Module 3 : Path planning with A* search

Johanna Martinez-Felix
California Polytechnic University, Pomona
Mechnaical Engineering Department
Pomona, California, United States
johannafelix@cpp.edu

***Abstract*— In this module we implement A* search in order to find the most reasonably shortest path from a start point to a goal point. In order to implement A* search we need to calculate the cost to travel to the neighboring nodes and selects the lowest cost value. To get this "cost" it is necessary to find the value of the shortest path (distance) from start nodes to the neighboring nodes. It most important and necessary to find the heuristic approximation of the nodes value. The heuristic approximation is what the A* concept is based on. Establishing a heap based priority queue will also be instrumental in the implementation of A*.**

Keywords—component, formatting, style, styling, insert (**key words**)

I. Introduction (*History*)

Pathfinding has evolved over the years. In this course, we have covered basic pathfinding algorithms, such as depth-first algorithm and breath-first algorithm applied on brushfire. The two basic algorithms function by going through all path possibilities and run-on linear time. More sophisticated search algorithms eliminate paths in a strategic manner. Such as Dijkstra algorithm. The algorithm begins by finding node candidates and marking them. It examines the lowest distance nodes first so the first route found will have the shortest path. It's down fall it that it cannot evaluate negative edge weights. A slightly improved search algorithm to Dijkstra is the A* algorithm, often called an extension of Dijkstra.

A* algorithm is called an extension of Dijjstra because it functions the same but improves the behavior by finding the heuristic. Heuristic represents minimum distance between a node and the goal and is used to find approximate distance. The quality of A* search heavily relies on the heuristic. Unlike Dijkstra A* only finds the shortest path from a given start to a given goal.

A* search was developed in 1966-1972 for the purpose of a robot to plan its own path as part of a larger project called the Shakey project. A* functions by gathering as much information as necessary in order to make a decision on the best course of action to take. This is called an informed decision; the algorithm has information on more grid points, from start, than the number of grid points it ends up using to get to the goal point. From this, we can get a glimpse of artificial intelligence. The invention of A* algorithm was one of the most notable results of the Shakey project. Shakey received media attention and, among other honors, has the AI Video Competition, "Shakeys", awards named after the project.

II. Method Description

A. *Heuristic value*

A heuristic value is way to direct the search to a node that will lead to a goal. It is to further support and refine the informed method of making an decision. In A* search the heuristic value is needed to find cost "f(n)" as shown in equation 1. Heuristic value of the nodes "h(n)" is added to reference cost ''g(n)".

$$f(n) = g(n) + h(n) \qquad (1)$$

There are three common methods for solving for heuristic value. We will take a brief look at Manhattan, Diagonal and Euclidean.

1. **Manhattan** distance heuristic application equation:

$$h = abs\ (current\_cell.x - goal.x) + abs\ (current\_cell.y - goal.y)$$

   The down fall of this method it that it can only move in 4 directions. (North, South, West, East)

2. **Diagonal** distance heuristic equation:

$$h = D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)$$

   Where,
   $$dx = abs(current_{cell}.x - goal.x)$$
   $$dy = abs(current_{cell}.y - goal.y)$$
   and D is length of each node(~1) and D2 is diagonal distance between each node (~ sqrt(2)). This method is strictly for moving in eight directions
3. **Euclidean** Method equation:
   $$h = sqrt\ (\ (current_{cell}.x - goal.x)^2 + (current_{cell}.y - goal.y)^2\ )$$
   Here, the number of neighbors we can move to does not matter.

