**Федеральное агентство связи**

**Государственное бюджетное образовательное учреждение высшего**

**образование**

**Ордена Трудового Красного Знамени**

**«Московский технический университет связи и информатики»**

**Кафедра «МКиИТ»**

**дисциплина «Объектно-ориентированное программирование»**

Отчет по Лабораторной работе №3: Алгоритм A\* («A star»)

Подготовила студентка

группы БВТ1901: Нкурикийе Х

Проверил: Мосева М.С.

Москва 2020

//1

package com.hafi.lab3;

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

/\*\*

\* A simple Swing application to demonstrate the A\* pathfinding algorithm. The

\* user is presented with a map, containing a start and end location. The user

\* can draw or clear obstacles on the map, and then press a button to compute a

\* path from start to end using the A\* pathfinding algorithm. If a path is

\* found, it is displayed in green.

\*\*/

public class AstarApp {

/\*\* The number of grid cells in the X direction. \*\*/

private int width;

/\*\* The number of grid cells in the Y direction. \*\*/

private int height;

/\*\* The location where the path starts from. \*\*/

private Location startLoc;

/\*\* The location where the path is supposed to finish. \*\*/

private Location finishLoc;

/\*\*

\* This is a 2D array of UI components that provide display and manipulation

\* of the cells in the map.

\*\*\*/

private JMapCell[][] mapCells;

/\*\*

\* This inner class handles mouse events in the main grid of map cells, by

\* modifying the cells based on the mouse button state and the initial edit

\* that was performed.

\*\*/

private class MapCellHandler implements MouseListener

{

/\*\*

\* This value will be true if a mouse button has been pressed and we are

\* currently in the midst of a modification operation.

\*\*/

private boolean modifying;

/\*\*

\* This value records whether we are making cells passable or

\* impassable. Which it is depends on the original state of the cell

\* that the operation was started within.

\*\*/

private boolean makePassable;

/\*\* Initiates the modification operation. \*\*/

public void mousePressed(MouseEvent e)

{

modifying = true;

JMapCell cell = (JMapCell) e.getSource();

// If the current cell is passable then we are making them

// impassable; if it's impassable then we are making them passable.

makePassable = !cell.isPassable();

cell.setPassable(makePassable);

}

/\*\* Ends the modification operation. \*\*/

public void mouseReleased(MouseEvent e)

{

modifying = false;

}

/\*\*

\* If the mouse has been pressed, this continues the modification

\* operation into the new cell.

\*\*/

public void mouseEntered(MouseEvent e)

{

if (modifying)

{

JMapCell cell = (JMapCell) e.getSource();

cell.setPassable(makePassable);

}

}

/\*\* Not needed for this handler. \*\*/

public void mouseExited(MouseEvent e)

{

// This one we ignore.

}

/\*\* Not needed for this handler. \*\*/

public void mouseClicked(MouseEvent e)

{

// And this one too.

}

}

/\*\*

\* Creates a new instance of AStarApp with the specified map width and

\* height.

\*\*/

public AstarApp(int w, int h) {

if (w <= 0)

throw new IllegalArgumentException("w must be > 0; got " + w);

if (h <= 0)

throw new IllegalArgumentException("h must be > 0; got " + h);

width = w;

height = h;

startLoc = new Location(2, h / 2);

finishLoc = new Location(w - 3, h / 2);

}

/\*\*

\* Simple helper method to set up the Swing user interface. This is called

\* from the Swing event-handler thread to be threadsafe.

\*\*/

private void initGUI()

{

JFrame frame = new JFrame("Pathfinder");

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

Container contentPane = frame.getContentPane();

contentPane.setLayout(new BorderLayout());

// Use GridBagLayout because it actually respects the preferred size

// specified by the components it lays out.

