COMP30024 Project Part A Report

Sunchuangyu Huang[1] & Wei Zhao[2]

April 4, 2022

$A*$ Searching Algorithm: Implementation Details

A^* search algorithm is an informed search algorithm, to implement it, first we construct an open set using a priority queue. The open set will contain next possible expanded nodes, their current f-score (calculated by evaluation function) and expand order to track the number of iteration. After expanding a node, A^* will close the current node (mark it as explored) and continue to expand the next node which has the lowest f-score to find the optimal path.

In the search algorithm, the f-score is calculated by functions $g(n)$ and $h(n)$.

- $g(n)$ is the cost/steps to next node from the current expanding node.
- $h(n)$ is a heuristic function which estimated cost from the current node to the goal.
- $f(n) = g(n) + h(n)$

The f-score will determine the node expanding order that means A^* will always choose the node in the open set which currently has the lowest cost to expand.

While expanding nodes, the first node with lowest f-score will pop out from the priority queue. After chosen the expanding nodes, A^* add neighbours of the node, their f-score, and expand order to the priority queue. This process will terminated until the priority queue is empty (no nodes for expanding, that start point can't reach the goal point), or f-score of the current node is the smallest among all the node scores (that A^* search find the optimal path).

$A*$ Searching Algorithm: Choice of Data Structures

We choose priority queue as the data structure to store the expanding node data. A priority queue can index the data by the target values (f-score) follow the FIFO. Therefore the node with lowest f-score will be expended first since it has the highest priority. This is consistent with the definition of the A* algorithm: a best-first search algorithm.

$A*$ Searching Algorithm: Time/Space Complexity

The time complexity of A^* algorithm is mostly depends on the heuristic function due to the cost of move a node to their neighbors is always 1. We assume that to expend each node, it has a branching factor b and the maximum depth of d. So the time complexity will be $O(b^d)$. The space complexity will be $O(b^d)$ as well since all nodes need to keep in the memory (priority queue).

[1]email: sunchuangyuh@student.unimelb.edu.au, student id: 1118472
[2]email: weizhao1@student.unimelb.edu.au, student id: 1118649

Heuristic Function: What heuristic function we used

In the A^* algorithm, the heuristic equation has the greatest impact on the overall complexity and efficiency of the algorithm, and a suitable heuristic can get more optimal results. Based to Hart, et al. (1968)[3], if A^* is admissible, the heuristic function $\hat{h}(n) \leq h(n)$. And $\hat{g}(n) = g(n)$. By contraction, if a node s has a $h(s)$ and an optimal h-score $\hat{h}(s)$. To satisfied the condition that f is a monotony increasing function. Assuming h is consistent. To increase the f-score, h function must increase because $\hat{g}(n) = g(n)$. Therefore, if $\hat{h}(s) > h(s)$, then $\hat{f}(s) > f(s)$. However, if $\hat{f}$ is increasing, then A^* will never choose to expand the node s since it has less priority, therefore for any given heuristic function, $\hat{h}(s) \leq h(s)$ is satisfied. Since we are exploring nodes in 2D space, we can choose Euclidean or Manhattan distance. Both formula will return an optimal cost $\hat{h}(n) = h(n)$, which satisfy the admissible condition. In part A, when choose Manhattan distance as the heuristic function.

Heuristic Function: Why?

By randomly generating initial cachex board data, we draw the comparison diagrams to show the difference between Euclidean distance and Manhattan distance in terms of node expansion times. By observing figure 1. We found that when the size of dimensions are small ($n < 70$), the performance of the two distance function are basically the same. But as the board size increases, in figure 2, the Manhattan distance formula performs sightly better than the Euclidean. Although this difference may influenced by the random board generation function, but in general Manhattan distance out perform Euclidean distance resulting less node expansion times. We consider the reason for this result is that the two distance algorithms lead to different balance between $h(n)$ and $g(n)$ and further affect the speed of A^* algorithm.

Challenge Question: How would you extend current solution to handle this?

Based on the current solution, we would consider creating a function that determines the $g(x)$ when the path went through the cell with the "color". There is no cost to going through these cells. Therefore the $g(n') = g(n)$ after moving to a color cell. Moreover, we design a new algorithm to resolve this problem. Basic idea is to find the shortest path by going through at least one 'color' cell if there exist any color cells. The final result will be the smaller one comparing to previous A^* algorithm that doesn't consider the color cells.

