МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ

КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ

Ордена Трудового Знамени федеральное государственное

Бюджетное образовательное учреждение высшего образования

МОСКОВСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ СВЯЗИ И

ИНФОРМАТИКИ

Кафедра математической кибернетики и информационных технологий

Лабораторная работа №3 по теме:

«Алгоритм A\* («A star»)»

Выполнил: студент группы БИБ1902 \_\_\_\_\_\_\_Арустамян А.Б.

Проверил: \_\_\_\_\_\_\_\_Херсонский А. В.

Москва 2021

**Цель работы:**

Дописать код алгоритма поиска A\*. Концепция алгоритма довольно проста, начиная с исходного местоположения, алгоритм постепенно строит путь от исходной точки до места назначения, используя наикратчайший путь, чтобы сделать следующий шаг. Это гарантирует, что полный путь будет также оптимальным.

**Код программы:**

**AStarApp.java**

*/\*\*  
 \* 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();  
 }  
}

**Location.java**

package com.company;  
  
*/\*\*  
 \* 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);  
 }  
  
 public boolean equals(Object obj) {  
  
 if (obj instanceof Location) {  
 Location other = (Location) obj;  
 if (xCoord == other.xCoord && yCoord == other.yCoord) {  
 return true;  
 }  
 }  
 return false;  
 }  
  
 public int hashCode() {  
 int result = 41;  
  
 result = 93 \* result + xCoord;  
 result = 93 \* result + yCoord;  
 return result;  
 }  
}

**AStarState.java**

package com.company;  
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 final Map2D map;  
 private final HashMap<Location, Waypoint> openedWaypoints = new HashMap<>();  
 private final HashMap<Location, Waypoint> closedWaypoints = new HashMap<> ();  
  
 */\*\*  
 \* 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 (numOpenWaypoints() == 0)  
 return null;  
  
 Set<Location> open\_waypoint\_keys = openedWaypoints.keySet();  
 Iterator<Location> i = open\_waypoint\_keys.iterator();  
 Waypoint best = null;  
 float best\_cost = Float.*MAX\_VALUE*;  
  
 while (i.hasNext())  
 {  
 Location location = i.next();  
 Waypoint waypoint = openedWaypoints.get(location);  
 float waypoint\_total\_cost = waypoint.getTotalCost();  
 if (waypoint\_total\_cost < best\_cost)  
 {  
 best = openedWaypoints.get(location);  
 best\_cost = waypoint\_total\_cost;  
 }  
 }  
 return best;  
 }  
  
 */\*\*  
 \* 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)  
 {  
 Location location = newWP.getLocation();  
 if (openedWaypoints!=null && openedWaypoints.containsKey(location))  
 {  
 Waypoint current\_waypoint = openedWaypoints.get(location);  
 if (newWP.getPreviousCost() < current\_waypoint.getPreviousCost())  
 {  
 openedWaypoints.put(location, newWP);  
 return true;  
 }  
 return false;  
 }  
 openedWaypoints.put(location, newWP);  
 return true;  
 }  
  
  
 */\*\* Returns the current number of open waypoints. \*\*/* public int numOpenWaypoints()  
 {  
 return openedWaypoints.size();  
 }  
  
  
 */\*\*  
 \* This method moves the waypoint at the specified location from the  
 \* open list to the closed list.  
 \*\*/* public void closeWaypoint(Location loc)  
 {  
 Waypoint waypoint = openedWaypoints.remove(loc);  
 closedWaypoints.put(loc, waypoint);  
 }  
  
 */\*\*  
 \* Returns true if the collection of closed waypoints contains a waypoint  
 \* for the specified location.  
 \*\*/* public boolean isLocationClosed(Location loc)  
 {  
 return closedWaypoints.containsKey(loc);  
 }  
  
}

**AStarPathfinder.java**

*\* This class contains the implementation of the A\* pathfinding algorithm. The  
 \* algorithm is implemented as a static method, since the pathfinding algorithm  
 \* really doesn't need to maintain any state between invocations of the  
 \* algorithm.  
 \*/*public class AStarPathfinder  
{  
 */\*\*  
 \* This constant holds a maximum cutoff limit for the cost of paths. If a  
 \* particular waypoint happens to exceed this cost limit, the waypoint is  
 \* discarded.  
 \*\*/* public static final float *COST\_LIMIT* = 1e6f;  
  
