHops() < sort.get(sm).getEuclidDist()+sort.get(sm).getHops())

sm = i;

}

if(c != sm) {

Node temp = sort.get(c);

sort.set(c, sort.get(sm));

sort.set(sm, temp);

}

c++;

}

return sort;

}

public ArrayList<Node> exploreNeighbors(Node current, int hops) { //EXPLORE NEIGHBORS

ArrayList<Node> explored = new ArrayList<Node>(); //LIST OF NODES THAT HAVE BEEN EXPLORED

for(int a = -1; a <= 1; a++) {

for(int b = -1; b <= 1; b++) {

int xbound = current.getX()+a;

int ybound = current.getY()+b;

if((xbound > -1 && xbound < cells) && (ybound > -1 && ybound < cells)) { //MAKES SURE THE NODE IS NOT OUTSIDE THE GRID

Node neighbor = map[xbound][ybound];

if((neighbor.getHops()==-1 || neighbor.getHops() > hops) && neighbor.getType()!=2) { //CHECKS IF THE NODE IS NOT A WALL AND THAT IT HAS NOT BEEN EXPLORED

explore(neighbor, current.getX(), current.getY(), hops); //EXPLORE THE NODE

explored.add(neighbor); //ADD THE NODE TO THE LIST

}

}

}

}

return explored;

}

public void explore(Node current, int lastx, int lasty, int hops) { //EXPLORE A NODE

if(current.getType()!=0 && current.getType() != 1) //CHECK THAT THE NODE IS NOT THE START OR FINISH

current.setType(4); //SET IT TO EXPLORED

current.setLastNode(lastx, lasty); //KEEP TRACK OF THE NODE THAT THIS NODE IS EXPLORED FROM

current.setHops(hops); //SET THE HOPS FROM THE START

checks++;

if(current.getType() == 1) { //IF THE NODE IS THE FINISH THEN BACKTRACK TO GET THE PATH

backtrack(current.getLastX(), current.getLastY(),hops);

}

}

public void backtrack(int lx, int ly, int hops) { //BACKTRACK

length = hops;

while(hops > 1) { //BACKTRACK FROM THE END OF THE PATH TO THE START

Node current = map[lx][ly];

current.setType(5);

lx = current.getLastX();

ly = current.getLastY();

hops--;

}

solving = false;

}

}

class Node {

// 0 = start, 1 = finish, 2 = wall, 3 = empty, 4 = checked, 5 = finalpath

private int cellType = 0;

private int hops;

private int x;

private int y;

private int lastX;

private int lastY;

private double dToEnd = 0;

public Node(int type, int x, int y) { //CONSTRUCTOR

cellType = type;

this.x = x;

this.y = y;

hops = -1;

}

public double getEuclidDist() { //CALCULATES THE EUCLIDIAN DISTANCE TO THE FINISH NODE

int xdif = Math.abs(x-finishx);

int ydif = Math.abs(y-finishy);

dToEnd = Math.sqrt((xdif\*xdif)+(ydif\*ydif));

return dToEnd;

}

public int getX() {return x;} //GET METHODS

public int getY() {return y;}

public int getLastX() {return lastX;}

public int getLastY() {return lastY;}

public int getType() {return cellType;}

public int getHops() {return hops;}

public void setType(int type) {cellType = type;} //SET METHODS

public void setLastNode(int x, int y) {lastX = x; lastY = y;}

public void setHops(int hops) {this.hops = hops;}

}

}