COMP30024 Artificial Intelligence – Project Part A – Report

Jiaheng Dong Zhirong Piao

Q1

Overview:

As for the A* search, the main idea is that we explore nodes by the lowest f(n) to find the shortest path, while occupied positions are considered as blocks throughout exploration.

Implementation details:

--- How to choose the current node

--- Explain the data structure (i.e. set) used for data storage in `visit` and `visited`

In order to choose the node with the lowest f(n) at each state, we have to create a set called `visit` to store the nodes which have been generated but not fully expanded. After this node has been fully expanded, we further remove it from the `visit` set and put it in the `visited` set. In this case, we will not take the original state into account when g(n) of the original state is stable.

The reason for choosing the data structure set but not list for `visit` and `visited` is that we consider it to work more analogously as a hash-table, which is more efficient and time-saving when checking, adding or removing its elements. Besides, using a set data structure will reduce the duplicates within it. Although this might not be the main reason for using set as we do add a cell into `visit` after checking this cell is not already in it, this may still be a good advantage to avoid any imperceptible issues.

--- How to show the path

--- Explain the reason for using the data structure dictionary to store state information

As for tracing the path, we use the dictionary data structure to store the parent of the current node. When we have reached the goal state, we trace back the dictionary of parents to find the path from start to goal. Primarily, the structure of dictionary will lead to lower time complexity in finding the parent of the current node and in finding the shortest path.

--- How to find neighbours of the current node

As for exploring the neighbours, we have a function to determine that which coordinates are the neighbours of the current node. A natural cell has six neighbours in this hexagonal board, but any cell adjacent to the boundaries has less neighbours. We calculate and find the neighbours of each current node after some preliminary check of boundary adjacency.

--- How to arrange neighbours, g-values and parents at different situations

The implementation details of adding neighbours consider three conditions.

-- has not been generated and expanded:

We directly add the neighbour into `visit` and assign values to g[cell] and parent[cell].

-- has been generated but not fully expanded:

If the cell is in `visit`, we have to determine whether it is closer to first visit the current node then the cell or not. If it is closer, we update g[cell] to g[curr] + 1 and update parent[cell].

-- has been generated and fully expanded:

We do the same comparison and update as the second situation. Besides, we need to remove the cell from `visited` and add it back to `visit` because g[cell] has been changed, hence f(n) might need to be taken into consideration again.

Time complexity (assuming fair game and the number of all nodes is n^2; we define N := n^2):

--- Dealing with the searching procedure

-- Loop the set `visit` = max(len(visit)) = O(r+q) = O(2*r) = O(r) = O(n) = O(log(N)) (assuming we have a good heuristic function which prevents us from exploring all the nodes)

-- Find the smallest f(n) in `visit` = O(1) * O(log(N)) = O(log(N))

-- Trace back to show the shortest path = O(1)

--- Dealing with operations on neighbours of the current node

-- Find the neighbours of the current node = O(1)

-- Check whether the neighbour has been blocked or not = O(1)

-- Check if it is in `visit` and `visited` = O(1) (average for set)

-- Update = O(1)
-- Update for the neighbours in `visit` or `visited` = O(1)
-- Check whether the cell is in `visited` or not = O(1) (average for set)
-- Remove the cell from `visited` = O(1) (average for set)
-- Add the cell back to `visit` = O(1) (average for set)
-- Remove current node from `visit` = O(1) (average for set)
-- Add current node to `visited` = O(1) (average for set)

--- average Sum of total time complexity = O(log(N)) * O(log(N)) + O(1) + O(1) * O(1) * O(1) + O(1) + O(1) * O(1) * O(1) + O(1) + O(1) = O(log(N)) * O(log(N)) = O(log^2(N))

--- worst case = O(N^2)

Space complexity

-- Visit = O(log(N)) (assuming we have a good heuristic function)
-- Visited = O(log(N))
-- g = O(log(N))
-- Parent = O(log(N))

--- Sum of total space complexity = O(log(N))

Q2

The heuristic function used in the search algorithm is Manhattan distance. Firstly, a z-coordinate is introduced as the inverse of the sum of r- and q-coordinates. Then, the heuristic value is computed as half of the sum of total absolute value differences among all three coordinates. These ideas are sourced from redblobgames.com/grids/hexagons/. An admissible heuristic is defined as some h(n) where its value is always smaller than or equal to the true path cost h*(n). In the case where there is no block on the path, the Manhattan distance is equivalent to the true path cost. In the circumstance where there are blocks on the true route, the Manhattan distance results in smaller heuristic values because it does not take the blocks into account during computation. Thus, this method is admissible.

When applied to the search algorithm, the cost of computing heuristic values is O(1) each time and O(1) * max(len(visit)) = O(1) * O(log(N)) = O(log(N)) in total. Without this heuristic function, we need to explore all n nodes on the board, and have to expand all neighbours of each node to the maximum width of six neighbours. This comes eventually as an O(N)-scaled algorithm, whereas the heuristic function helps reduce the algorithm to a logarithmic scale. Relative to the overall reduction in time complexity, the additional computation complexity generated from using a heuristic function is somewhat negligible, so this method proves beneficial.

The Manhattan methodology is much better as a heuristic function than other candidates such as Euclidean distance. Since the latter only calculates direct distances but fails to incorporate those zigzags in a hexagonal board when estimating path cost, the consequent heuristic values derived might be significantly underestimated. In this sense, Manhattan distance is closer to and is a better representation of the true distance, while the condition of admissibility is also preserved.

Q3

We regard the key point of this question as how we select the current node in `visit` to be expanded. For each node, we can first calculate f(node->goal) by using the heuristic function and the value of g(n) like previously. We then calculate min(f(node->one cell which needs to be captured->goal)) among all provided cells on the board that have the same colour as ours (i.e. are possible to be captured). It means that in this second calculation we have to find values of f(n) of the current node using each cell on the board, and then find the minimum value f(n). After these two calculations, we select min(f(node->goal), f(node->one cell which needs to be captured->goal)) to be the distance of each node. We subsequently compare the distance of each node to determine which node to expand. In this case, we use the heuristic function in three aspects: 1) h(node->goal); 2) h(node->cell to be captured); 3) h(cell to be captured->goal). When we compare the h(n) values, the smallest one always dominates the result, so f(n) will still be optimal and the original heuristic function is still admissible such that h(n) <= h*(n).