B. *Priority Queue*

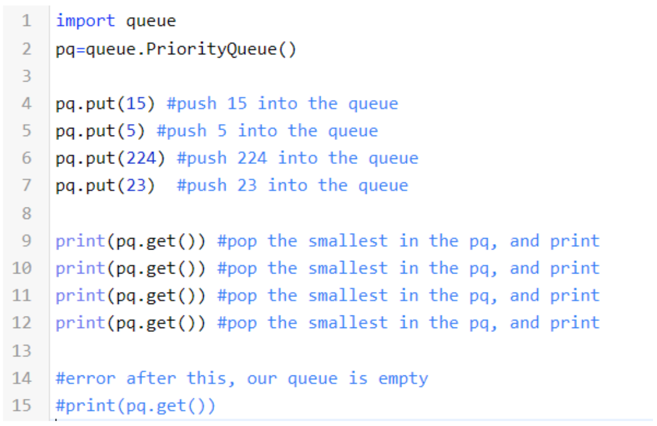
Of course, our goal is still to find the shortest path, so the lowest cost path needs to be found, for this we need to implement a priority queue. A priority queue is similar to a data stack structure with priority, an abstracts data type. We need this so that we only search cells that we deem worth of being searched. For example, a lower costing cell will have higher priority over a higher costing cell. When the algorithm encounters two cells with equal priority, priority queue will have the algorithm search first the cell that was queued first.

The implementation of priority queue can be accomplished with a heap queue. Heap is a tree-based data structure. The highest priority element is stored in the root, which is at the top of the heap. This means that the root is a parent but has no parents. This will be helpful when because it will be necessary to remove elements or nodes from our list of potential paths to take. The root element will be changing as we proceed with the search.

Heap queue algorithm is also known as the priority queue algorithm and can be implemented through pythons imported library of queue. The heap function is already in that queue function. Looking at the code bellow, figure1 and figure2, implementation of queue and heap we can get a better understanding of how priority que work and how to use it in the A* algorithm.

```
import heapq
import itertools

unorderable_data = {"foo": "bar"}

heap = []
counter = itertools.count()

entry1 = (2, next(counter), unorderable_data)
entry2 = (1, next(counter), unorderable_data)
entry3 = (2, next(counter), unorderable_data)

heapq.heappush(heap, entry1)
heapq.heappush(heap, entry2)
heapq.heappush(heap, entry3)

priority, tie_breaker, data = heapq.heappop(heap)

print(priority, data) # prints 1, {"foo": "bar"}
```

Figure 1. Understanding heap , heappq.



```
import queue
pq=queue.PriorityQueue()

pq.put(15) #push 15 into the queue
pq.put(5) #push 5 into the queue
pq.put(224) #push 224 into the queue
pq.put(23)  #push 23 into the queue

print(pq.get()) #pop the smallest in the pq, and print
print(pq.get()) #pop the smallest in the pq, and print
print(pq.get()) #pop the smallest in the pq, and print
print(pq.get()) #pop the smallest in the pq, and print

#error after this, our queue is empty
#print(pq.get())
```

Figure 2. Understanding and implementing priority queue.

C. *A Star Search Concept*

Now that we understand heuristic value and priority queue, we can start looking at how they play a role in the star search algorithm. For A* description of methos we can look at figure 3 from Red Blob Games.

We begin by identifying our "graph" (map), start point and end point. Calling priority queue and setting up variables to help us keep track of current nodes, cost, parents and so on. We place our start node in the queue as we run through the program the queue will contain the past node information while placing the node with the lowest cost at the top of the data list until the data list is empty. In this while loop is also a positional check. It pushes the search to end when the current node is the goal node. This means it has completed finding the shortest path.

This while loop is also determining which neighbors are worth searching by weighing their worth against each other. As mentioned previously this is what allows us to cut off unnecessary searching of other nodes. It adds the cost it has found from the start not to the current node it is on with the cost of the neighbor it has found worth exploring. To reiterate it selects the neighbor worth exploring by implementing queue, which is prioritizing the lowest cost node. This lowest cost node, or worthy node then become the new current. The algorithm keeps track of the cost is has accumulated as it moves from node to node using cost so far in figure 3.

```
def a_star_search(graph: WeightedGraph, start:
Location, goal: Location):
    frontier = PriorityQueue()
    frontier.put(start, 0)
    came_from: dict[Location,
Optional[Location]] = {}
    cost_so_far: dict[Location, float] = {}
    came_from[start] = None
    cost_so_far[start] = 0

    while not frontier.empty():
        current: Location = frontier.get()

        if current == goal:
            break

        for next in graph.neighbors(current):
            new_cost = cost_so_far[current] +
graph.cost(current, next)
            if next not in cost_so_far or new_cost <
cost_so_far[next]:
                cost_so_far[next] = new_cost
                priority = new_cost + heuristic(next,
goal)
                frontier.put(next, priority)
                came_from[next] = current

    return came_from, cost_so_far
```

