ME5751 Lab 3 – A* Planning

Ryan Chao
Cal Poly Pomona
Pomona, CA
rchao@cpp.edu

Cameron R. Weigel
Cal Poly Pomona
Pomona, CA
crweigel@cpp.edu

Abstract— **In this lab, A* Search for path planning is utilized to generate an optimal path from an initial start position to a designated point through the combination of a Euclidean distance heuristic with an 8-connection neighbor search and Brushfire cost mapping for individual cell cost. Several issues faced throughout the module were long compute times and obstacle interference that led to incomplete paths after a maximum iteration condition was met. Compute times ranged from a tenth of a second to half an hour depending on the complexity of the map, with some not completing. The majority of the issues arised from paths around large or long obstacles which would cause the algorithm to oscillate between points near obstacles due to similar cost values. With enough time, this algorithm may have found a path around the objects but for practicality, early stopping was employed.**

Keywords— A* algorithm; Dijisktra's algorithm; Greedy best first; mobile robotics; robot motion planning; heuristic function

I. INTRODUCTION

In order to establish proper robot navigation, several operations must be incorporated which are self-localization, path planning, and map building. Luckily, working in a simulated environment allows for ease in determining localization and map building as the robot will be represented in 2D space. Path planning will be the main focus for this module in which the A* algorithm is utilized. The A* algorithm is the most commonly used heuristic search method developed from the Dijkstra algorithm which searches for the best path between nodes in a given map. Due to Dijkstra's algorithm's inefficiency which stemmed from long computation time, A* was created as a "modified" version of Dijkstra's, utilizing Dijkstra's simplicity with an added heuristic function. The A* algorithm performs better in that it is capable of determining the best nodes to explore first due to its heuristic function, which cuts down computation time greatly. This is done by computing a node's cost using the potential cost and a heuristic value of a cell. Tuning the heuristic value is required in order to increase the algorithm's efficiency and will be the main subject of this module.

II. BACKGROUND

Robot path planning saw immense improvements from Edsger W. Dijkstra's 1956 paper where he defined Dijkstra's search algorithm. Initially the algorithm sought to solve for the shortest path between two nodes, but a variant was developed which would search for the shortest path from a source cell to neighboring cells. The major flaw in Dijkstra's algorithm was its blind search method which consumed lots of time.

Around a decade later, in 1968, the A* Search algorithm was established by the Stanford Research Institute as an extension of Dijkstra's algorithm. This algorithm achieved better performance by using a heuristic function to aid in its search. It focused on finding the shortest path between a starting point to a goal point by exploring the "best" paths first. To determine the next path iteration, surrounding nodes of the current node would be assigned cost values through summing the cost to move to the next node with a heuristic function such as Euclidean distance.

An alternative heuristic approach is the Greedy Best First search which expands towards the smallest heuristic. Although this allows for the quickest path towards the goal point, it does not work all the time in the case there are obstacles which causes the algorithm to go off track in its bias for diagonal traversing. Therefore, the A* algorithm is the focus for this module as it can be seen as a fusion between Dijkstra's and Greedy Best First as it has a balanced search quality to search speed.

III. METHODS

In the A* search method, a cost function is implemented based on cost to travel to each cell and its distance from the goal cell. The algorithm then searches through the map from the start using a *best-first* search method. This *best-first* search method expands first to the nodes that have the most promising value based on its associated cost function. In A*, the cost function is built from two functions, one that describes the cost of travel up to the current node $g(n)$, and the other is usually a distance function from the current node to the goal node $h(n)$. The A* search algorithm we developed utilized a modified $g(n)$ that was the running sum-total of cost values derived from the associated cost map, and a scaling value d, that favors moving away from one cell that has been visited multiple times. This modified $g(n)$ was proposed as a method of handling situations where the algorithm would oscillate on tiles next to walls with the goal being directly on the other side of the wall.

$$g_k(n) = \sum_{i=0}^{n} p_i + costmap(k_x, k_y) * d_j$$

$$where\ p_i\ is\ the\ i^{th}\ parent\ of\ k$$

$$d_j = 1, \ldots N$$

In practice, the function looks as follows.

```
# Cost of child is the cost to get here plus a
scaling value based on how many times we've
visited this cell
# This was done in hopes to move around walls to
combat the euclidean distance dominating the total
cost
child.g = child.parent.g +
self.costmap.costmap[child.pose.map_i,child.pose.m
ap_j]*d[child.pose.map_i,child.pose.map_j]*10
```