  
 */\*\*  
 \* Attempts to compute a path that navigates between the start and end  
 \* locations of the specified map. If a path can be found, the waypoint of  
 \* the <em>final</em> step in the path is returned; that waypoint can be  
 \* used to walk backwards to the starting point. If no path can be found,  
 \* <code>null</code> is returned.  
 \*\*/* public static Waypoint computePath(Map2D map)  
 {  
 // Variables necessary for the A\* search.  
 AStarState state = new AStarState(map);  
 Location finishLoc = map.getFinish();  
  
 // Set up a starting waypoint to kick off the A\* search.  
 Waypoint start = new Waypoint(map.getStart(), null);  
 start.setCosts(0, *estimateTravelCost*(start.getLocation(), finishLoc));  
 state.addOpenWaypoint(start);  
  
 Waypoint finalWaypoint = null;  
 boolean foundPath = false;  
  
 while (!foundPath && state.numOpenWaypoints() > 0)  
 {  
 // Find the "best" (i.e. lowest-cost) waypoint so far.  
 Waypoint best = state.getMinOpenWaypoint();  
  
 // If the best location is the finish location then we're done!  
 if (best.getLocation().equals(finishLoc))  
 {  
 finalWaypoint = best;  
 foundPath = true;  
 }  
  
 // Add/update all neighbors of the current best location. This is  
 // equivalent to trying all "next steps" from this location.  
 *takeNextStep*(best, state);  
  
 // Finally, move this location from the "open" list to the "closed"  
 // list.  
 state.closeWaypoint(best.getLocation());  
 }  
  
 return finalWaypoint;  
 }  
  
 */\*\*  
 \* This static helper method takes a waypoint, and generates all valid "next  
 \* steps" from that waypoint. The new waypoints are added to the "open  
 \* waypoints" collection of the passed-in A\* state object.  
 \*\*/* private static void takeNextStep(Waypoint currWP, AStarState state)  
 {  
 Location loc = currWP.getLocation();  
 Map2D map = state.getMap();  
  
 for (int y = loc.yCoord - 1; y <= loc.yCoord + 1; y++)  
 {  
 for (int x = loc.xCoord - 1; x <= loc.xCoord + 1; x++)  
 {  
 Location nextLoc = new Location(x, y);  
  
 // If "next location" is outside the map, skip it.  
 if (!map.contains(nextLoc))  
 continue;  
  
 // If "next location" is this location, skip it.  
 if (nextLoc == loc)  
 continue;  
  
 // If this location happens to already be in the "closed" set  
 // then continue on with the next location.  
 if (state.isLocationClosed(nextLoc))  
 continue;  
  
 // Make a waypoint for this "next location."  
  
 Waypoint nextWP = new Waypoint(nextLoc, currWP);  
  
 // OK, we cheat and use the cost estimate to compute the actual  
 // cost from the previous cell. Then, we add in the cost from  
 // the map cell we step onto, to incorporate barriers etc.  
  
 float prevCost = currWP.getPreviousCost() +  
 *estimateTravelCost*(currWP.getLocation(),  
 nextWP.getLocation());  
  
 prevCost += map.getCellValue(nextLoc);  
  
 // Skip this "next location" if it is too costly.  
 if (prevCost >= *COST\_LIMIT*)  
 continue;  
  
 nextWP.setCosts(prevCost,  
 *estimateTravelCost*(nextLoc, map.getFinish()));  
  
 // Add the waypoint to the set of open waypoints. If there  
 // happens to already be a waypoint for this location, the new  
 // waypoint only replaces the old waypoint if it is less costly  
 // than the old one.  
 state.addOpenWaypoint(nextWP);  
 }  
 }  
 }  
  
 */\*\*  
 \* Estimates the cost of traveling between the two specified locations.  
 \* The actual cost computed is just the straight-line distance between the  
 \* two locations.  
 \*\*/* private static float estimateTravelCost(Location currLoc, Location destLoc)  
 {  
 int dx = destLoc.xCoord - currLoc.xCoord;  
 int dy = destLoc.yCoord - currLoc.yCoord;  
  
 return (float) Math.*sqrt*(dx \* dx + dy \* dy);  
 }  
}

**JMapCell.java**

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.  
 \*/*public 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());  
 }  
}