GridBagLayout gbLayout = new GridBagLayout();

GridBagConstraints gbConstraints = new GridBagConstraints();

gbConstraints.fill = GridBagConstraints.BOTH;

gbConstraints.weightx = 1;

gbConstraints.weighty = 1;

gbConstraints.insets.set(0, 0, 1, 1);

JPanel mapPanel = new JPanel(gbLayout);

mapPanel.setBackground(Color.GRAY);

mapCells = new JMapCell[width][height];

MapCellHandler cellHandler = new MapCellHandler();

for (int y = 0; y < height; y++)

{

for (int x = 0; x < width; x++)

{

mapCells[x][y] = new JMapCell();

gbConstraints.gridx = x;

gbConstraints.gridy = y;

gbLayout.setConstraints(mapCells[x][y], gbConstraints);

mapPanel.add(mapCells[x][y]);

mapCells[x][y].addMouseListener(cellHandler);

}

}

contentPane.add(mapPanel, BorderLayout.CENTER);

JButton findPathButton = new JButton("Find Path");

findPathButton.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) { findAndShowPath(); }

});

contentPane.add(findPathButton, BorderLayout.SOUTH);

frame.pack();

frame.setVisible(true);

mapCells[startLoc.xCoord][startLoc.yCoord].setEndpoint(true);

mapCells[finishLoc.xCoord][finishLoc.yCoord].setEndpoint(true);

}

/\*\* Kicks off the application. Called from the {@link #main} method. \*\*/

private void start()

{

SwingUtilities.invokeLater(new Runnable() {

public void run() { initGUI(); }

});

}

/\*\*

\* This helper method attempts to compute a path using the current map

\* state. The implementation is rather slow; a new {@link Map2D} object is

\* created, and initialized from the current application state. Then the A\*

\* pathfinder is called, and if a path is found, the display is updated to

\* show the path that was found. (A better solution would use the Model

\* View Controller design pattern.)

\*\*/

private void findAndShowPath()

{

// Create a Map2D object containing the current state of the user input.

Map2D map = new Map2D(width, height);

map.setStart(startLoc);

map.setFinish(finishLoc);

for (int y = 0; y < height; y++)

{

for (int x = 0; x < width; x++)

{

mapCells[x][y].setPath(false);

if (mapCells[x][y].isPassable())

map.setCellValue(x, y, 0);

else

map.setCellValue(x, y, Integer.MAX\_VALUE);

}

}

// Try to compute a path. If one can be computed, mark all cells in the

// path.

Waypoint wp = AstarPathfinder.computePath(map);

while (wp != null)

{

Location loc = wp.getLocation();

mapCells[loc.xCoord][loc.yCoord].setPath(true);

wp = wp.getPrevious();

}

}

/\*\*

\* Entry-point for the application. No command-line arguments are

\* recognized at this time.

\*\*/

public static void main(String[] args) {

AstarApp app = new AstarApp(40, 30);

app.start();

}

}

//2

package com.hafi.lab3;

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

/\*\*

\* A simple Swing application to demonstrate the A\* pathfinding algorithm. The

\* user is presented with a map, containing a start and end location. The user

\* can draw or clear obstacles on the map, and then press a button to compute a

\* path from start to end using the A\* pathfinding algorithm. If a path is

\* found, it is displayed in green.

\*\*/

public class AstarApp {

/\*\* The number of grid cells in the X direction. \*\*/

private int width;

/\*\* The number of grid cells in the Y direction. \*\*/

private int height;

/\*\* The location where the path starts from. \*\*/

private Location startLoc;

/\*\* The location where the path is supposed to finish. \*\*/

private Location finishLoc;

/\*\*

\* This is a 2D array of UI components that provide display and manipulation

\* of the cells in the map.

\*\*\*/

private JMapCell[][] mapCells;

/\*\*

\* This inner class handles mouse events in the main grid of map cells, by

\* modifying the cells based on the mouse button state and the initial edit

\* that was performed.

\*\*/

private class MapCellHandler implements MouseListener

{

/\*\*

\* This value will be true if a mouse button has been pressed and we are

\* currently in the midst of a modification operation.

\*\*/

private boolean modifying;

/\*\*

\* This value records whether we are making cells passable or

\* impassable. Which it is depends on the original state of the cell

\* that the operation was started within.