Our algorithm uses the 8-connection neighbor search with heuristic function $h(n)$, equal to the Euclidean distance function of the current node to the goal node.

$$h(n) = \sqrt{(x_{goal} - x_{current})^2 + (y_{goal} - y_{current})^2}$$

This function was chosen as our heuristic because of our use of the 8-connection neighbor search method that allows diagonal movements, [7]. The 4-connection neighbor search method was tested with a heuristic $h(n)$ that utilized the manhattan distance function.

$$h(n) = \left|x_{goal} - x_{current}\right| + \left|y_{goal} - y_{current}\right|$$

Both methods proved useful, but we decided to use the 8-connection and Euclidean distance method due to the optimized paths traveling along the diagonal. Our heuristic is implemented as follows.

```
# Euclidean distance because we are
using an 8-connection style
child.h = self.heuristicEuc(child.pose,
goal_node.pose)
```

Finally, a scaling factor $\boldsymbol{\mu}$, is used to tune the heuristic. The total cost $f(n)$ is then described as the sum of both costs with scaling factor $\boldsymbol{\mu}$.

$$f(n) = g(n) + \mu * h(n)$$

Our method utilizes a simple Node class developed to build a tree of connected nodes and their associated costs. This Node class consists of a *parent* class variable, a *pose* class variable which is itself an object of type *Pose* defined in the supporting file “Path.py”, three cost class variables *g, h,* and *f*, and an overloaded equals(=) operator to compare nodes. This Node class simplifies the process and understanding of the A* search algorithm and path reconstruction mainly due to its use of its *parent* class variable. This variable is used to store the parents and the parents’ parents and so on, of the current node. From this, we can not only easily calculate total cost to travel to the current node, we can use the helper function *return_path(node)* to travel back up the tree and reconstruct our path.

```
# Simple node class to build parent/child tree, store
poses and costs
class Node:
   def __init__(self, parent=None, pose=None):
       self.parent = parent
       self.pose = pose

       self.g = 0
       self.h = 0
       self.f = 0

       def __eq__(self, other):
           return self.pose == other.pose
```

Another priority of our method was to reduce computation time so that our path could be calculated and displayed as quickly as possible. The largest improvements in computation time came from replacing expensive list searches with fast and efficient array accesses. Due to the grid-structure of this project’s map, it was very straightforward to implement several grids for checking visited nodes, nodes within the queue, total cost to travel to a node, and how many times the node has been visited before. The drawbacks to this method are storing several arrays in memory. Since all of these helper-arrays are constructed using Numpy’s *np.zeros()* method, all elements are intialized to uint8 type elements. In the case of a 500x500 size map, each array is approximately 2MB in size. Furthermore, since most elements are not accessed or modified outside of the uint8 value range, this is a reasonable estimation for the added memory usage of this method. In total, four helper-arrays are used, totaling 8MB extra RAM required, which is negligible on modern computers.

A last but important feature to our method is a stopping condition in case the algorithm fails or would take prohibitively long. From our investigations, there were issues with goal points assigned on other sides of walls with a relatively long path required to navigate around the wall. These cases will be discussed in depth in the next section. In order to prevent the program from running indefinitely or for an extremely long time, a stop condition was put in place. This stop condition can be adjusted to test changes to the algorithm by changing the number of iterations the program is allowed to run before exiting. One iteration can be described as one check of the current node’s cost value compared to all nodes currently in the list, find and calculate its 8 surrounding neighbor’s cost values and add them to this list if they meet the conditions.

IV. RESULTS AND DISCUSSION

The results of the experiments will consist of testing two maps with varying heuristic mu values and a set goal point. The mu-values will be trialed at 0.5, 1, and 5 while the goal point for map 1 will be (233,92).

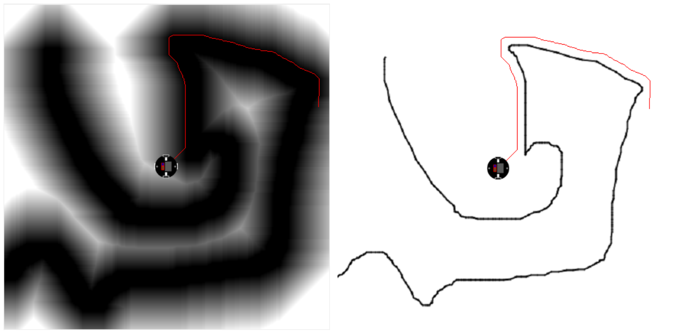

Fig 1. A potential map 1 (left) and normal map 1 (right).

