CSC3206 Artificial Intelligence

Assignment 2

Group 16:

1. Choon Han Wen Benjamin (22018345)
2. Chen Wei Lun (19112697)
3. Tai Yoong Wei (20093001)
4. Lew Ze Yu (20062832)

Contents

1.0 Introduction

This report will explain the implementation of the algorithm and present the output of the algorithm which is the solution to the problem. Moreover, the limitations and strengths of the algorithm will be evaluated.

2.0 Implementation and explanation

This section explains the implementation for the algorithm and the reasoning behind the implementation.

2.1 Map Representation

Firstly, the virtual map of the problem has to be represented in a way that allows the algorithm to function properly. In order to represent this, each grid will be presented with a node that has coordinates. The coordinate system used to represent the hexagon grid is offset even-q coordinate [1]. In this coordinate system, the neighbour of the node can be gotten through a number of calculations based on whether the column is even or odd. For example, if we want the top-right neighbour of an even column node, we can get the neighbour by (x,y) + (1,0) with x representing the column of the node and y representing the row of the node. Moreover, each node will be assigned a node type such as obstacles, treasure, traps and more based on the provided assignment question. We went with this coordinate system as it is the least complicated one and it requires less computations compared to the coordinate system such as axial coordinate and cube coordinate.

2.2 A* Algorithm Flow

The algorithm chosen for this assignment is A* algorithm. A* algorithm allows us to find the least cost by utilizing heuristic to reduce the number of nodes to be explored. The general flow of the A* algorithm will first be explained before going into a specific explanation of the other parts.

Once the map representation has been created along with the initialization of all node coordinates and the list of goal nodes, the algorithm can start to find the least path cost to reach all treasure nodes. The algorithm first appends the coordinate for the initial starting node into the frontier to start the path finding loop. The algorithm first sorts the frontier by the heuristic included, allowing the lowest heuristic node to be expanded first. It then checks for whether the nodes are Trap 3 or Trap 4 respectively in order to activate the effect of the traps since those traps have effects which could affect the parent nodes. Once the verification

and execution has been done, the algorithm checks whether the current node is part of the list of goal nodes. If it is part of the goal nodes, the node will then be removed from the goal list. The frontier, explored list and nodes will be reset so that the algorithm can proceed finding the other treasures. After this, the algorithm checks whether the goal state has been achieved and stops the algorithm if it is achieved. The goal state in this case is once all the treasure has been collected.

```
for child in children:
    # Calculate the total cost of the child
    tentative_cost = calculateCost(stateSpace, path(final_solution,current_node),  child)
    # If the child is not in explored and frontier, the current node will be assigned as this node's parent.
    # Heuristic and total_cost will be assigned to this node and it will be appended
    if not any(explored_node.coordinates == child.coordinates for explored_node in explored) and not any(
            frontier_node.coordinates == child.coordinates for frontier_node in frontier):
        parent[child] = current_node
        child.parent = current_node
        child.total_cost = tentative_cost
        child.heuristic = heuristicFunction(child,goals)
        frontier.append(child)
    # if any node in frontier is more than the tentative cost, the node will be removed and replaced
    # with the same node but with new parent, total_cost and heuristic
    elif any(frontier_node.coordinates == child.coordinates for frontier_node in frontier):
        existing_node = next((n for n in frontier if n.coordinates == child.coordinates), None)
        if existing_node and existing_node.total_cost > tentative_cost:
            frontier.remove(existing_node)
            child.parent = current_node
            child.total_cost = tentative_cost
            child.heuristic = heuristicFunction(child, goals)
            frontier.append(child)
```

Figure 1: Loop for checking whether child node is to be added into the frontier

After all the checks, the algorithm proceeds to add the current node to the explored list and expands the node to find its neighbouring nodes. The cost for each neighbouring node is calculated before going through two if statements. The first if statement appends the node to the frontier if the neighbouring node is not found in both explored list and frontier. The second if statement first checks whether there is a same node in the frontier list. If there is a same node in the list, the total cost between the newly calculated total cost and the frontier list's existing node total cost will be compared. The lower cost node will be assigned to the node instead with a new parent. For both of the if statements, the node is assigned a newly calculated total cost and heuristic as well as assigning the current node as its parent before being appended into the frontier list. This whole cycle goes on until the goal state has been achieved, returning the solution and total cost of the solution.