\*\*/

private boolean makePassable;

/\*\* Initiates the modification operation. \*\*/

public void mousePressed(MouseEvent e)

{

modifying = true;

JMapCell cell = (JMapCell) e.getSource();

// If the current cell is passable then we are making them

// impassable; if it's impassable then we are making them passable.

makePassable = !cell.isPassable();

cell.setPassable(makePassable);

}

/\*\* Ends the modification operation. \*\*/

public void mouseReleased(MouseEvent e)

{

modifying = false;

}

/\*\*

\* If the mouse has been pressed, this continues the modification

\* operation into the new cell.

\*\*/

public void mouseEntered(MouseEvent e)

{

if (modifying)

{

JMapCell cell = (JMapCell) e.getSource();

cell.setPassable(makePassable);

}

}

/\*\* Not needed for this handler. \*\*/

public void mouseExited(MouseEvent e)

{

// This one we ignore.

}

/\*\* Not needed for this handler. \*\*/

public void mouseClicked(MouseEvent e)

{

// And this one too.

}

}

/\*\*

\* Creates a new instance of AStarApp with the specified map width and

\* height.

\*\*/

public AstarApp(int w, int h) {

if (w <= 0)

throw new IllegalArgumentException("w must be > 0; got " + w);

if (h <= 0)

throw new IllegalArgumentException("h must be > 0; got " + h);

width = w;

height = h;

startLoc = new Location(2, h / 2);

finishLoc = new Location(w - 3, h / 2);

}

/\*\*

\* Simple helper method to set up the Swing user interface. This is called

\* from the Swing event-handler thread to be threadsafe.

\*\*/

private void initGUI()

{

JFrame frame = new JFrame("Pathfinder");

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

Container contentPane = frame.getContentPane();

contentPane.setLayout(new BorderLayout());

// Use GridBagLayout because it actually respects the preferred size

// specified by the components it lays out.

GridBagLayout gbLayout = new GridBagLayout();

GridBagConstraints gbConstraints = new GridBagConstraints();

gbConstraints.fill = GridBagConstraints.BOTH;

gbConstraints.weightx = 1;

gbConstraints.weighty = 1;

gbConstraints.insets.set(0, 0, 1, 1);

JPanel mapPanel = new JPanel(gbLayout);

mapPanel.setBackground(Color.GRAY);

mapCells = new JMapCell[width][height];

MapCellHandler cellHandler = new MapCellHandler();

for (int y = 0; y < height; y++)

{

for (int x = 0; x < width; x++)

{

mapCells[x][y] = new JMapCell();

gbConstraints.gridx = x;

gbConstraints.gridy = y;

gbLayout.setConstraints(mapCells[x][y], gbConstraints);

mapPanel.add(mapCells[x][y]);

mapCells[x][y].addMouseListener(cellHandler);

}

}

contentPane.add(mapPanel, BorderLayout.CENTER);

JButton findPathButton = new JButton("Find Path");

findPathButton.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) { findAndShowPath(); }

});

contentPane.add(findPathButton, BorderLayout.SOUTH);

frame.pack();

frame.setVisible(true);

mapCells[startLoc.xCoord][startLoc.yCoord].setEndpoint(true);

mapCells[finishLoc.xCoord][finishLoc.yCoord].setEndpoint(true);

}

/\*\* Kicks off the application. Called from the {@link #main} method. \*\*/

private void start()

{

SwingUtilities.invokeLater(new Runnable() {

public void run() { initGUI(); }

});

}

/\*\*

\* This helper method attempts to compute a path using the current map

\* state. The implementation is rather slow; a new {@link Map2D} object is

\* created, and initialized from the current application state. Then the A\*

\* pathfinder is called, and if a path is found, the display is updated to

\* show the path that was found. (A better solution would use the Model

\* View Controller design pattern.)

\*\*/

private void findAndShowPath()

{

// Create a Map2D object containing the current state of the user input.

Map2D map = new Map2D(width, height);

map.setStart(startLoc);

map.setFinish(finishLoc);

for (int y = 0; y < height; y++)

{

for (int x = 0; x < width; x++)

{

mapCells[x][y].setPath(false);

if (mapCells[x][y].isPassable())

map.setCellValue(x, y, 0);

else

map.setCellValue(x, y, Integer.MAX\_VALUE);

}

}

// Try to compute a path. If one can be computed, mark all cells in the

// path.

