COMP30024 Project Part A Report

Sunchuangyu Huang[1] & Wei Zhao[2]

April 5, 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 put it in the close set) 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 function $g(n) + h(n)$.

- $g(n)$ 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.

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 (highest priority) to expand.

While expanding nodes, the first node with lowest f-score will pop out from the open set. After that, A^* add its neighbours with their f-score, and expansion order to the open set. This process will terminate until the open set is empty (no nodes for expanding, that start point can't reach the goal point), or current node reach the goal state and its g-score 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 store the data with order by target values (f-score). 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 mostly depends on the heuristic function due to the cost of moving a node to their neighbors is always 1. We assume that to expend a node, it has a branching factor b and the maximum depth 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 (open set/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 A^* algorithm, the heuristic equation has the greatest impact on the overall complexity and efficiency, and a suitable heuristic function guarantee A^* always return a optimal path if a solution exists. Based to Hart, et al. (1968)[3], if A^* is admissible, the heuristic function $\hat{h}(n) \leq h(n)$[1], while $\hat{g}(n) = g(n)$. By contraction, if a node s has a score $h(s)$ and an optimal score $\hat{h}(s)$. To satisfied the condition that h is a monotony increasing function, h must be 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. A^* will never choose the sub-optimal solution as the final answer. Therefore for any given heuristic function, $\hat{h}(s) \leq h(s)$ must be satisfied. Since we are exploring nodes in a 2D space, we can choose Euclidean or Manhattan distance functions. Both functions will return an optimal cost $\hat{h}(n) = h(n)$ that $h(n)$ never overestimate the cost, which satisfy the admissible condition. In part A, we 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 and Manhattan distance functions in term of node expansion times. In figure 1, we found that when the size of board dimension is small (about $n < 70$), the performance of two distance functions are basically the same. But as the board size increase, in figure 2, Manhattan gradually performs better than the Euclidean. Although this difference may influenced by the random board generating function, in general Manhattan out perform Euclidean in most cases. We consider the reason is if convert a cachex board and hexagon cells into a 2D space, a node cannot move along the board major axis with cost 1. As the result, if a node wants to move to along the major axis, Euclidean will expand the node at least twice but Manhattan only expand node ones since Manhattan knew it will cost at least 2 steps if we want to move along the major axis.

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

Based on the current solution, we would expand the current A^* function by considering all colored cells and goal point are sub-goals. The program[2] will continue explore these nodes using existing A^* function until the optimal solution found where the solution is also the optimal sub-optimal solution (reach the goal state with the minimum unoccupied cells). While expanding nodes, if neighbors are colored node, the path cost will be 0, otherwise it will be 1. In worst case the algorithm have the similar time complexity $O((n+1)b^d)$ but space complexity increase to $O(b^{d(n+1)})$ where n is the number of colored nodes. But generally A^* should terminate since nodes have higher f-score won't be expanded.

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 consistent and $h(n)$ 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. This means when move a node to another node, the cost could be 0 or 1, that means $g(n)$ will never decrease. From the previous section, we prove that our heuristic function choice will always satisfy $\hat{h}(n) \leq h(n)$. In this problem, $h(n)$ is the same as the previous, as a result $\hat{h}(n) = h(n)$ that still satisfied the admissible condition.

[1] $\hat{h}(n)$ standand for the optimal cost from the current node to the goal point

