Name: Sidhu Arakkal, Eoin O'Hare, Mohammed Awais Zubair
Date: 9/28/2022
Fall 2022

Assignment 1 (CS 440)
Submission 1

---

1. Any-Angle Path Planning

   (a) For our interface, we decided to use matplotlib and matplotlib.pyplot. We represented the starting location with a blue circle on the vertex, and the goal location with an orange circle on the vertex. After computing the path, the path is represented through a red dotted line drawn between each of the vertices on the path, as shown in the figure below. The blocked cells are represented with a gray cell. The visualization is capable of representing the 50

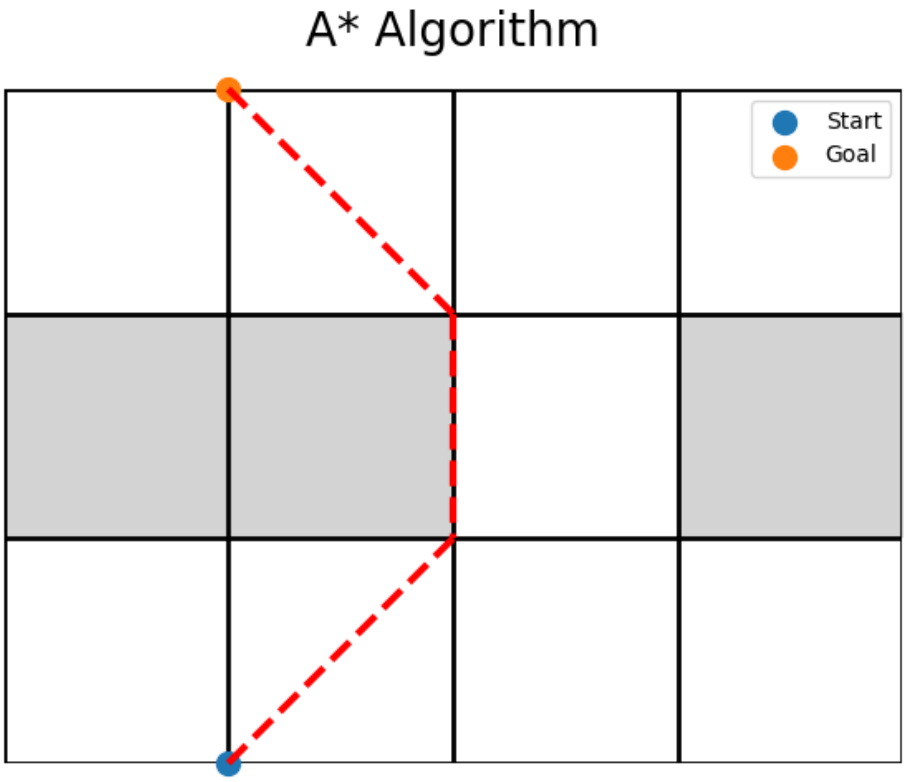

Figure 1: A* Trace Visualization

eight-neighbor grids for the experiments, and is already capable of showing the blocked cells and optimal path from the start and end goal. Currently, we are using the terminal to represent the h, f, and g values of the vertices. While computing the path, the values of the vertex after it is popped from the fringe will be printed and then all the neighboring vertices' values will be printed. We will add the capability for buttons by the full submission of the project to better represent the values of the experiments. An example of the terminal's output is shown below:

```
Current Vertex: [1, 4], H: 3.414213562373095 , F:18.31009655751752 , G:14.895882995144426
Current Vertex: [1, 4], H: 3.414213562373095 , F:19.092244416535554 , G:15.678030854162458
Current Vertex: [4, 2], H: 2.414213562373095 , F:19.61886409645835 , G:17.204650534085253
Neighboring Vertex:[5, 2], H: 3.414213562373095 , F:30.467721898254453 , G:27.053508335881357
Neighboring Vertex:[4, 3], H: 2.82842712474619 , F:29.467058790888046 , G:26.638631666141855
Neighboring Vertex:[4, 1], H: 2.0 , F:27.748654279402786 , G:25.748654279402786
Neighboring Vertex:[3, 1], H: 1.0 , F:25.820423639949162 , G:24.820423639949162
Neighboring Vertex:[5, 1], H: 3.0 , F:29.69148351459039 , G:26.69148351459039
Current Vertex: [3, 4], H: 3.414213562373095 , F:20.07879657771427 , G:16.664583015341176
Neighboring Vertex:[4, 4], H: 3.82842712474619 , F:31.123155952822017 , G:27.294728828075826
Neighboring Vertex:[4, 3], H: 2.82842712474619 , F:29.39250507669903 , G:26.56407795195284