2.3 Heuristic Function

```python
def heuristic(current_node, treasure_available):
    if len(treasure_available) > 0:
        current_coordinates = current_node.coordinates
        treasure_distances = []
        # convert the current node coordinates to axial coordinate
        x, y = current_coordinates
        y = (x + (x & 1)) / 2

        # Iterate through each treasure to get their distance from current node. Distance used is manhattan distance
        for treasure in treasure_available:
            # convert treasure node coordinates to axial coordinate
            a, b = treasure.coordinates
            b = (a + (a & 1)) / 2

            distance = (abs(x - a) + abs(y + b) + abs(x + y - a - b)) /2
            treasure_distances.append((treasure, distance))
        # Sort treasure and return the closest treausre
        treasure_distances.sort(key=lambda x: x[1])
        return treasure_distances[0]
    else:
        return None
```

```python
def heuristicFunction(node, treasure_available):
    # the node's total cost so far
    g = node.total_cost
    # heuristic calculated
    h = heuristic(node, treasure_available)[1]
    if h is None:
        return 0
    return g + h
```

Figure 2.1 and 2.2: Heuristic and Heuristic Function

The heuristic function works through a formula f(n) = g(n) + h(n) where g(n) represents the total cost of the node and h(n) represents the distance between the node and its goal node. In our implementation, we calculated h(n) by first calculating the Manhattan distance between one node and the other 4 treasures. The shortest distance is then chosen as the h(n), allowing the algorithm to work towards the closest treasure first. As the map also includes rewards and traps where it affects the energy or steps required to the next node, the calculation for g(n) (total cost) needs to take that into account. For the total cost calculation, it is done by having a base multiplier for energy and step which is 1. The calculation then iterates through each node in the path and adds the cost to move to each node into a total cost variable. The cost will be calculated by multiplying the energy and steps taken. Throughout the iteration, if the nodes are either one of the traps or rewards that affect the energy or step multiplier, it will be executed by modifying the multiplier of the steps and energy. After that, every iteration of the node will also double or half the multiplier depending on the effect activated. For example, if

Reward 1 was activated, the energy multiplier will be halved each iteration. After both calculations have been done, the g(n) is then calculated by adding f(n) and h(n) together.

2.4 Trap 3 and Trap 4

As Trap 3 and trap 4 do not directly affect the total cost of a node, they are not checked when calculating the total cost. The implementation of Trap 3 works by getting the direction of the previous movement which is done by subtracting the node's parent coordinate from the current node's coordinate. The column of the parent node's coordinate is then checked to see whether it is row or even in order to find the right direction. Once the direction has been determined, the coordinate of the current node will be added by the direction's coordinates twice (even column and odd column direction coordinates) to move it twice in the direction. Trap 4 removes all goal nodes from the goals list, meaning that they would not go through the algorithm to append its neighbour into the frontier, effectively ending its path. The goal node is then restored at the end to ensure that the algorithm will continue working until it has found the least cost path to all treasure nodes.

2.5 Goal State

As this assignment requires a goal state whether there are multiple goal nodes, it is important to modify the algorithm to take that into account. Our modification of this works by basically resetting the whole frontier, explored list and nodes when it reaches one of the goal nodes. This basically resets the algorithm to treat the found goal node as the initial coordinate so it can continue finding the goal nodes. Moreover, this allows the algorithm to explore previously entered paths so we can ensure that the path is truly the shortest path. We have also implemented a way to store the path towards this goal node to be used for the total cost calculation, allowing us to maintain the effects of the traps and rewards even after the reset. This is due to the way our total cost calculation is implemented. Our total cost calculation traces back the path of the current node while also taking in the stored path from the previous goal node so that it calculates the energy and step multiplier to get the total cost. This cycle keeps going until all four treasures have been collected. Since collecting four treasures is the goal state, that will be included in an if statement to end the algorithm.

3.0 Results Display / Visualisation

The visualization done in the code utilizes the 'tkinter' library, which is a standard GUI toolkit for Python [2] [3]. The toolkit is used to create a graphical representation of a hexagonal grid similar to what was given in the assignment prompt. In total, the visualization

is done in 3 separate windows, each respectively displaying the map with solution, a window to display a text based solution and a final window to display legends to assist in interpreting the map.

The decision to implement a GUI based visualization is when we discovered that using text-based representation to represent the path is highly inefficient since the reader has to slowly compare the results with a picture of the map. After discussion, we decided to switch over to using 'tkinter' instead of going with a text-based system. Additionally, using a GUI based interface means that we can implement colours to our map, which enables immediate understanding since the reader can immediately distinguish between normal nodes, traps, rewards, obstacles and treasures.