[2] Challenge question pseudo code is available on reference page

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

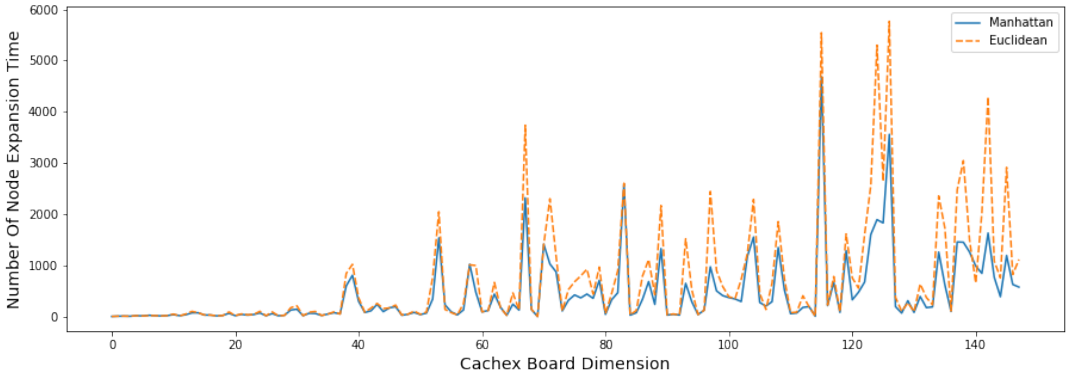

Figure 1: Random board generation with dimension range 2 to 150

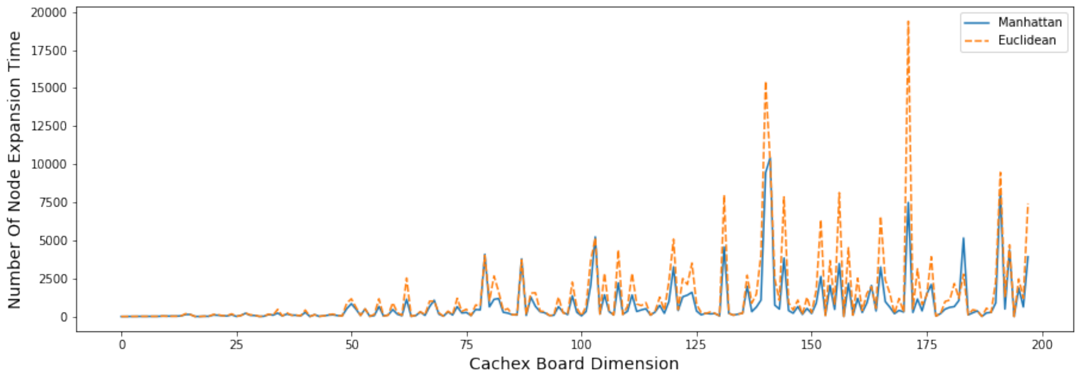

Figure 2: Random board generation with dimension range 2 to 200

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

    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

    Put start point state [0, insert_order, start] in the priority queue
    AStarScores[start].g = 0
    AStarScores[start].h = distance_difference(start, goal)

    while priority queue is not empty:
        currentNode = pop the first item in priority queue
        remove currentNode from queueTracker

        if currentNode reach the goal:
            return search path

        for node in currentNode neighbors:

            CASE 1: block type is given
            if block is not None:
                check current cell has a state == block:
                    if currentNode (g + 1) greater than nextNode g cost:
                        update the nextNode g, h, f scores
                        update path history for next node
                    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
                    update path history for next node
                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

```
Step 1 Detect the existing stick
Detect the number of sticks where sub-goal nodes are connect to each other.
If a point has no connection between other points, also consider it as a stick
with length 1

Step 2 Find the Optimal Solution to Each Stick
initialise priority queue open set, explored close set, queueTracker
for stick in stickList:
minimum h_score = minimum distance from the start point to any node in stick
 f_score = h_score # since start point has a cost 0
         put [f_score, g_score=0, h_score, start] in priority queue

while priority is not empty
current explored point = minimum f-score item in priority queue

for node in current explored node neighbors:

if goal in current expanding stick or current expanding node is goal:
return path

CASE 1: current point neighbors not in priority queue
    then add them in priority queue

CASE 2: current point neighbor is a point
put the point f-scores to each stick to priority queue

CASE 3: current point neighbor in a stick
update the stick scores and put it back in priority queue
note: if expand a stick, the algorithm should expand the node
      which provides the lowest f-score

if goal in a stick while the stick has the lowest f-score and g-score:
return path
```