Current Vertex: [4, 3], H: 2.82842712474619 , F:20.650296849081705 , G:17.821869724335514
Neighboring Vertex:[5, 3], H: 3.82842712474619 , F:32.466950675473676 , G:28.638523550727484
Neighboring Vertex:[5, 4], H: 4.242640687119286 , F:33.46626466244618 , G:29.223623975326895
Neighboring Vertex:[4, 4], H: 3.82842712474619 , F:32.28044266181635 , G:28.452015537070164
Current Vertex: [3, 4], H: 3.414213562373095 , F:21.23608328670861 , G:17.821869724335514
Neighboring Vertex:[4, 4], H: 3.82842712474619 , F:32.28044266181635 , G:28.452015537070164
Current Vertex: [4, 3], H: 2.82842712474619 , F:22.16190319341446 , G:19.33347606866827
Neighboring Vertex:[5, 3], H: 3.82842712474619 , F:33.97855701980643 , G:30.15012989506024
Neighboring Vertex:[5, 4], H: 4.242640687119286 , F:34.97787100677893 , G:30.73523031965965
Neighboring Vertex:[4, 4], H: 3.82842712474619 , F:33.79204900614911 , G:29.96362188140292
Path from Goal to Start
[2, 1]
[3, 2]
[3, 3]
[2, 4]
```

Figure 2: Visualization of Values on Terminal

(b) Search Algorithms

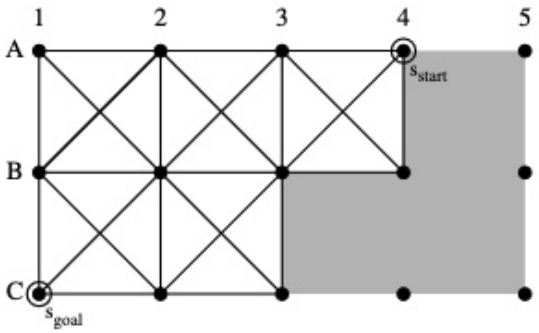

Figure 3: Sample Search Problem

i. *Shortest Grid Path*
The shortest grid path based on the sample problem in Figure 1 would be the following path: [A4, B3, C2, C1]. The weights of the path are: $\sqrt{2}$, $\sqrt{2}$, and 1. Thus the total distance would be $1 + 2\sqrt{2}$. Similarly, path [A4, B3, B2, C1] and [A4, A3, B2, C1] would have the same distance.

ii. *Shortest Any-Angle Path*
The shortest any-angle path based on the sample problem in Figure 1 would be the direct path from $A4 \rightarrow C1$. The distance of this path is calculated via the distance formula, which results in a distance of $\sqrt{13}$.

iii. *A* Trace*

Using the formula provided, Figure 2 shows the trace of the A* algorithm. The labels of the vertices are f-values (written as the sum of g- and h- values) and parents. We must assume that all numbers are precisely calculated even though the diagram has everything rounded to two decimal places. The arrows point to parents and red circles indicate expanded vertices. The A* algorithm follows the parents from the goal vertex, C1, to the start vertex, A4, via the dashed red path [A4, B3, C2, C1], which is one of the shortest grid paths found earlier.