Figure 3. A* Conceptual pseudocode

D. *Path Reconstruction*

A second while loop is implemented to reconstruct the shortest path of nodes taken to get to our goal. The pseudo code shown in figure 4 reads; while we are searching, save the current node in a list of nodes taken; the past nodes that are parents that lead to the current node. Implemented, we can say while our goal is our current not and the search has ended we can reverse the nodes taken from goal to start in order to show our result; our shortest path.

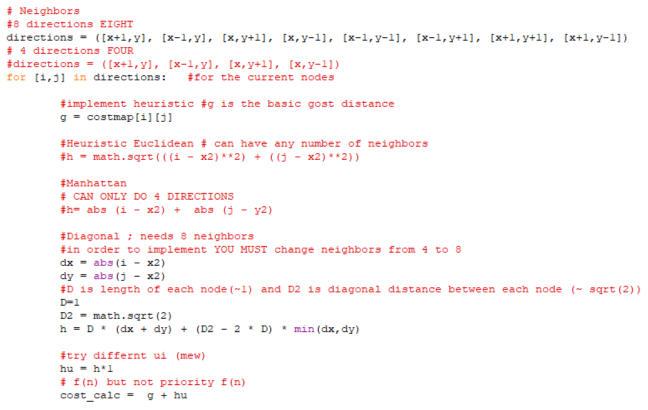
```
while current != start:
   path.append(current)
   current = came_from[current]
path.append(start) # optional
path.reverse() # optional
return path
```

Figure 4. Reconstructing path

III. Method Implementation and Results

A. *Implementation*

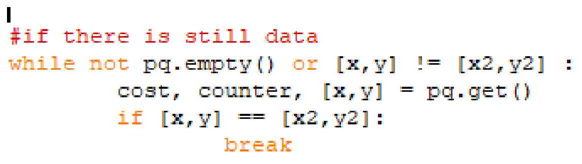
In this module heuristic was implemented in the while loop rather that called from its own function. Figure 5 depicts the implementation of heuristic equation in the code. The set of 4 and 8 neighbors are commented out depending on which heuristic method will be used. As mentioned in method description some heuristic methods have limitations. Start nodes are [x1,y1], goal nodes are [x2,y2], current notes are [x,y].



```
# Neighbors
#8 directions EIGHT
directions = ([x+1,y], [x-1,y], [x,y+1], [x,y-1], [x-1,y-1], [x-1,y+1], [x+1,y+1], [x+1,y-1])
# 4 directions FOUR
#directions = ([x+1,y], [x-1,y], [x,y+1], [x,y-1])
for [i,j] in directions:   #for the current nodes

        #implement heuristic #g is the basic gost distance
        g = costmap[i][j]

        #Heuristic Euclidean # can have any number of neighbors
        #h = math.sqrt(((i - x2)**2) + ((j - x2)**2))

        #Manhattan
        # CAN ONLY DO 4 DIRECTIONS
        #h= abs (i - x2) +  abs (j - y2)

        #Diagonal ; needs 8 neighbors
        #in order to implement YOU MUST change neighbors from 4 to 8
        dx = abs(i - x2)
        dy = abs(j - x2)
        #D is length of each node(~1) and D2 is diagonal distance between each node (~ sqrt(2))
        D=1
        D2 = math.sqrt(2)
        h = D * (dx + dy) + (D2 - 2 * D) * min(dx,dy)

        #try differnt ui (mew)
        hu = h*1
        # f(n) but not priority f(n)
        cost_calc =  g + hu
```