Waypoint wp = AstarPathfinder.computePath(map);

while (wp != null)

{

Location loc = wp.getLocation();

mapCells[loc.xCoord][loc.yCoord].setPath(true);

wp = wp.getPrevious();

}

}

/\*\*

\* Entry-point for the application. No command-line arguments are

\* recognized at this time.

\*\*/

public static void main(String[] args) {

AstarApp app = new AstarApp(40, 30);

app.start();

}

}

//3

package com.hafi.lab3;

import java.util.HashMap;

import java.util.Map;

public class AstarState {

/\*\* This is a reference to the map that the A\* algorithm is navigating. \*\*/

private Map2D map;

private HashMap<Location, Waypoint> opened = new HashMap();

private HashMap<Location, Waypoint> closed = new HashMap();

public AstarState(Map2D map)

{

if (map == null)

throw new NullPointerException("map cannot be null");

this.map = map;

}

/\*\* Returns the map that the A\* pathfinder is navigating. \*\*/

public Map2D getMap()

{

return map;

}

/\*\*

\* This method scans through all open waypoints, and returns the waypoint

\* with the minimum total cost. If there are no open waypoints, this method

\* returns <code>null</code>.

\*\*/

public Waypoint getMinOpenWaypoint()

{

Waypoint min = null;

for (Map.Entry<Location, Waypoint> entry : opened.entrySet()) {

if (min == null) {

min = entry.getValue();

} else {

if (min.getTotalCost() > entry.getValue().getTotalCost() ) {

min = entry.getValue();

}

}

}

return min;

}

/\*\*

\* This method adds a waypoint to (or potentially updates a waypoint already

\* in) the "open waypoints" collection. If there is not already an open

\* waypoint at the new waypoint's location then the new waypoint is simply

\* added to the collection. However, if there is already a waypoint at the

\* new waypoint's location, the new waypoint replaces the old one <em>only

\* if</em> the new waypoint's "previous cost" value is less than the current

\* waypoint's "previous cost" value.

\*\*/

public boolean addOpenWaypoint(Waypoint newWP)

{

if (!opened.containsKey(newWP.getLocation())) {

opened.put(newWP.getLocation(), newWP);

return true;

} else {

Location location = newWP.getLocation();

for (Map.Entry<Location, Waypoint> entry : opened.entrySet()) {

if (entry.getKey().equals(location) && entry.getValue().getRemainingCost() > newWP.getRemainingCost()) {

opened.put(location, newWP);

return true;

} else {

return false;

}

}

}

return false;

}

/\*\* Returns the current number of open waypoints. \*\*/

public int numOpenWaypoints()

{

return opened.size();

}

/\*\*

\* This method moves the waypoint at the specified location from the

\* open list to the closed list.

\*\*/

public void closeWaypoint(Location loc)

{

closed.put(loc, opened.remove(loc));

}

/\*\*

\* Returns true if the collection of closed waypoints contains a waypoint

\* for the specified location.

\*\*/

public boolean isLocationClosed(Location loc)

{

return closed.containsKey(loc);

}

}

//4

package com.hafi.lab3;

import java.awt.\*;

import javax.swing.\*;

import javax.swing.border.\*;

/\*\*

\* This class is a custom Swing component for representing a single map cell in

\* a 2D map. The cell has several different kinds of state, but the most basic

\* state is whether the cell is passable or not.

\*/

class JMapCell extends JComponent

{

private static final Dimension CELL\_SIZE = new Dimension(12, 12);

/\*\* True indicates that the cell is an endpoint, either start or finish. \*\*/

boolean endpoint = false;

/\*\* True indicates that the cell is passable; false means it is not. \*\*/

boolean passable = true;

/\*\*

\* True indicates that this cell is part of the path between start and end.

\*\*/

boolean path = false;

/\*\*

\* Construct a new map cell with the specified "passability." An input of

\* true means the cell is passable.