Challenge Question: Discuss whether the heuristic used originally would still be admissible

We think our heuristic to be still admissible in this problem. Again, according to Hart, et al. (1968)[3], A^* is admissible by achieve $g(n') = g'(n)$, $\hat{h}(n) \leq h(n)$, $h(n)$ is consistence and $f(n)$ has a property monotone increasing. In this question, the cost function $g(n)$ has changed to number of unoccupied cells instead of moving one step from a node to node. From the previous section, we prove that our heuristic function choice will always satisfy $\hat{h}(n) \leq h(n)$. Because $\hat{h}(n) = h(n)$, to increase f-score for next expand nodes, g-score must increase. If $g(n)$ decrease, then $f(n)$ will decrease. In this case, A^* will re-open the next node and potentially stuck in a infinity loop. Therefore, $g(n)$ must increase. In addition, we know that the cost function count the number of unoccupied cells. Therefore, if a colored cell has been selected, the cost of move to the cell be 0, $\hat{f}(n) \leq f(n)$, At the end A^* still guarantee to find the optimal solution because $\hat{f}(n) \leq f(s), \forall s \in N$.

End of Project Part A Report

References

[1] Anindita Das (2022, January, 15) *A* Search is Admissible — Proof*[Video]. https://www.youtube.com/watch?v=uNx7IyxMgS8

[2] Bert Huang (2016, September, 14) *A* Optimality*[Video]. Youtube. https://www.youtube.com/watch?v=iasWgMItGao

[3] Hart, Peter E., Nils J. Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths." Systems Science and Cybernetics, IEEE Transactions on 4.2 (1968): 100-107.

[4] Subbarao Kambhampati (2016, October, 17) *Qn 4: A* search proof of optimality*[Video]. Youtube. https://www.youtube.com/watch?v=UWqykNhcQlQ

[5] Rachmawati, D., & Gustin, L. (2020, June). Analysis of Dijkstra's Algorithm and A* Algorithm in Shortest Path Problem. In Journal of Physics: Conference Series (Vol. 1566, No. 1, p. 012061). IOP Publishing.

[6] Wikipedia. (2022, 23 February). *A* search algorithm*. https://en.wikipedia.org/wiki/A*_search_algorithm

[7] Mila, L. (2022). *Dijkstra's Algorithm vs A* Algorithm*. https://stackabuse.com/dijkstras-algorithm-vs-a-algorithm/

[8] Computerphile (2017, February, 16) *A* (A Star) Search Algorithm - Computerphile*[Video]. Youtube. https://www.youtube.com/watch?v=ySN5Wnu88nE&t=476s

[9] Jhon Levine (2017, March, 27) *A* Search*[Video]. Youtube. https://www.youtube.com/watch?v=6TsL96NAZCo&t=351s

[10] Tech With Tim (2020, July, 17) *A* Pathfinding Visualization Tutorial - Python A* Path Finding Tutorial*[Video]. Youtube. https://www.youtube.com/watch?v=JtiK0DOeI4A&t=4873s

[11] Amit, P. (2022) *Amit's Thoughts on Pathfinding*. http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