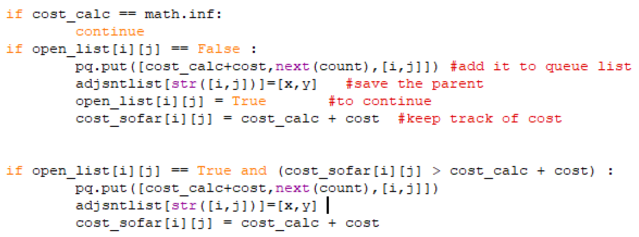
Figure 5 Heuristic code

A* search while loop functions with queue implemented to conduct priority-based search shown in figure 6 and 7. Figure 6 is the set up for the search. It says, if our queue list is not empty and our current node is not our goal run this while loop. It our current node is our goal stop this while loop. Figure 7 is implemented to move from one priority node to the next while tracking cost and nodes we have searched.



```
#if there is still data
while not pq.empty() or [x,y] != [x2,y2] :
        cost, counter, [x,y] = pq.get()
        if [x,y] == [x2,y2]:
                break
```

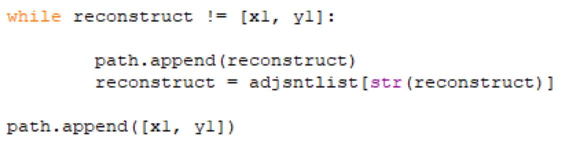
Figure 6. A* while loop set up



```
if cost_calc == math.inf:
        continue
if open_list[i][j] == False :
        pq.put([cost_calc+cost,next(count),[i,j]]) #add it to queue list
        adjsntlist[str([i,j])]=[x,y]   #save the parent
        open_list[i][j] = True        #to continue
        cost_sofar[i][j] = cost_calc + cost  #keep track of cost


if open_list[i][j] == True and (cost_sofar[i][j] > cost_calc + cost) :
        pq.put([cost_calc+cost,next(count),[i,j]])
        adjsntlist[str([i,j])]=[x,y]
        cost_sofar[i][j] = cost_calc + cost
```

Figure 7. move, calculate and track

Path reconstruction is prompted when the current node is our goal node. This is the second while loop, we call up the parents we have been saving from our main A* search loop. These notes have been saved in a path appendix which has the end point and now that we have reached out goal we move backwards using our parent nodes to get to the start nodes, (x1,y1).



```
while reconstruct != [x1, y1]:

        path.append(reconstruct)
        reconstruct = adjsntlist[str(reconstruct)]

path.append([x1, y1])
```

Figure 8. Reconstructing the path

B. *Results and Analysis*

It was found that different methods, and even different number of neighbors searched in the case of the Euclidean method for heuristic value, gave different paths.

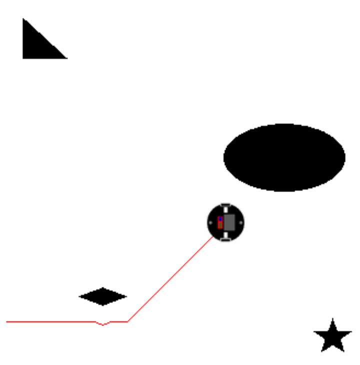


Figure 9. Diagonal with 8 neighbors to coordinate [-200.0, -90.0]

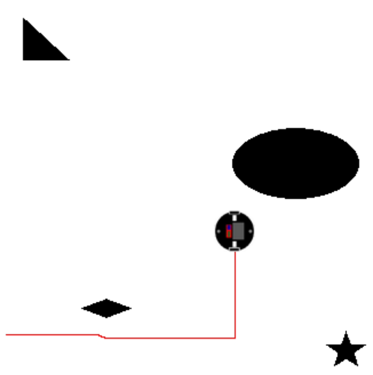


Figure 10. Euclidean with 4 neighbors to coordinate [-200.0, -90.0]

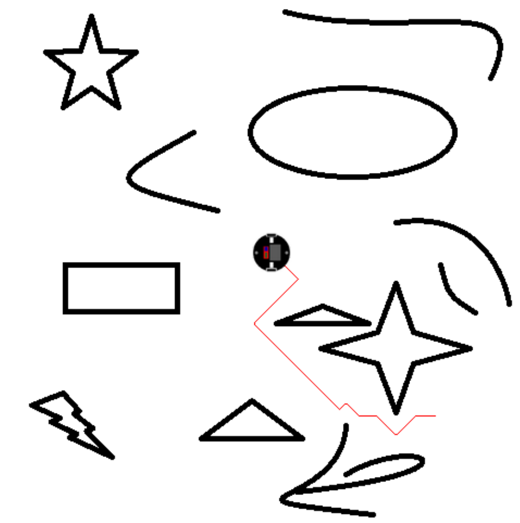


Figure 13. Diagonal with 8 neighbors to coordinate [150, -150]