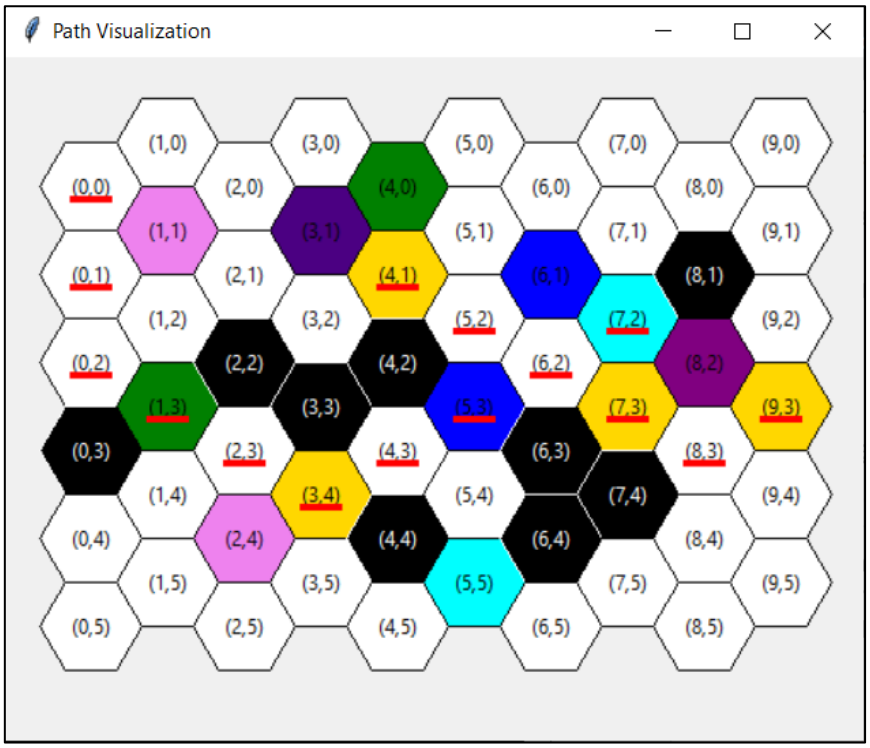

Figure 3: Overall Layout Of The Map

(Associated classes: Node, createMap, draw_hexagon, draw_grid, draw_path)

We have chosen to recreate the map provided in the assignment question to best represent what the algorithm can output. Thus, a grid layout composed of hexagonal cells arranged in a staggered layout is created [1]. The coordinate system used to represent the hexagon grid is offset even-q coordinate as mentioned above. Each cell is labelled with a coordinate system (x,y), where x represents the column and y represents the row. In the map, the underlined coordinates represent the path generated by the A* algorithm to reach all treasure nodes. All nodes other than normal nodes are also colour coded to ensure the reader can understand the map at first glance.

A special thing to take into consideration is the generation of hexagons, which is done via a for-loop for each hexagon. Each vertices of the hexagon is generated and stored in an empty list, with the x,y coordinate of each vertices calculated via the cosine and sine of the angle. The angle in question is calculated by a math equation which ensures the vertices are spaced 60 degrees apart ($\pi/3$ radians) apart. After all 6 vertices for a hexagon are generated, the canvas.create_polygon method is called to 'draw' the hexagon on the existing canvas.

