Computational Robotics Homework 1

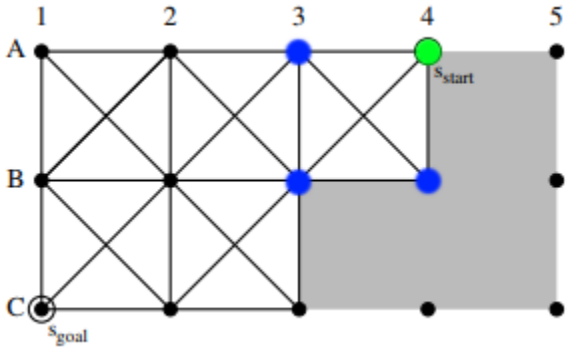
Jack Chiu
Bhargav Desai

October 2, 2018

1 Question 1

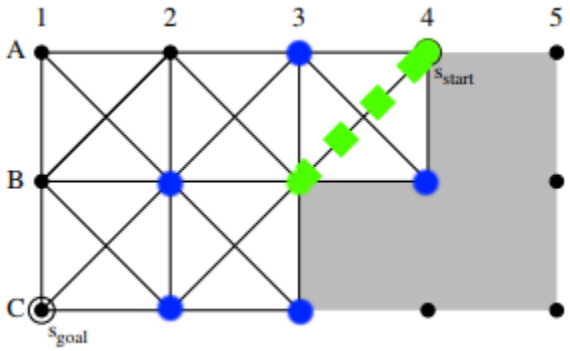
1.1 A*

Iteration 1: checking the three available nodes next to the start node.



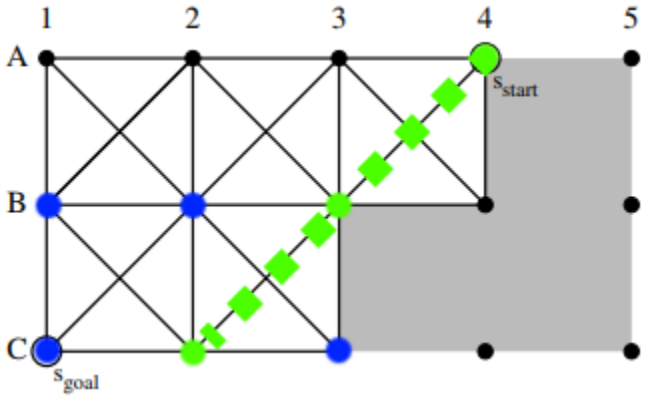
Node	h	g
start	$2\sqrt{2}+1 \approx 3.83$	0
A3	$2\sqrt{2} \approx 2.83$	1
B3	$\sqrt{2}+1 \approx 2.41$	$\sqrt{2}$
B4	$\sqrt{2}+2 \approx 3.41$	1

Since the nodes A3 and B3 have the same h+g value, the tie breaker is the higher g value. As such, B3 is chosen to be the next node.
Iteration 2: checking the five available nodes next to node B3.



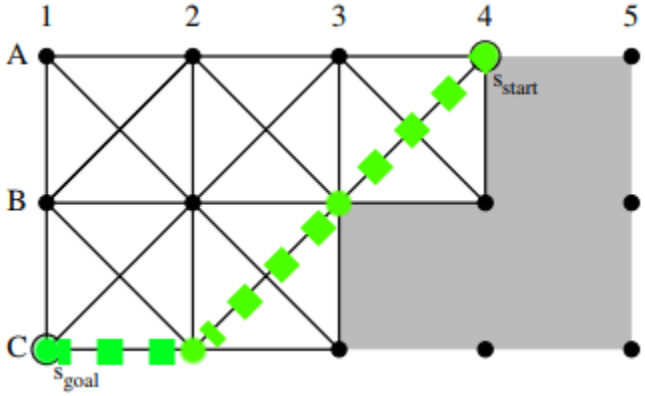
Node	h	g
A3	$2\sqrt{2} \approx 2.83$	$\sqrt{2}+1$
B2	$\sqrt{2} \approx 1.41$	$\sqrt{2}+1$
B4	$\sqrt{2}+2 \approx 3.41$	$\sqrt{2}+1$
C2	1	$2\sqrt{2}$
C3	2	$\sqrt{2}+1$

Evidently, the lowest h+g value comes from node B2 and C2. However, since C2 has a higher g value, it is chosen.
Iteration 3: checking the four available nodes next to C2



Node	h	g
B1	1	$3\sqrt{2}$
B2	$\sqrt{2}$	$2\sqrt{2}+1$
C1	0	$2\sqrt{2}+1$
C3	2	$2\sqrt{2}+1$

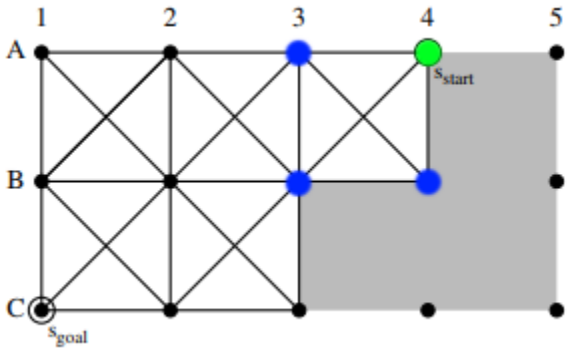
From the table, the lowest h+g value is from C1, which is the goal. As such, the A* algorithm completes.