\*\*/

public JMapCell(boolean pass)

{

// Set the preferred cell size, to drive the initial window size.

setPreferredSize(CELL\_SIZE);

setPassable(pass);

}

/\*\* Construct a new map cell, which is passable by default. \*\*/

public JMapCell()

{

// Call the other constructor, specifying true for "passable".

this(true);

}

/\*\* Marks this cell as either being the starting or the ending cell. \*\*/

public void setEndpoint(boolean end)

{

endpoint = end;

updateAppearance();

}

/\*\*

\* Sets this cell to be passable or not passable. An input of true marks

\* the cell as passable; an input of false marks it as not passable.

\*\*/

public void setPassable(boolean pass)

{

passable = pass;

updateAppearance();

}

/\*\* Returns true if this cell is passable, or false otherwise. \*\*/

public boolean isPassable()

{

return passable;

}

/\*\* Toggles the current "passable" state of the map cell. \*\*/

public void togglePassable()

{

setPassable(!isPassable());

}

/\*\* Marks this cell as part of the path discovered by the A\* algorithm. \*\*/

public void setPath(boolean path)

{

this.path = path;

updateAppearance();

}

/\*\*

\* This helper method updates the background color to match the current

\* internal state of the cell.

\*\*/

private void updateAppearance()

{

if (passable)

{

// Passable cell. Indicate its state with a border.

setBackground(Color.WHITE);

if (endpoint)

setBackground(Color.CYAN);

else if (path)

setBackground(Color.GREEN);

}

else

{

// Impassable cell. Make it all red.

setBackground(Color.RED);

}

}

/\*\*

\* Implementation of the paint method to draw the background color into the

\* map cell.

\*\*/

protected void paintComponent(Graphics g)

{

g.setColor(getBackground());

g.fillRect(0, 0, getWidth(), getHeight());

}

}

//5

package com.hafi.lab3;

public class Location {

/\*\* X coordinate of this location. \*\*/

public int xCoord;

/\*\* Y coordinate of this location. \*\*/

public int yCoord;

/\*\* Creates a new location with the specified integer coordinates. \*\*/

public Location(int x, int y)

{

xCoord = x;

yCoord = y;

}

/\*\* Creates a new location with coordinates (0, 0). \*\*/

public Location()

{

this(0, 0);

}

@Override

public boolean equals(Object obj) {

if (obj instanceof Location) {

Location location = (Location) obj;

return xCoord == location.xCoord && yCoord == location.yCoord;

}

return false;

}

@Override

public int hashCode() {

int hash = 31;

hash \*= xCoord + (yCoord \* (xCoord - yCoord));

hash \*= yCoord + (xCoord \* (yCoord - xCoord));

return hash;

}

}

//6

package com.hafi.lab3;

/\*\*

\* This class represents a simple two-dimensional map composed of square cells.

\* Each cell specifies the cost of traversing that cell.

\*\*/

public class Map2D

{

/\*\* The width of the map. \*\*/

private int width;

/\*\* The height of the map. \*\*/

private int height;

/\*\*

\* The actual map data that the pathfinding algorithm needs to navigate.

\*\*/

private int[][] cells;

/\*\* The starting location for performing the A\* pathfinding. \*\*/

private Location start;

/\*\* The ending location for performing the A\* pathfinding. \*\*/

private Location finish;

/\*\* Creates a new 2D map, with the specified width and height. \*\*/

public Map2D(int width, int height)

{

if (width <= 0 || height <= 0)

{

throw new IllegalArgumentException(

"width and height must be positive values; got " + width +

"x" + height);

}

this.width = width;

this.height = height;

cells = new int[width][height];

// Make up some coordinates for start and finish.

start = new Location(0, height / 2);

finish = new Location(width - 1, height / 2);

}

/\*\*

\* This helper method checks the specified coordinates to see if they are

\* within the map's boundaries. If the coordinates are not within the map

\* then the method throws an <code>IllegalArgumentException</code>.