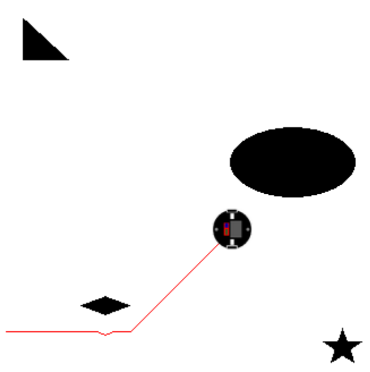


Figure 11. Euclidean with 8 neighbors to coordinate [-200.0, -90.0]

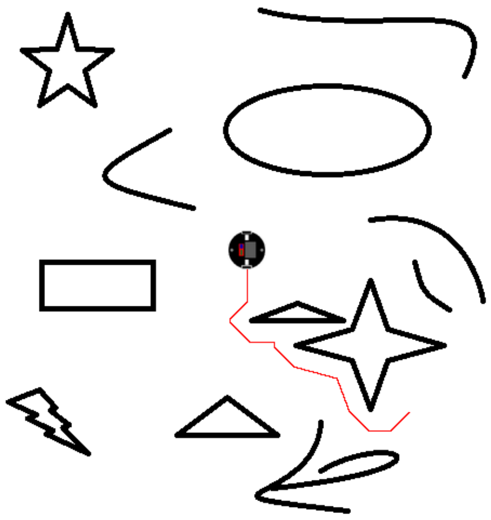


Figure 14. Euclidean with 4 neighbors with coordinates [150 -150]

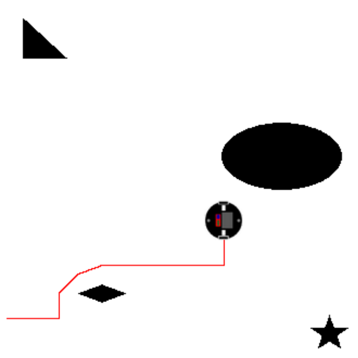


Figure 12. Manhattan with 4 neighbors to coordinate [-200.0, -90.0]

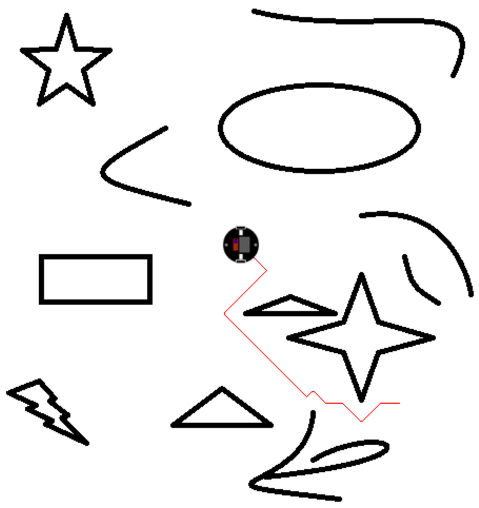


Figure 15. Euclidean, 8 neighbors, [150, -150]

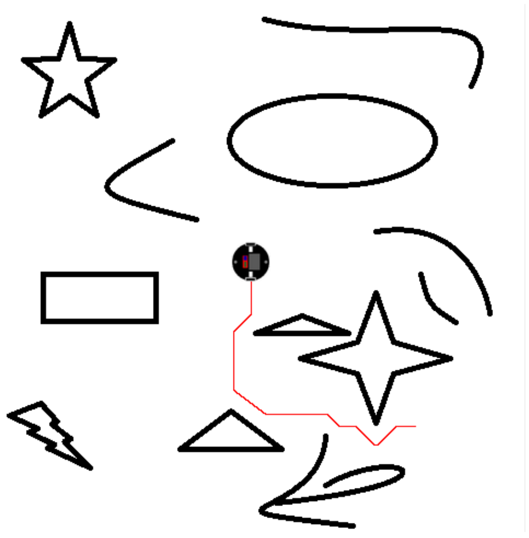


Figure 16. Manhattan, 4 neighbors, [150 -150]

Analyzing result figures 9-16 as well as other test runs a pattern of similarity started to arise between Diagonal methods which requires 8 neighbors and Euclidean method when selecting the 8 neighbors' option for it.