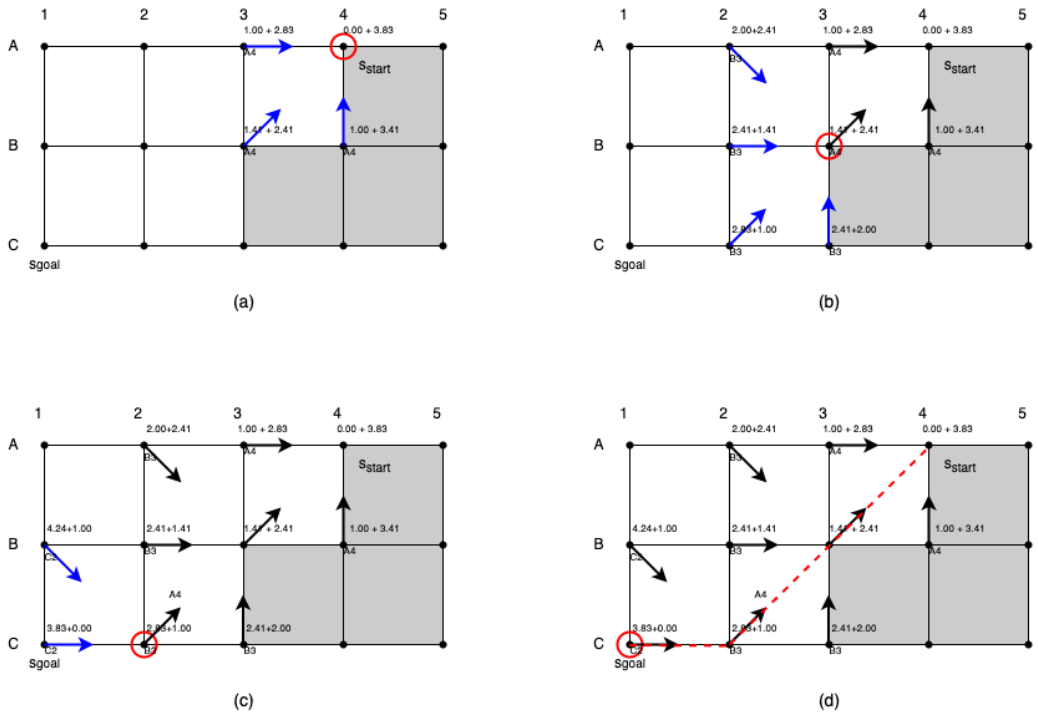

Figure 4: A* Trace

iv. *Theta* Trace*

Using the formula provided, Figure 3 shows the trace of the Theta* algorithm. Firstly, the algorithm expands vertex A4 as seen in Figure 3(a) and sets the parent of the unexpanded successors of vertex A4 to be A4. Then Theta* exapnds to vertex B3 with parent A4, as seen in Figure 3(b), and the only straight line that is blocked is between C3, thus its parent is updated to B3. The remaining vertexes (A2, B2, C2) have their parents remain A4 since the path is unblocked. Thirdly, the algorithm expands to vertex B2 with parent A4 and sees that all the remaining points (A1, B1, and C1) still have a line of sight to the starting point A4, thus their parent remains A4. However, since C1 is the goal node, the algorithm expands to C1 and identifies the direct dashed path from C1 to A4, which is a shortest any-angle path gotten above.