In Figure 1, map 1 was tested using a variety of μ-values which all ended up with the same result. The time it took to reach the goal point from the center starting point was 25.61 seconds for a μ of 1, 26.47 second for a μ of 0.5, and 25.13 seconds for a μ of 5 which can be considered negligible. The time error tolerance for this case is about 3% from the average time of 25.74 seconds. As for the path lengths from testing the μ-values, it was found that all three μ-values shared a common length of 472 units. Thus, we can make the conclusion that μ is near negligible in computing time and path length.

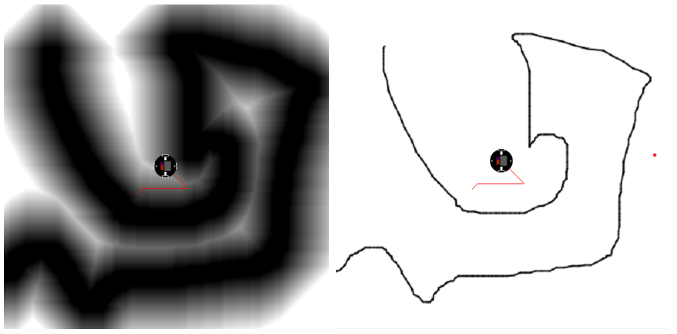

Fig 2. A potential map 1 (left) and normal map 1 (right).

During our trials, we ran into difficulty with goal points that were near horizontal to the starting point and obstructed by large obstacles. The result of this can be seen in Figure 2 where the goal point is set at (233,-34) where the red dot is roughly positioned. For μ-values of 0.5, 1, and 5, the elapsed time was 41.23 seconds, 39.69 seconds, and 39.25 seconds with a path length of 111 units. In this case, the computation timed out due to exceeding an iteration limit of 15,000. From our investigation, the primary reason this path does not work in a reasonable amount of time is based on the heuristic function favoring nodes that are closest to the goal, even when those nodes are on the other side of an obstacle. In order to combat this, we increase cost on nodes that have been visited more than a set number of times. While this could be effective in producing a valid path eventually, it is not a reasonable solution since computation time could take 30 or more minutes. In other words, when the algorithm reaches a node with local minimal Euclidean distance, it favors this node over moving around the obstacle which would lead to a temporary higher Euclidean distance cost. To better visualize this, imagine the cell's potential cost playing tug-of-war with the heuristic function's cost. There are several methods to combat this issue, one method that could be explored in future implementations would be map reduction to key waypoints around large obstacles. Below is courtesy of Red Blob Games [4].

"Waypoints are the key decision points where you might have to change direction. In the above diagram, I mark places where you have to go around a corner or wall. I call these "exterior corners".

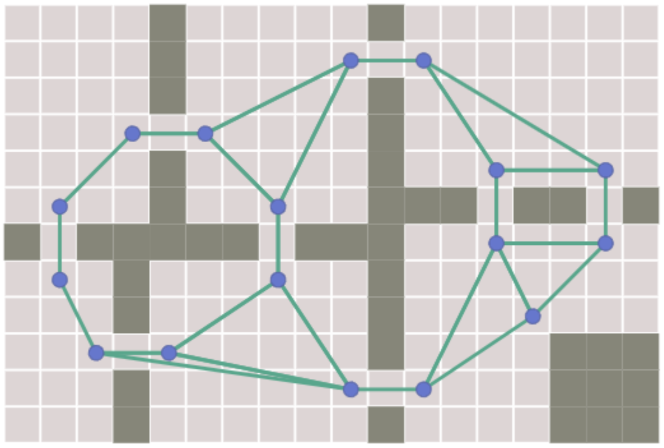

Fig 3. Courtesy of Red Blob Games, Map reduction based on waypoints

For the map in Figure 4, the algorithm did not seem to have any trouble reaching points behind certain large objects. The goal point in this case is (233,40) with the same set of μ-values. Similar results as testing for map 1 occurred where all three μ-values shared roughly the same elapsed times which came out to be 3.14 seconds for μ of 1, 3.13 seconds for μ of 0.5, and 3.24 seconds for μ of 5. From these results, any time difference is highly likely caused by human error. The length of the path is 282 for all cases with this map.