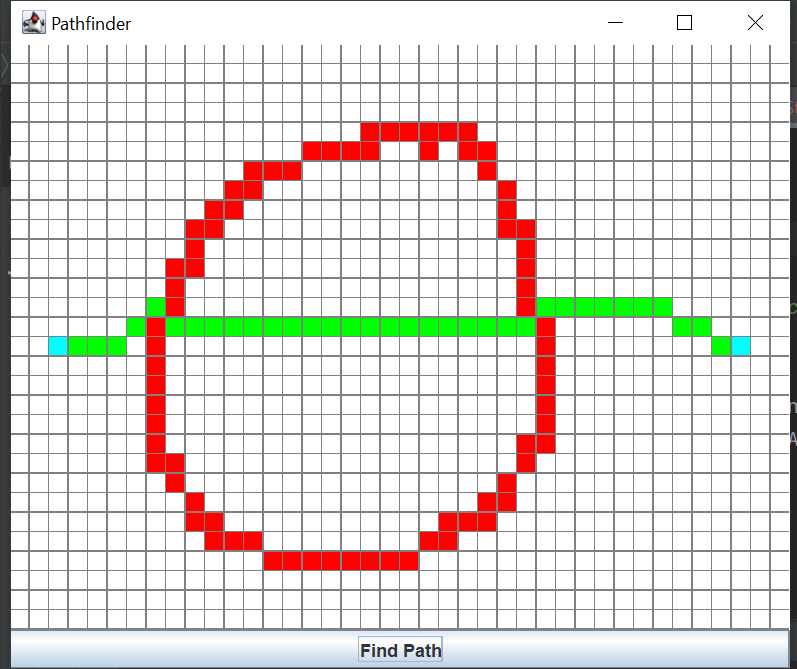
**Map2D.java**

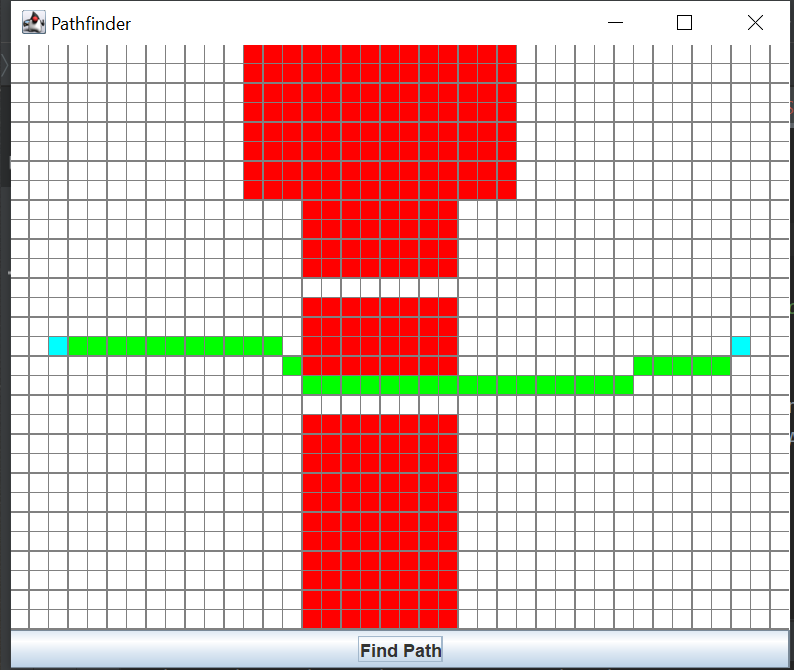
package com.company;  
  
*/\*\*  
 \* This class represents a simple two-dimensional map composed of square cells.  
 \* Each cell specifies the cost of traversing that cell.  
 \*\*/*public class Map2D  
{  
 */\*\* The width of the map. \*\*/* private int width;  
  
 */\*\* The height of the map. \*\*/* 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;  
 }  
}

**Waypoint.java**

*/\*\*  
 \* This class represents a single step in a path generated by the A\* pathfinding  
 \* algorithm. Waypoints consist of a location, the previous waypoint in the  
 \* path, and some cost values used to determine the best path.  
 \*\*/*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;  
 }  
}

**Пример работы программы:**

****

****

**Вывод:**

Благодаря этой лабораторной работе я разобрался с алгоритмом поиска A\* и оптимизировал его код для поиска кратчайшего пути.