In regard to H constant, or as represented in the code the number multiplier of h resulting in hu. The change of value would potentially result in a change of path. The H constant is the average cost of nodes. So, having an extremely large variation between the real average cost and the average constant manually input will mess up the path. If H constant is closer to the real average value the graph will be more accurately representing the shortest path possible. Figure 16 shows constant H to equal 1.

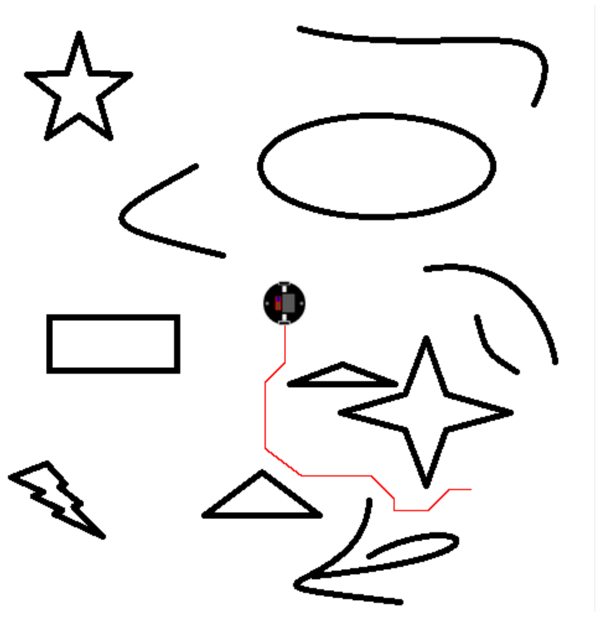


Figure 17. Constant H = 0

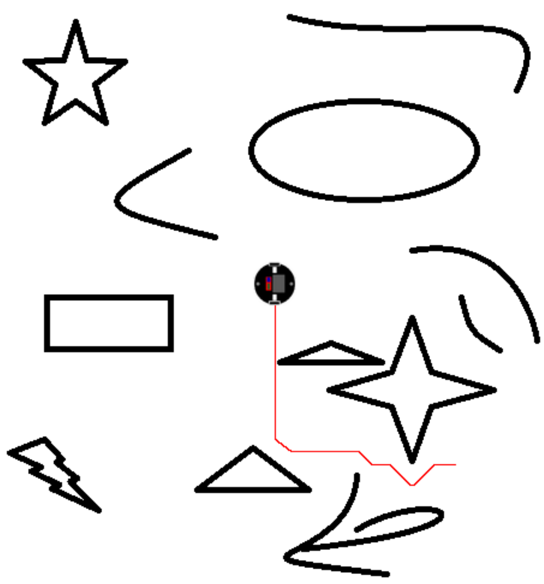


Figure 18. Constant = 1.84

Figure 18 shows the max allowed value for constant before beginning to behave as figure 19. This is an example of not so good path. The robot will hit the first triangle when moving down.

My cost map is made of ones and zeros so inputting something like 30 gives me figure 19.

