Kyle Sherman

Tanveer Ahmed

6/25/2022

CS 4310

Final Project Report – A\* Search Algorithm

**How to run our code**

You can run the code via a Python IDE or via console window. 3 parameters are allowed: a json file to read (default is data.json), a starting city (default is Ulm), an ending city (default is Bern). If no parameters are specified, then the program will use the default values.

To execute run either:

To specify: *Python main.py [file].json [starting city] [ending city]*

For default: *Python main.py*

**Implementation**

With our specific problem, we wanted to make use of cities as a form of waypoint system. We needed to allow for a starting city, an ending city, and a group of cities to act as “waypoints” that must be reached along the path from the starting city to ending city. One way of considering our problem is to assume you are flying a general aviation aircraft between two points and must stop at arbitrary points to refuel along the way. Obviously, the best solution would be to find what the best group of cities along your path would allow you to refuel while maintaining the most cost-effective route. Problems like ours is where the A\* algorithm really shines.

The A\* search algorithm is often seen as a modified version of Dijkstra’s algorithm, with the exception of it (A\*) making use of a heuristic function to guarantee the most efficient path. Our implementation of A\* Search algorithm differed from the most common implementation of this algorithm. Typically, this algorithm will use a grid style layout and make use of either Manhattan Grid solution or octile distance solution, both of which walk down “blocks” in either 4-directions or 8-directions respectively. Whereas our solution makes use of a graph data structure connecting nodes containing coordinates for waypoints. Otherwise, our solution follows the standard general form solution of A\* algorithm.

Our implementation makes use of the following three files:

* **Main.py**: contains general code as well as the *astar\_search* function which performs the algorithm
* **Graph.py:** contains the graph class – this is our implementation of a graph data structure
* **Node.py:** contains the node class – this is how we implement a node class. There is a primary utility function called *get\_distance* that has two coordinates as input and returns the distance between in kilometers

The main function declares a start city variable, an end city variable, a coordinate list with each index being the name of a city, an instance of the graph structure using a connection function to define the connection between any two cities, and a heuristics list.

We initially had difficulty determining what we would use as our heuristics, however we opted to use the distance from the starting city to each node as that respective node’s heuristic cost. This would guarantee that the heuristic value would be either lower than or at most equal to the true value, which is a requirement for the A\* algorithm to properly work. The cost of each edge is the distance between the two nodes the respective edge connects.

Algorithm steps:

1. Call the *astar\_search* function passing through the graph, the heuristics list, the start city, and the end city
2. Create a node for the start city and the end city
3. Append the start node to the opened\_nodes list
4. Perform the following until the nodes list is empty:
   1. Sort the opened\_nodes list
   2. Pop the lowest cost node from the front of the list and mark this node as the current node
   3. Append the current node to the closed\_nodes list
   4. Base case: check if the current node is the goal node:
      1. Loop from the current (goal) node back to the start node appending each node to the goal array and return the final path list
   5. Get neighbors of the current node from the graph structure. A neighbor is the adjacent nodes
   6. Loop through each neighbor:
      1. Generate the nodes for the neighbor and calculate each neighbors total cost value (using f = g+h function where f equals the approximate distance from the start to the goal node)
      2. If the neighbor is in the open list and the neighbor has a lower total cost then append that neighbor to the opened\_nodes list
5. Return the completed path if one was found our return none if no path could be found between the start and end nodes