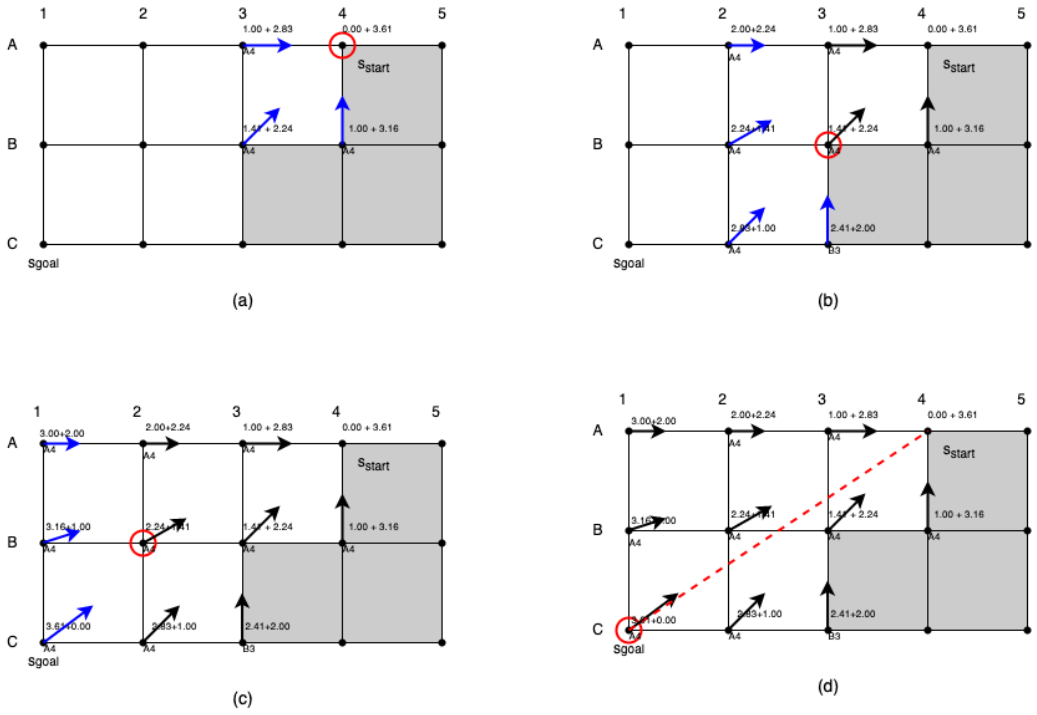

Figure 5: Theta* Trace

(c) As the difference between the A* algorithm and the Theta* algorithm lies in the updating of the vertices, we decided to create a common function that popped vertices out of the fringe (our min-Heap priority queue) and added all of the adjacent vertices into the fringe that were not added to it already. We implemented a closed set to ensure that vertices would not be added to the fringe multiple times and only be expanded a maximum of one time. In order to add all adjacent vertices, we implemented an adjacency list. This adjacency list is a python dictionary where the indexes are the coordinates of a vertex and the accompanying terms are the neighboring vertices. In order to achieve this, we first added the popped vertex's coordinates as a string to be a term in the dictionary. Then, we ran a function that verified that the edges of the neighboring vertices were acceptable through checking various conditional statements. If the edge between the vertices was passing diagonally through a blocked cell, the edge was invalid and the neighboring vertex was not added to the adjacency list. Also, if the edge was passing vertically or horizontally between two blocked cells, that was also a violation and the neighbor would not be added to the adjacency list of the current vertex. If the adjacent vertex had a valid edge, it was then determined whether the vertex was in the closed set. If not, then the fringe would be checked if it contained the neighboring vertex. If the fringe does not contain the neighboring vertex, then the g value of the vertex is set to infinity so all other g values would be less then it. Also, the parent would be set equal to null. Then, if the command-line argument was set to "aStar", the update vertex function for the A* algorithm would be used.

In the update vertices function for the A* algorithm, the distance between the popped vertex (s) and the adjacent vertex (s') would be calculated. This distance would then be used for comparing the g value of the adjacent vertex with the g value of s and the c(s,s'). If the g value and c(s,s') (or distance) was less than the g value of the adjacent vertex, then the g value of the adjacent vertex would be re-assigned to be g(s) + c(s,s'). Then, the parent of the adjacent vertex would be set as s. If the fringe already contained the adjacent vertex, then it would be removed from the fringe. Afterwards, the adjacent vertex's h and f values will be set and the adjacent vertex would be inserted into the fringe. The rest of the current vertex's valid neighbors will be checked against the closed set, fringe, have its values updated, and then be inserted into the fringe if meeting the conditional statement. Finally, the current vertex would be fully expanded and the next vertex would be popped from the fringe and this process would continue until either the fringe is empty or the vertex being popped is the goal vertex. The path would then be printed onto the grid and visualized, completing the A* algorithm.

(d) As mentioned in part c, the algorithm leading up to the updating of the vertices is the same for both the A* and Theta* algorithms. Therefore, our implementation of Theta* uses the same method of popping from the fringe, generating the adjacency list, examining the closed set, and then checking conditionals of whether to update the values of the vertex before being added to the fringe. If the command-line argument was set to "thetaStar," then the update vertex function for the Theta* algorithm would be used.

