**COMP30024 Artificial Intelligence**

**Project Part A: Searching**

Authors: Ying Shan Saw (1118861) & Joeann Chong (1152260)

**A\* Search**

**Data Structures**

We decided on using a **Priority Queue** heap structure to implement A\* search algorithm as it allows us to order the nodes by priority – to store accessible adjacent nodes, where they were prioritised based on their distance to the goal node. [code referenced from Red Blob Games[[1]](#footnote-1)]

The Priority Queue heap structure is a heap structure that arranges nodes in a **list** by their priority. It will serve the highest priority element in the queue before any other item. In this environment, the closer they are to the goal node, the more likely we are to consider them as the better option and process them.

We tested our algorithm on 16 self-authored test cases and confirmed that all returns the shortest path cost (based on A\* search). We also added some classes and encapsulation via Object-Oriented Programming to better consolidate the information about each structure. Information regarding each class is available in the code comments and metadata.

**Implementation**

First and foremost, the data is initialised into 3 different classes: Board, Nodes, and PriorityQueue.

|  |  |
| --- | --- |
| **Class** | **Description** |
| PriorityQueue | Orders each item in the PriorityQueue list by its priority. The priority is an estimate of the total cost to goal node |
| Nodes | Contains information about the travelled nodes, (cost to travel to node (cost\_so\_far)/ where the (best path’s) preceding node is (came\_from)) |
| Board | Stores information about the board: size, start node, goal node, and the nodes already on the board. Also contains a “legend” dictionary to store what each node is. |

**Method:**

1. The data input via the json file would be processed and stored into a Board object in the main class, and which would then generate a Nodes object within the Board object upon initialisation (to store future node information)
2. After the Board object is generated, the algorithm begins by generating a PriorityQueue object, and putting in the start node as the first node to explore by popping it out of the priority queue.
3. The node being explored will be used to generate the available and valid adjacent nodes
4. For each valid adjacent node, we will calculate **the total cost of travelling from the start node to the next node**, using the formula below:

cost\_to\_node = board.nodes.cost\_so\_far[current\_node] + UNIT\_COST

where UNIT\_COST is always = 1 (as the cost to travel to an adjacent node)

and cost\_so\_far[current\_node] takes the cost to travel to the current node explored

1. If the adjacent node is “unexplored”, or the current path has a shorter cost to travel to the current node:
   1. the cost\_so\_far property in the Nodes object will be updated
   2. the priority will be updated, using the formula below

total\_cost:float = new\_cost + distance\_to\_goal(next\_node, goal)

*where priority is the “total cost to goal = travel\_to\_node cost + distance\_to\_goal”*

*and distance\_to\_goal is measured using our heuristics (Explained below)*

Background pattern

Description automatically generated

* 1. the nodes’ came\_from property will be updated to be the current explored node



1. Step 3-5 will repeat while the goal is yet to be the node explored.



The function pathfinding(board: Board)takes in a Board object that pre-processes data read from an input json file and if a solution is found, calls find\_print\_path(board: Board)that prints the solution onto the terminal.

node1



Figure 1. Denoting the cost\_so\_far(red) and estimate cost to goal(green) for node1

**Time and Space Complexity Analysis**

Let b = branching factor, d = the depth of the solution

The Big-O time complexity of the algorithm would be **O(bd).**

This is derived from the equation, where the total nodes generated would be

1+ b1 + b2 + … + bd  = **=**

The Big-O space complexity would be **O(bd).** This is because we store all the nodes and paths in the memory, and none of them are ever removed.

**Heuristic**

The hex cells are of same size and the cost path from one cell to an adjacent one is considered 1. Therefore, we believe that using Axial Hex distance is a permissible choice.

* path cost = 1 + axial\_hex\_distance (from next\_node to goal), since 1 is always constant (from next node)
* assume by calculating axial hex distances gives us the best estimation
* overall time complexity O(n)

To start off, we need to understand how the distance between two hex nodes are derived.

Axial hex distance is derived from the cube-coordinate system where there exists a plane at x+y+z = 0 on a cube grid, with the coordinates (q, r, s) stored, and the algorithm ensures that the sum of these coordinates equals to 0. It stores (q, r) and calculates s = - q - r since q+r+s = 0. [heuristic referenced from Red Blob Games[[2]](#footnote-2)]

Because Euclidean (straight-line) distance does not work with 3-D planes (and Axial coordinates), we will instead use the Manhattan distance formula. It derives the distance between two points measured along axes at right angles, and it is originally used to calculate the distance between two cubes on a 3-D plane:

manhattan\_distance = (abs(dq) + abs(ds) + abs(dr))/2

*where dq, ds, dr is the difference between the two cubes’ coordinates (q,r,s)*

Using the same formula, if we enter the axial coordinates from hex1 and hex2, we get

hex\_manhattan\_distance = (abs(hex1.q – hex2.q) + abs(hex1.q + hex2.r – hex1.q – hex2.r) + abs(hex1.r – hex2.r)) / 2

which simplifies to

hex\_manhattan\_distance = (abs(x) + abs(x+y) + abs(y)) / 2

**Challenge Question**

As stated in the question, there will be both “team” nodes and blocking nodes, and placing a minimal subset of new team nodes to create an optimal path to the goal nodes.

In this case, we would just have to check whether the “adjacent node” is an existing team node, and take off the UNIT\_COST variable from the cost\_to\_node formula (which is the distance from the start node to the adjacent node). In this case, the priority would be reduced, and there is a higher likelihood of the node being “captured” than other empty nodes. By doing this, the algorithm will try to find a continuous set of “team nodes” rather than using the shortest path using the empty nodes.

1. <https://www.redblobgames.com/pathfinding/a-star/implementation.html> [↑](#footnote-ref-1)
2. <https://www.redblobgames.com/grids/hexagons/> [↑](#footnote-ref-2)