Figure

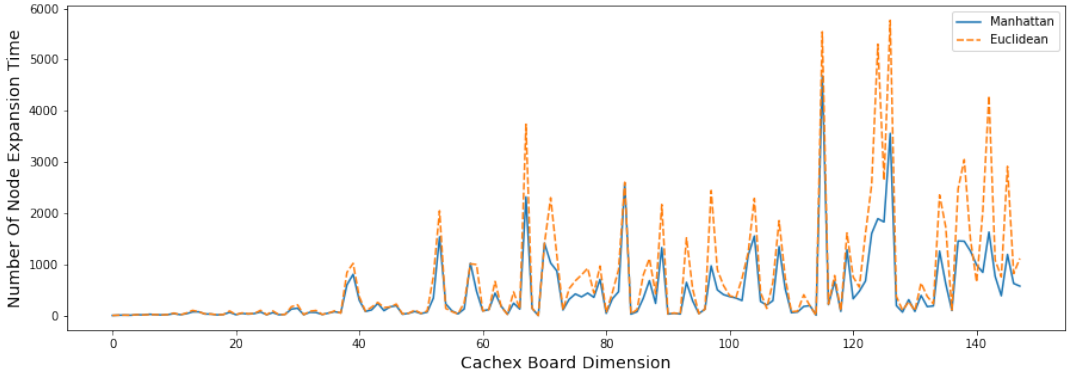

Figure 1: Random board generation with dimension range 2 to 150

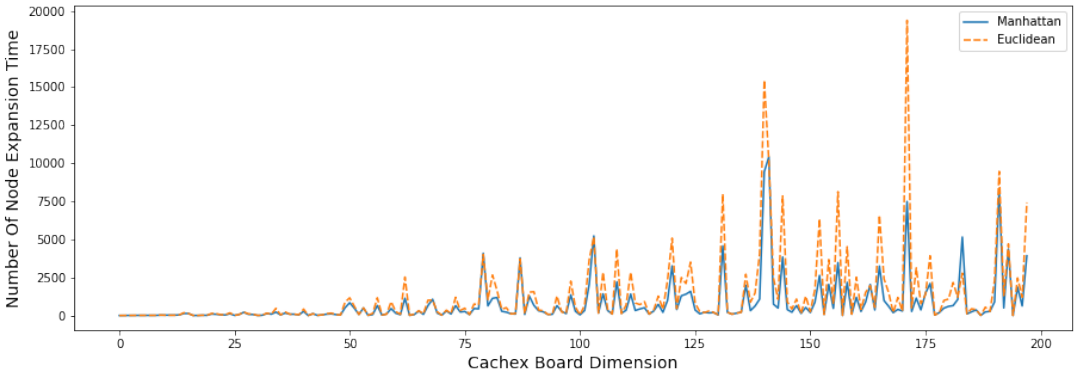

Figure 2: Random board generation with dimension range 2 to 200

A* Searching Algorithm

```
def AStar(self, start=None, goal=None, heuristic='manhattan', p=None, block=None):
    if start or goal not in the game board, raise error

    Define the required priority queue, g_score, f_score, h_score for each node
    priorityQueue = PriorityQueue()
    AStarScores = dict{node for node in board if a node has a state none
                       or node has a state which not equal specified block}

    Store explored nodes & define a queue tracker to track items in priority queue
    explored, queueTracker, order, path = dict(), {start}, 0, []

    # initialise the priority queue
    priorityQueue.put([0, order, start])

    # initialise the start state with cost 0 and
    # distance difference based on given heuristic function
    AStarScores[start].g, AStarScores[start].h = 0, distance_difference(start, goal)

    while priority queue. is not empty:
        currentNode = pop the first item in priority queue
        queueTracker.remove(currentNode) # untrack the current node

        if currentNode reach the goal:
            return search path

        for node in currentNode's next nodes:
            CASE 1: block type is given
            if block is not None:
                if self.NodeDict[nextNode].state != block:
                    if currentNode g cost + 1 greater than nextNode g cost:
                        update the nextNode g, h, f scores
                        explored[nextNode] = currentNode # update path history
                    if nextNode is untracked (not in queueTracker):
                        put nextNode in priority queue and queueTracker

            CASE 2: block type is not given
            if nextNode has a state None:
                if currentNode g cost + 1 greater than nextNode g cost:
                    update the nextNode g, h, f scores
                    explored[nextNode] = currentNode # update path history
                if nextNode is untracked (not in queueTracker):
                    put nextNode in priority queue and queueTracker

    return empty path if no path found
```

Algorithm designs for challenge question

```
def update_algorithm(start, goal):
    # the array that store the nodes needed to go through
    route = []

    # flag of status
    arrive_goal = False

    #find the closest 'color' node from the starting node
    new_node = find_closest_node_color(start, color_nodes);

    # update current node
    current_node = new_node

    #append the node to array
    path_queue.append(current_node)

    # the array for shortest path result
    path = []
    while( arrive_goal != True):
        # find next color node with lower estimate cost to goal than current node
        temp_color_nodes = [node for node in color_nodes
                    if color node with lower estimate cost to goal than current node]

        # find a new color node
        if (temp_color_nodes is not empty):
            new_node = find_closest_node_color(current_node, temp_color_nodes);
            if g(new_node) < g(current):
                route pop out all nodes
                route append new_node
            else:
                route append new_node
            current_node = new_node

        # no color node could be found
        else:
            #add the goal to route array
            arrive_goal = True
            route.append(goal)
    for node in route:
        # get shortest path from previous A* algorithm between two nodes
        path += Astar(starting, node)

        # update starting node
        starting = node
    return path
```