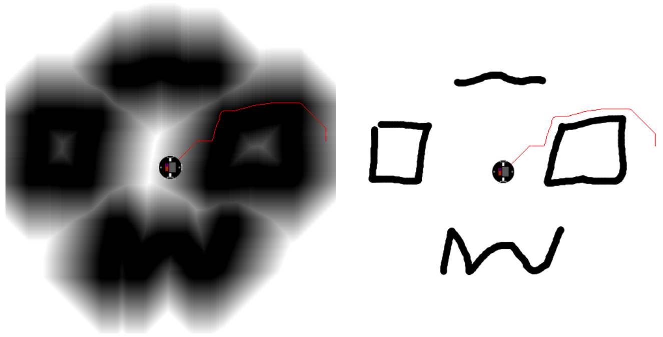
Fig 4. A potential map 2 (left) and normal map 2 (right).

Comparing maps 1 and 2, it can be established that time and path length are not significantly affected by the μ-value. However, the heuristic function experiences complications when trying to wrap around obstacles that cause the horizontal and vertical distance to have an inverse relationship with one another. The heuristic part of the A* function causes the algorithm to oscillate or repeat a series of points. When roughly horizontal or vertical to the goal point (local minimum Euclidean distance) and against an obstacle, the neighboring cells experience drastic changes in their cost value. In order to prevent this, a condition must be met to override the algorithm and allow it to continue along the obstacle regardless of the cost.

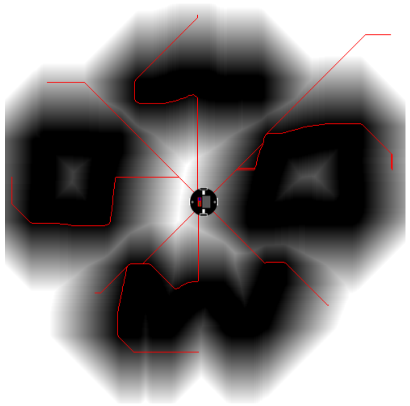
Fig 5. A potential map 2 (left) with multiple goal points.

Figure 5 demonstrates a test where goal points were triggered via right-clicking an arbitrary position on the map. The path length and elapsed time for each goal point can be found in Table 1.

Relative Position	Elapsed Time (s)	Path Length
Right	~ 0.10	278
Up Right	~ 0.10	239
Up	~ 1.00	313
Up Left	~ 0.10	195
Left	~ 0.25	321
Bottom Left	~ 0.10	136
Bottom	~ 8.00	372
Bottom Right	~ 0.10	155

Table 1. Summary of results for Figure 5.

V. CONCLUSION

The A* Search method is a valuable tool to understand for robot motion planning as it is relatively simple to implement, can be extremely fast to compute and with proper heuristics, the algorithm can compute optimal paths. For many cases where objects were within some bounded size, our algorithm performed its path generation well under one second and with paths that looked to be optimal. Optimality could be controlled based on how we vary our cost function and its scalar μ. We found drawbacks to our implementation of the A* search method especially around long or large obstacles where a significant heuristic cost must be incurred to move around the object. With enough time, we believe our algorithm would find paths through a brute force elimination of those nodes that are local minima of the Euclidean distance. Unfortunately, the computation time to complete those paths is unreasonable for any robotics motion planning software. Further research should be focused on heuristics that combat this situation as well as an audit of the entire program for errors in the method implemented.

VI. REFERENCES

[1] R. Siegwart, I. Nourbakhsh, D. Scaramuzza, Introduction to Autonomous Mobile Robots, 2nd ed., The MIT Press : Cambridge, 2011.

[2] Y Chang, California Polytechnic University, ME5751 Lecture Slides, Fall 2022

[3] H Choset, CMU CS Lecture Series, Robotic Motion Planning, Robotic Motion Planning: A* and D* Search

[4] A Patel, Red Blob Games, Implementation of A*

[5] N Swift, Easy A* Pathfinding, Medium webpage

[6] Wikipedia, A* search algorithm

[7] Geeks for Geeks, A* search algorithm