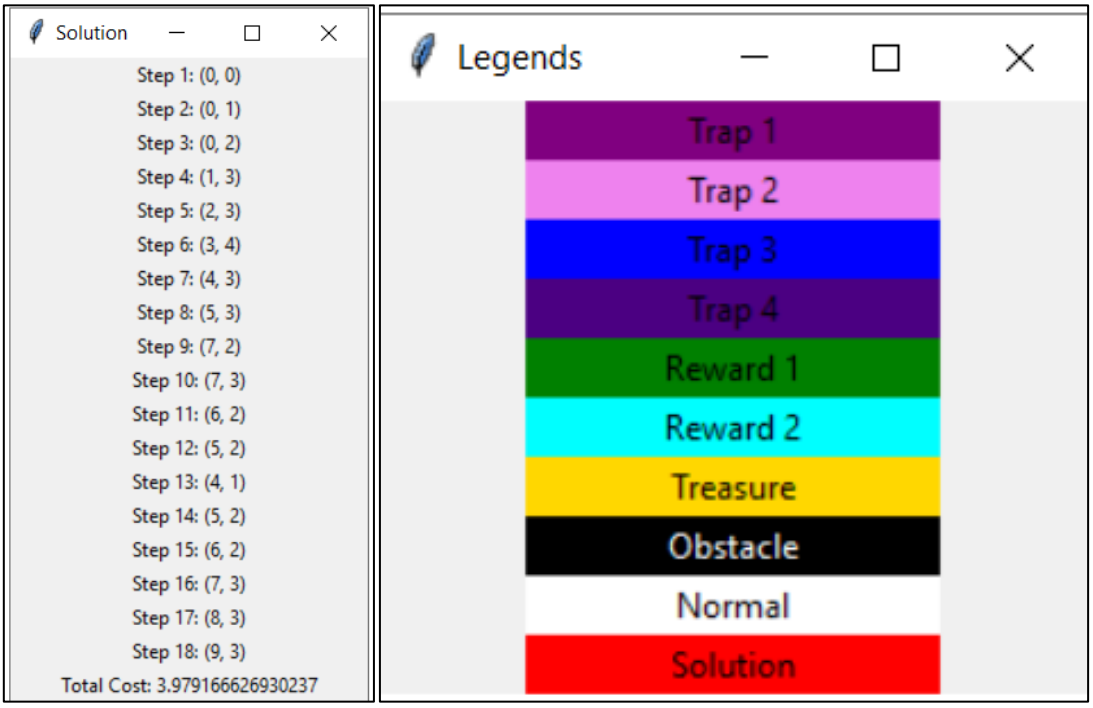

Figure 3.1 and 3.2: Text-based solution and legends

(Associated classes for "Solution": Node, a_star, display_solution)

(Associated classes for "Legends": Node, display_legends)

As for the window displaying the solution, it is implemented by looping through step in the solution list provided by the A* algorithm. A label for each step is included to display the step number and the step itself. At the bottom of the window, an additional label is created to display the total cost of the solution provided. For the steps and total cost, the .pack() method is used to add the two elements to the window and arrange it accordingly.

Moving on to the window for legends, it is also implemented similarly to the above-mentioned. The biggest difference is the list of tuples, where each tuple contains the name of a node type and its corresponding colour. Again, a for-loop is used to loop through the list of legends and unpacks the tuple into its respective name, background colour and text colour. Labels are created to accommodate each legend and the .pack() method is used to add the legend labels to the window and arrange it.

4.0 Evaluation & Discussions

The algorithm seems to be able to find its solution for the least cost path towards collecting all of the treasures while considering the obstacles, rewards and traps. For example, it was able to avoid Trap 4 at (8,2) in order to ensure that the goal state would be able to be achieved. It has also utilized Trap 3 at (5,3) in order to reach Reward 2 at (7,2), reducing the step multiplier after stepping to other nodes. The solution avoided Trap 1 at (8,2) and Trap 2 at (1,1) and at (2,4) which could increase the total cost by modifying the step or energy multiplier. However, going for Reward 1 or Reward 2 can help decrease the total cost by also modifying the multipliers. The algorithm was also able to enter a node twice during the path in order to take the shortest route to another route instead of taking a longer way to avoid entering a node twice.

Figure 4: Number of Explored Nodes

As it is an A* Algorithm, it is important to evaluate the usage of heuristics. The usage of heuristics allowed the algorithm to only expand a total number of 33 nodes without expanding all of the nodes. However, it has to be noted that it could potentially take the same amount of time or more compared to expanding all the notes as extra time is taken to compute the heuristic function in order to sort the function list. If the virtual map were to be bigger, we believe that this algorithm could potentially be faster compared to the other algorithms.

As mentioned above, the algorithm has been modified to ensure that all the treasures can be collected. Based on the output Figure 1 and 2, the algorithm was able to take into account of all of the treasures and collect all of it successfully in the solution. Moreover, it was able to re-enter a node that has been entered before during the path to the previous goal node. For example, Step 10 shows that it has enter (7,3) which is one of the goal nodes. It then proceeds to the other treasure node until Step 14. After collecting the treasure at (4,1), it was able to pass by (7,3) again to collect the treasure node at (9,3). This shows that the algorithm has been implemented in a way to ensure that the previous paths of previous goal nodes does not affect the path to the other goal nodes.

5.0 Reference List

[1] A. J. Patel. “Hexagonal grids.” redblobgames.com. Accessed: July. 16, 2024. [Online]. Available: https://www.redblobgames.com/grids/hexagons/.

[2] “tkinter — Python interface to Tcl/Tk.” python.org. Accessed: July. 16, 2024. [Online]. Available: https://docs.python.org/3/library/tkinter.html#tkinter-modules

[3] GeeksforGeeks. “Python Tkinter Canvas Widget,” geeksforgeeks.org. Accessed: July. 16, 2024. [Online]. Available: https://www.geeksforgeeks.org/python-tkinter-canvas-widget/