**Федеральное агентство связи**

**Ордена Трудового Красного Знамени**

**Федеральное государственное бюджетное образовательное учреждение высшего образования**

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

**Кафедра Информатики**

****

**Отчет по лабораторной работе №3**

по предмету «КТП»:

Выполнил: студент группы БВТ1802

Каринов Евгений Александрович

Руководитель:

Ксения Андреевна Полянцева

Москва 2020

**1 Цель работы**

Цель работы: изучить алгоритм A\*.

**2 Задание**

Дополнить исходный текст программы таким образом, чтобы она находила кратчайший путь в обход препятствий.

**3 Текст программы**

**Class AStarApp**

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();  
 }   
}

**Class AStarState**

import java.util.\*;  
  
*/\*\*  
 \* This class stores the basic state necessary for the A\* algorithm to compute a  
 \* path across a map. This state includes a collection of "open waypoints" and  
 \* another collection of "closed waypoints." In addition, this class provides  
 \* the basic operations that the A\* pathfinding algorithm needs to perform its  
 \* processing.  
 \*\*/*public class AStarState  
{  
 */\*\* This is a reference to the map that the A\* algorithm is navigating. \*\*/* private Map2D map;  
 private Map<Location, Waypoint> Opened = new java.util.HashMap<Location, Waypoint>();  
 private Map<Location, Waypoint> Closed = new java.util.HashMap<Location, Waypoint>();  
  
  
 */\*\*  
 \* Initialize a new state object for the A\* pathfinding algorithm to use.  
 \*\*/* 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()  
 {  
 if (Opened.size() == 0) return null;  
 ArrayList<Waypoint> waypoints = new ArrayList<Waypoint>(Opened.values());  
 float mincost = waypoints.get(0).getTotalCost();  
 Waypoint min = waypoints.get(0);  
 for (int i = 1; i < waypoints.size(); i++) {  
 if (waypoints.get(i).getTotalCost() < mincost) {  
 min = waypoints.get(i);  
 mincost = min.getTotalCost();  
 }  
 }  
 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.get(newWP.getLocation()) == null ) {  
 Opened.put(newWP.getLocation(), newWP);  
 return true;  
 }  
 else  
 {  
 if (Opened.get(newWP.getLocation()).getPreviousCost() >  
 newWP.getPreviousCost()) {  
 Opened.put(newWP.getLocation(), newWP);  
 return true;  
 }  
 }  
 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)  
 {  
 if (Closed.containsKey(loc)) return true;  
 return false;  
 }  
}

**Class Location**

import java.util.Objects;  
  
*/\*\*  
 \* This class represents a specific location in a 2D map. Coordinates are  
 \* integer values.  
 \*\*/*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 o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Location location = (Location) o;  
 return xCoord == location.xCoord &&  
 yCoord == location.yCoord;  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(xCoord, yCoord);  
 }  
}

**4 Работа программы**

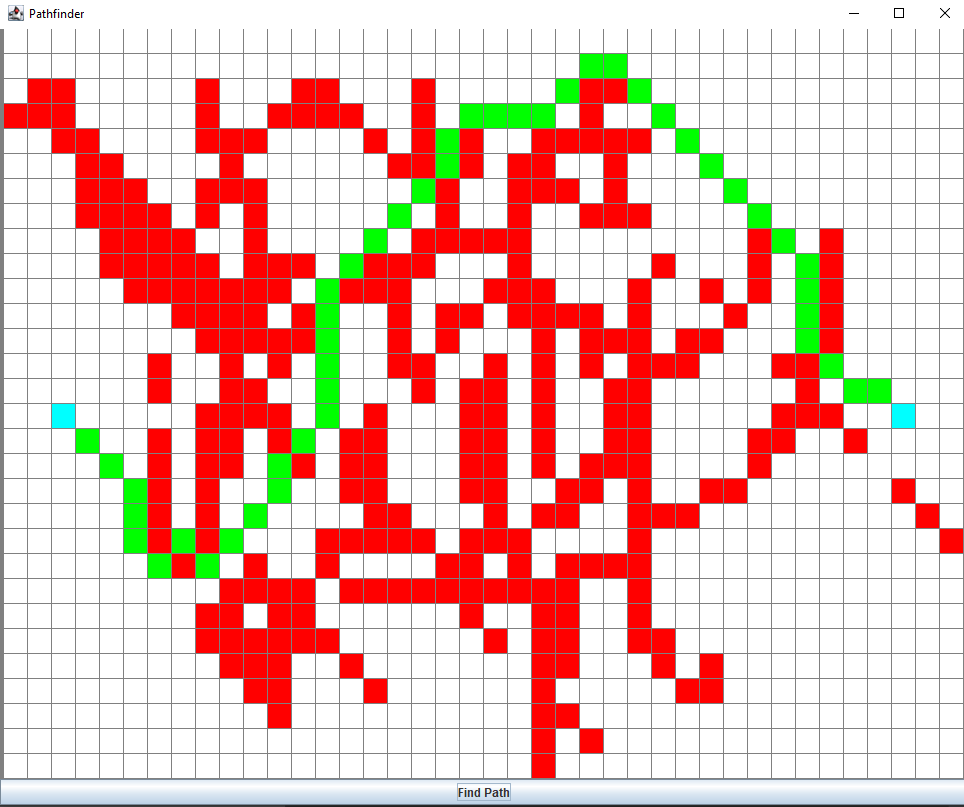


Рисунок 1 — Результат работы программы