**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 дисциплине «КТП»

на тему:

«**Алгоритм А\*(A star)»**

Выполнила: студентка группы БВТ1802

Лаврухина Елена Павловна

Руководитель:

Полянцева Ксения Андреевна

Москва 2020

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.

Выполнение

Код программы

1. 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);  
 }  
 **public boolean** equals(Object obj) {  
 **if** (**this** == obj)  
 **return true**;  
 **if** (obj == **null** || getClass() != obj.getClass())  
 **return false**;  
 Location location = (Location) obj;  
 **return xCoord** == location.**xCoord** && **yCoord** == location.**yCoord**;  
 }  
 **public int** hashCode() {  
 **return** Objects.*hash*(**xCoord**, **yCoord**);  
 }  
  
}

1. 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 point = waypoints.get(0);  
 **for** (**int** i = 1; i < waypoints.size(); i++) {  
 **if** (waypoints.get(i).getTotalCost() < mincost) {  
 point = waypoints.get(i);  
 mincost = point.getTotalCost();  
 }  
 }  
 **return** point;  
 }  
  
 */\*\*  
 \* 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**;  
 }  
}

Скриншоты работы программы

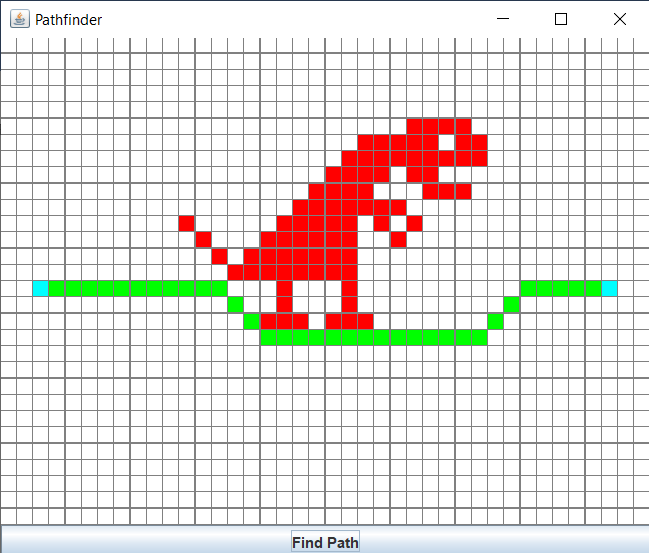


Рисунок 1 – Работа программы

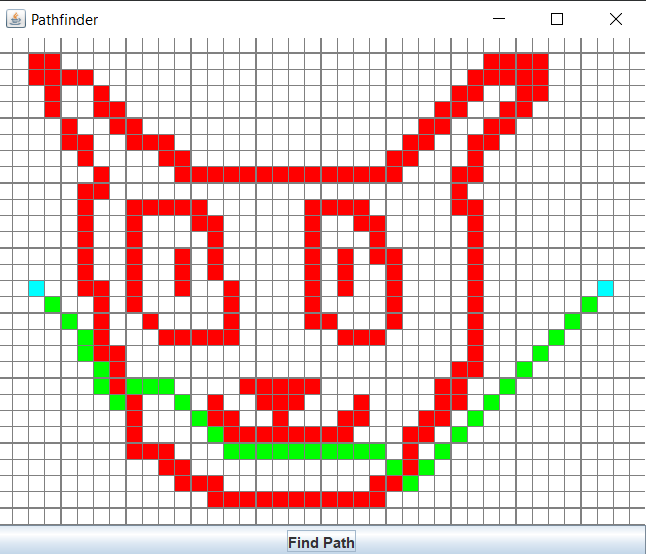


Рисунок 2 – Работа программы

Вывод

В ходе работы мы познакомились с алгоритмом A\* и с помощью него реализовали работу программы по нахождению кратчайшего пути от начального местоположения до пункта назначения с успешным преодолением препятствий.