ITG – Graph Search Algorithms Task by Mahmoud Anbousi

Introduction

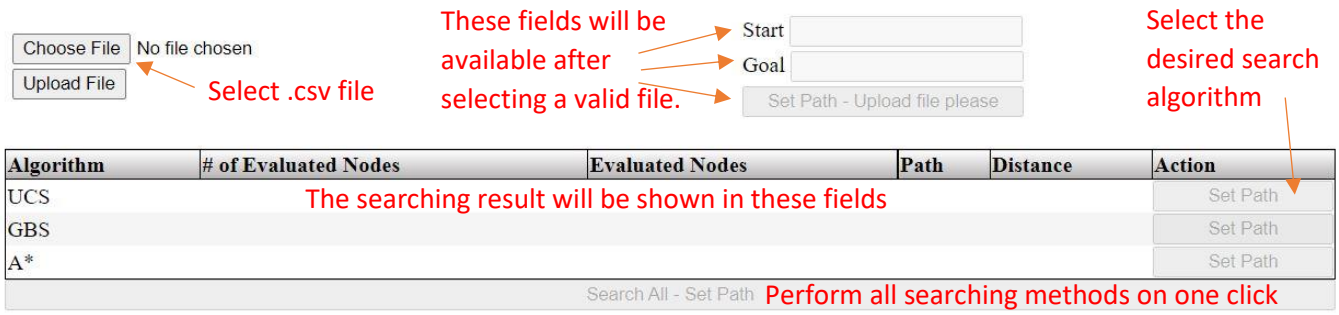
This web program was built using Java programming language with Spring boot 4.

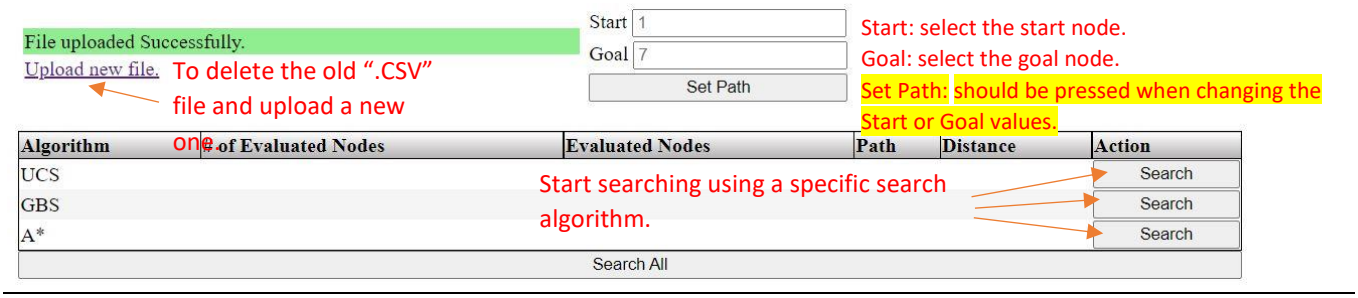
The main purpose of this program is to calculate the Path between two node “Start and Goal nodes” using three different graph searching algorithms.

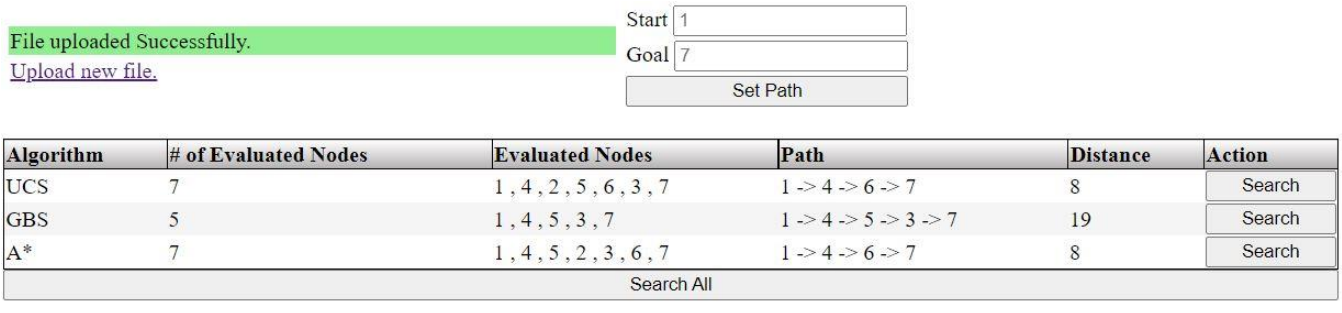
1- Uniform Cost Search Algorithm.
2- Greedy Best Search Algorithm.
3- A* Search Algorithm.

The program will receive the data through uploading a “.CSV file”. This file will contain an adjacency matrix, which define the Nodes and the connections between these Nodes. These nodes will be transformed to a weighted and directed Graph data structure at first, then the program will be allowed to start the search depending on the selected search algorithm.