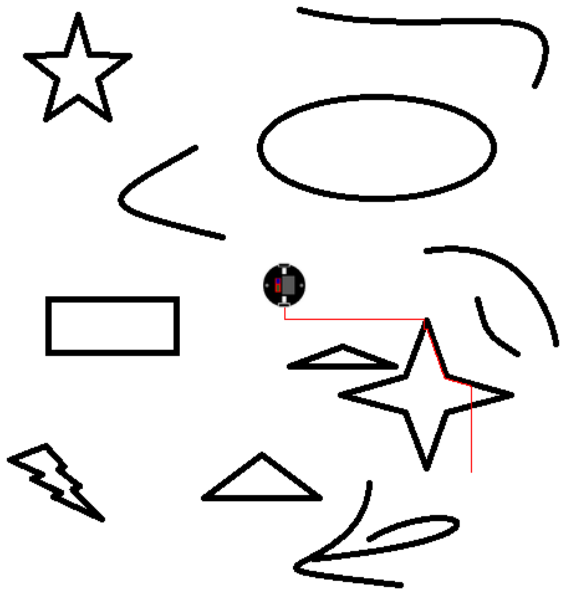


Figure 19. Average cost = 30 , something way off from actual average constant H.

This H constant representation is accurate in these instances. The average value of H should be around 1.

IV. Conclusion/ Analysis Continuesd

Different heuristic methods impact path differently. Different cost maps have different average cost, so they have different constant H values. Having a constant H value more accurate to the real average cost renders a higher quality path. The more nodes it takes to get to the goal the longer the gui takes to generate path. Higher accuracy path results in less nodes which mean shorter wait time for gui to load with path line image.

Acknowledgment

Thank you google for proving an abundant amount of information. 😊 Especially the Red blog games website, for breaking down the A* algorithm with several examples. Also, professor, for helping save the planet by riding buses.

References

[1] R. Siegwart, I. Nourbakhsh, D. Scaramuzza, Introduction to Autonomous Mobile Robots, 2nd ed., The MIT Press : Cambridge, 2011. pp 57 – 98.

[2] Y. Chang, Robotics Motion Planning , Lecture Note Set #12-13. CPP: ME5751, 2022.

[3] https://stackoverflow.com/questions/40205223/priority-queue-with-tuples-and-dicts

[4] https://www.geeksforgeeks.org/python-dictionary/

[5] https://www.w3schools.com/python/python_dictionaries.asp

[6] https://www.geeksforgeeks.org/priority-queue-set-1-introduction/

[7] https://www.mygreatlearning.com/blog/a-search-algorithm-in-artificial-intelligence/

[8] https://plainenglish.io/blog/a-algorithm-in-python

[9] https://stackabuse.com/courses/graphs-in-python-theory-and-implementation/lessons/a-star-search-algorithm/

[10] https://www.redblobgames.com/pathfinding/a-star/implementation.html#optimize-graph

[11] https://www.geeksforgeeks.org/a-search-algorithm/

[12] https://en.wikipedia.org/wiki/A*_search_algorithm

[13] https://www.vtupulse.com/artificial-intelligence/implementation-of-a-star-search-algorithm-in-python/

[14] https://leetcode.com/problems/shortest-path-in-binary-matrix/discuss/313347/A*-search-in-Python

[15] https://www.codementor.io/blog/basic-pathfinding-explained-with-python-5pil8767c1

[16] https://leetcode.com/problems/cut-off-trees-for-golf-event/discuss/1295156/A-Star-Search-Algorithm-in-Python-3-(Faster-than-90-solutions)

[17] https://www.google.com/search?q=a8+search+leetcode&rlz=1C1CHBF_enUS950US950&oq=A8+search+lee&aqs=chrome.1.69i57j33i160l3.8324j0j4&sourceid=chrome&ie=UTF-8

[18] https://leetcode.com/problems/shortest-path-in-binary-matrix/discuss/1602336/Simple-BFS-Python-solution-with-comments

[19] https://leetcode.com/problems/cut-off-trees-for-golf-event/discuss/1033258/Python-BFS-and-PriorityQueue-w-comments-and-prints-for-visualization

[20] https://rkhajuriwala.github.io/assets/documents/Robot-Navigation-and-Path-planning.pdf

[21] https://note.nkmk.me/en/python-multi-variables-values/#:~:text=You%20can%20assign%20the%20same,variables%20to%20the%20same%20value.&text=It%20is%20also%20possible%20to%20assign%20another%20value,after%20assigning%20the%20same%20value.

[22] https://leetcode.com/problems/top-k-frequent-words/discuss/297221/python-priority-queue-easy-to-understand-and-straightforward