COMP30024 PROJECT PART A REPORT

Implementation details

Data structures used for A* search implementation would be a graph and a priority queue. The steps we took for this A* search would be:

1. Starting out from the start node (which would be our root node).
2. Find all neighboring unoccupied nodes with their own evaluation function derived by summing g(n) (cost so far to reach n) and h(n) (estimated cost to goal from n). The algorithm checks duplicate nodes to make sure h(n) is calculated once for each node since h(n) remains unchanged throughout the search. Also, duplicates with equal or worse g(n) are ignored, which helps reducing unneccessary nodes being considered, while duplicates with lower g(n) would replace the existing node in the queue, improving the overall performance of the algorithm.
3. Add each valid neighbor into the queue and sort the queue based on their evaluation function in ascending order.
   - **i)** This way we will always pick the least cost possible path by the time we get to the goal node.
   - **ii)** Python sort() takes O(NlogN), where N is the number of nodes in the queue, however, the duplicates checking performed at step 2 reduces the number of nodes entering the queue, which also reduces overall time complexity of sorting.
4. Then we will pop the first node from the queue for expansion. Since the queue has already been sorted, popping will take constant time.
5. Repeat (2)-(4) until the goal node is found or the queue is empty (no possible paths found).
6. If a goal node is found, find the path from the start node by backtracking the parent of each node from the goal node and reversing the list.
7. If no path is found after going through every possible path, the algorithm will just return a 0 to show that no path was found.

Overall, the time and space complexity would be $O(b^d)$ where d is the maximum depth of the solution, b is the branching factor. With some optimisation discussed above, the overall complexity can be significantly reduced.

Heuristic function

h = Euclidean distance from cell to goal.

Heuristic is admissible because a valid move is to move to one adjacent cell each time to get closer to the goal while the heuristic assumes moving arbitrarily.

The heuristic is computed using Euclidean distance, which takes constant time for each computation. The algorithm checks for duplicates when computing heuristic to make sure every node (cell) is only computed once. Overall, for an n x n board, the cost of computing the heuristic function is O(N) where we define N := n^2. In practice, the cost can be lower since A* biases search toward the goal state and becomes more focused around the optimal path. In other words, it will effectively prune nodes (cells) that are unnecessary for finding the optimal path, reducing the overall heuristic computing cost. This reduction improves the time and space complexity of the algorithm.

Test board:

```
{
   “n” : 5;
   “board” : [
         [“b”, 1, 0],
         [“b”, 1, 1],
         [“b”, 1, 3],
         [“b”, 3, 2]
    ],
    “start”: [4, 2],
    “goal”: [0, 0]
}
```

Number of nodes (cells) considered for heuristic computation: 18 < 25.

Extended algorithm

Initially, the algorithm excludes the existing board pieces from being generated and expanded. Now, they will be included in the generation and expansion process with their fixed cost being 0. The heuristic is still admissible because these pieces are not involved in the heuristic computation.