The result will be shown in a table with fields of (Algorithm, # of Evaluated Nodes, Evaluated Nodes, Path and Distance).







Searching Methods

Uniform Cost Search Algorithm (UCS):

This method uses the lowest cumulative edge cost to find the shortest path “lowest cost” from the start node to the goal node. This method uses a Graph data.

Pseudocode:

Create two list “visited list” and “checking list”.

Insert the star node into the checking list.

Loop while this checked list is not empty. {

Sort this list on the lowest edge cost values.

Remove the lowest cost node from that list.

Check if the removed node is the goal node. “If yes, add this node to the visited list, stop the search and calculate the costs between nodes in visited list”, stop the while loop.

If no, add this node the visited bring all nodes that can be reached directly by this node “Child nodes”.

Check if any child node is in checked list or visited list. “If its not, add this node to checked list. If yes, compare the edge cost between the old node and the new node and keep the node with the lower cost value”.

}

Greedy Best Search Algorithm

This method selects the path from the start node to the goal node depending on the lowest cumulative heuristic value “Weight or predicted cost value” for the next node. This method uses a Weighted Graph data.

Pseudocode:

Create two list “visited list” and “checking list”.

Insert the star node into the checking list.

Loop while this checked list is not empty. {

Sort this list on the lowest Heuristic values “of the next node”.

Remove the lowest node from the checked list.

Check if the removed node is the goal node. “If yes, add this node to the visited list, stop the search and calculate the costs between nodes in visited list”, stop the while loop.

If no, add this node the visited bring all nodes that can be reached directly by this node “Child nodes”.

Check if any child node is in checked list or visited list. “If it’s not, add this node to checked list. If yes, compare the edge cost between the old node and the new node and keep the node with the lower Heuristic value”.

}

A* Search Algorithm

It is a searching algorithm that is used to find the shortest path between a start and a goal point. This method depends on both the lowest Heuristic value for the next node and the lowest cumulative cost value.

f(n) = g(n) + h(n), where:

g(n) = cost of traversing from one node to another. This will vary from node to node. *

h(n) = heuristic approximation of the node's value. This is not a real value but an approximation cost. *

Pseudocode:

Create two list “visited list” and “checking list”.

Insert the star node into the checking list.

Loop while this checked list is not empty. {

Sort this list on the lowest f(n) value.

Remove the lowest node from the checked list.

Check if the removed node is the goal node. “If yes, add this node to the visited list, stop the search and calculate the costs between nodes in visited list”, stop the while loop.

If no, add this node the visited list and bring all nodes that can be reached directly by this node “Child nodes”.

Check if any child node is in checked list or visited list. “If it’s not, add this node to checked list. If yes, compare the f(n) cost between the old node and the new node and keep the node with the lower f(n) value”.

}

---

* https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/a-star-algorithm

Technical Data

Overview

The program will receive the data through uploading a ".CSV file". This file will contain an adjacency matrix, which define the Nodes and the connections between these Nodes. These nodes will be transformed to a weighted and directed Graph data structure at first, then the program will be allowed to start the search depending on the selected search algorithm.

Matrix Example:

0,5,0,3,0,0,0
5,0,1,0,4,0,0
0,0,0,0,6,0,8
0,0,0,0,2,2,0
0,4,6,2,0,0,0
0,0,0,2,0,0,3
0,0,8,0,4,3,0
8,6,3,5,4,9,0

All rows except the last one are describing the **edge cost** between nodes.
The last row is describing the weight "**Heuristic value**" for each node.
Each row except the last one is for a one node, and the values in that row contains the cost from that node to other nodes respectively. The "0" zero value, means there is no direct way between these two nodes.

Classes, methods and variables

Node Class

```
public Node (int value, int weight) {
            this.value = value;
            this.weight = weight;
      }
```

This class is used to create nodes from the given matrix.

Each node has a unique new value and a specific weight "Heuristic value".

Auto increment value (i) through a "for loop with (i++)" is used in this program to give each new node a new value.

Edge Class

```
public Edge (int value, int cost, int from, int weight) {
      super(value,weight);
      this.setCost(cost);
      this.setFrom(from);
}
```

This class is used to create edges "ways" between "Nodes" from the given matrix.

This class is extended the Node class, because it has some mutual properties, values and methods.

Each Edge is between two nodes “Start and End nodes”.

This Edge contains the “End” node value which has a value and weight, the cost between the Start Node and the End Node and the Start node value.

Graph Class

```
HashMap<Integer , List<Node>> graph = new HashMap<>();
```

This class is used to create Nodes and Edges between these nodes depending on the given adjacency matrix.

A HashMap data structure (Key, Value) is used to create the Graph.

The key contains the value of a node and the value contains all the nodes directly connected to that node.

UploadCSV Class

This class is used to get the .csv file, read it and convert it to 2D array “List<List<Integer>>

UploadCSV.readFile(path) is a static method used to read the file.