\*\*/

private void checkCoords(int x, int y)

{

if (x < 0 || x > width)

{

throw new IllegalArgumentException("x must be in range [0, " +

width + "), got " + x);

}

if (y < 0 || y > height)

{

throw new IllegalArgumentException("y must be in range [0, " +

height + "), got " + y);

}

}

/\*\* Returns the width of the map. \*\*/

public int getWidth()

{

return width;

}

/\*\* Returns the height of the map. \*\*/

public int getHeight()

{

return height;

}

/\*\*

\* Returns true if the specified coordinates are contained within the map

\* area.

\*\*/

public boolean contains(int x, int y)

{

return (x >= 0 && x < width && y >= 0 && y < height);

}

/\*\* Returns true if the location is contained within the map area. \*\*/

public boolean contains(Location loc)

{

return contains(loc.xCoord, loc.yCoord);

}

/\*\* Returns the stored cost value for the specified cell. \*\*/

public int getCellValue(int x, int y)

{

checkCoords(x, y);

return cells[x][y];

}

/\*\* Returns the stored cost value for the specified cell. \*\*/

public int getCellValue(Location loc)

{

return getCellValue(loc.xCoord, loc.yCoord);

}

/\*\* Sets the cost value for the specified cell. \*\*/

public void setCellValue(int x, int y, int value)

{

checkCoords(x, y);

cells[x][y] = value;

}

/\*\*

\* Returns the starting location for the map. This is where the generated

\* path will begin from.

\*\*/

public Location getStart()

{

return start;

}

/\*\*

\* Sets the starting location for the map. This is where the generated path

\* will begin from.

\*\*/

public void setStart(Location loc)

{

if (loc == null)

throw new NullPointerException("loc cannot be null");

start = loc;

}

/\*\*

\* Returns the ending location for the map. This is where the generated

\* path will terminate.

\*\*/

public Location getFinish()

{

return finish;

}

/\*\*

\* Sets the ending location for the map. This is where the generated path

\* will terminate.

\*\*/

public void setFinish(Location loc)

{

if (loc == null)

throw new NullPointerException("loc cannot be null");

finish = loc;

}

}

//7

package com.hafi.lab3;

public class Waypoint {

/\*\* The location of this waypoint. \*\*/

Location loc;

/\*\*

\* The previous waypoint in this path, or <code>null</code> if this is

\* the root of the A\* search.

\*\*/

Waypoint prevWaypoint;

/\*\*

\* This field stores the total previous cost of getting from the starting

\* location to this waypoint, through the chain of waypoints. This is an

\* actual cost of following the path; it does not include any estimates.

\*\*/

private float prevCost;

/\*\*

\* This field stores an estimate of the remaining cost of traveling from

\* this waypoint to the final destination.

\*\*/

private float remainingCost;

/\*\*

\* Construct a new waypoint for the specified location. A previous waypoint

\* can optionally be specified, or the reference can be <code>null</code> to

\* indicate that the waypoint is the start of the path.

\*\*/

public Waypoint(Location loc, Waypoint prevWaypoint)

{

this.loc = loc;

this.prevWaypoint = prevWaypoint;

}

/\*\* Returns the location of the waypoint. \*\*/

public Location getLocation()

{

return loc;

}

/\*\*

\* Returns the previous waypoint in the path, or <code>null</code> if this

\* is the start of the path.

\*\*/

public Waypoint getPrevious()

{

return prevWaypoint;

}

/\*\*

\* This mutator allows both the previous cost and the remaining cost to be

\* set in one method call. Normally these values will be set at the same

\* time anyway.

\*\*/

public void setCosts(float prevCost, float remainingCost)

{

this.prevCost = prevCost;

this.remainingCost = remainingCost;

}

/\*\*

\* Returns the actual cost of getting to this point from the starting

\* location, through the series of waypoints in this chain.

\*\*/

public float getPreviousCost()

{

return prevCost;

}

/\*\*

\* Returns an estimate of the remaining cost of traveling from this

\* point to the final destination.

\*\*/

public float getRemainingCost()

{

return remainingCost;

}

/\*\*

\* Returns the total cost estimate for this waypoint. This includes the

\* actual cost of getting to this point from the starting location, plus

\* the estimate of the remaining cost of traveling from this point to

\* the final destination.

\*\*/

public float getTotalCost()

{

return prevCost + remainingCost;

}

}