In order to update the vertices according to the Theta* algorithm, a line of sight function had to be implemented. The line of sight function returned a conditional statement of whether the neighboring node was in the line of sight of the current node's parent. The function used a variety of conditional statements to ensure that if the path between the vertices crossed through a blocked cell at any-angle, then the neighboring vertex would be considered to be out of the line of sight of the current node's parent and return a false value. If the line of sight function returned a false value, then the vertex being added to the fringe would be updated according to the A* algorithm. If the line of sight function returned a true value, then the g

value of the parent and the distance between the parent and the neighboring vertex would be calculated. If the calculated value is less than the g value of the neighboring vertex in the line of sight, then g value of the neighboring vertex would be set to the calculated value and the parent of the vertex would be set to the parent of the current node popped from the fringe. Then, if the fringe contains the neighboring vertex, then the vertex would be removed from the fringe. Afterwards, the vertex would have its h and f values set before adding the vertex to the fringe and then returning the fringe. The current vertex would be fully expanded and have each neighboring vertex checked by the line of sight function with its parents. Finally, the next vertex would be popped from the fringe and this process would continue until either the fringe is empty or the vertex being popped is the goal vertex. The path would then be printed onto the grid and visualized, completing the Theta* algorithm.

(e) We can prove that A* with the h-values from Equation 1 is guaranteed to find the shortest path by utilizing the properties of the given h-value(s). From lecture, we know our heuristic must be admissible. It can also be consistent, because we know a heuristic that is consistent. must also be admissible. A consistent heuristic has consistent h-values if and only if they satisfy the triangular inequality:

$$\textit{iff } h(s_{goal}) = 0 \textit{ and } h(s) \leq c(s,s') \textit{ for all vertices } s,\ s' \in S$$

Additionally, Equation 1 is taking the max distance ($max(|s^k - s^k_{goal}|)$) and the minimum distance of the diagonals. This must guarantee the shortest path, since it cannot over predict due to the inequality shown above. Since each edge costs one unit and going past one block would two edges (i.e. two units) we must minimize the edge use. Using diagonals, which cost $\sqrt{2}$ units would lower the overall cost assuming there is no blocking of diagonals. This best case scenario created through analyzing Equation 1 incorporates some number of edges and some number of diagonals to minimize total cost and result in the shortest path. Equation 1 is shown below:

$$h(s) = \sqrt{2} * min(s^x - s^x_{goal}, s^y - s^y_{goal}) + max(s^x - s^x_{goal}, s^y - s^y_{goal}) - min(s^x - s^x_{goal}, s^y - s^y_{goal}))$$

The maximum distance between the current point and the goal point would require taking a path where no diagonals are used, as the edge cost of 2 units whenever there is a vertical and horizontal difference would be higher than $\sqrt{2}$ (which is around 1.4). Therefore, the heuristic equation in this problem cannot neglect diagonals, or it gives the possibility of over-estimating and would thus not be admissible. The only situation in which to never over-estimate would to use every diagonal possible and never use the 2 edge cost path. Therefore, the minimum between the horizontal and vertical difference of the current point and the goal point would provide the number of diagonals that it would take to either completely remove the vertical difference or horizontal difference. After the vertical or horizontal difference is removed, the only edge costs remaining would be the 1 edge cost path to the goal point. While this equation does not account for blocked cells through using every diagonal possible, it fulfils the admissible property of the heuristic equation by leaving no possibility that the h value is greater than the optimal path.

(f) In our project, we decided to implement our own binary heap to use in both the A* and Theta* implementations. Our min-heap priority queue is located in the priorityQueue.py file in the classes folder. In order to create the priority queue for the algorithms, we chose to use a minimum Heap. The key of the min-Heap would be the f-value of the vertices being expanded upon. Our min-Heap has functions to find the parent of a given index, the right child of a

given index, and the left child of a given index. Using these functions, we were able to create a min-Heapify function where the min-heap is ensured to be following the principle where the parent nodes have smaller f-values than its two children. Therefore, if the parent is smaller than any children, the min-Heapify function switches the parent's position with the child with the lower f value and then checks the rest of the min-Heap recursively. We provided functions to switch the position of two vertices in the heap, pop the minimum value of the heap, and to insert a vertex into the heap. After inserting into a heap, our code ensures that the heap is still following the behavior of a min-heap, where every parent has a smaller f-value. Finally, we provided functions to check whether the heap is empty, see if a heap contains a certain vertex, and to initialize the class.