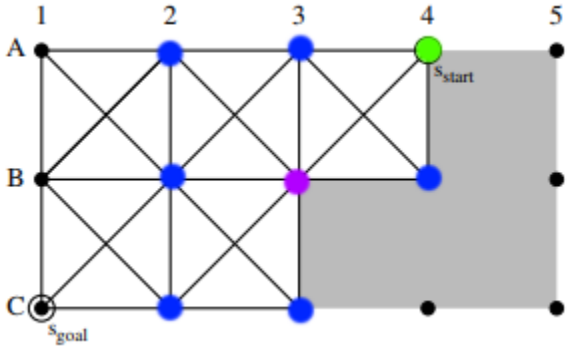
1.2 FDA*

Iteration 1: checking the three available nodes next to the start node.

Node	h	g
start	$\sqrt{13} \approx 3.61$	0
A3	$2\sqrt{2} \approx 2.83$	1
B3	$\sqrt{5} \approx 2.24$	$\sqrt{2}$
B4	$\sqrt{10} \approx 3.16$	1



The lowest h+g value is B3 with 3.65. As such, we look at the successors of B3 and see if there exists a line of sight from the start to the lowest h+g node.

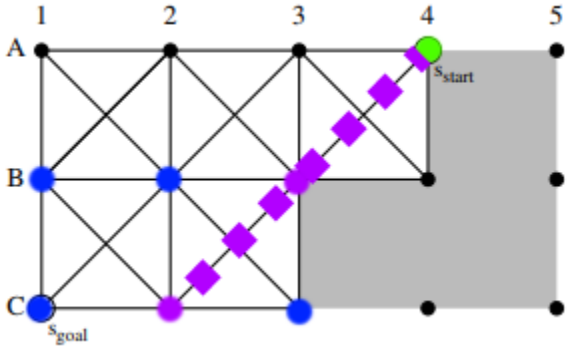


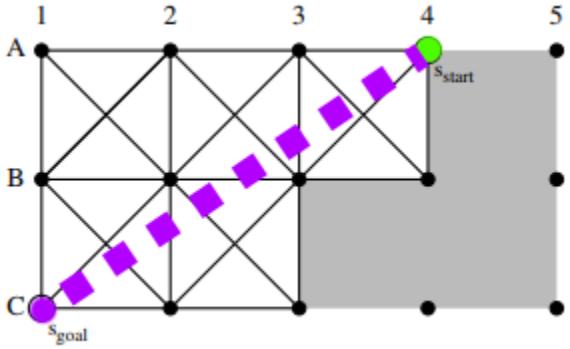
Node	h	g
A2	$\sqrt{5} \approx 2.24$	2
A3	$2\sqrt{2} \approx 2.83$	1
B2	$\sqrt{2} \approx 1.41$	$\sqrt{5} \approx 2.24$
B4	$\sqrt{10} \approx 3.16$	1
C2	1	$2\sqrt{2}$
C3	2	$\sqrt{5} \approx 2.24$

(Since C3 is not in the line of sight of the start node, we remove it from the list)
From the table, the lowest h+g value is from C2 and there is a line of sight from the start node. So we repeat the algorithm and check the nodes around C2.

Node	h	g
B1	1	$\sqrt{10} \approx 3.16$
B2	$\sqrt{2} \approx 1.41$	$\sqrt{5} \approx 2.24$
C1	0	$\sqrt{13} \approx 3.61$
C3	2	$\sqrt{5} \approx 2.24$

The lowest h+g is C1 at $h + g = 3.61$. Since there is a line of sight from the start node to C1, we choose C1. Since C1 is the goal, we are finished.





2 Question 2

In our grid implementation, the package matplotlib was used in order to define the polygonal obstacles. Rather than drawing blocked out grids to determine the possible paths of the robot, the obstacles are generated with a buffer such that the robot will not collide with the actual obstacles. The only positions the robot can go to are specified by the interval between points on the grid. By default, the grid size is set to intervals of 0.25. As such, the robot can only move to points with x and y values that are multiples of 0.25. The polygonal representation is chosen rather than a completely shaded grid square representation since it provides for closer distance to the obstacle. If a shaded grid square representation for the obstacles was chosen, there would be less points available for the robot to choose from to traverse. Thus, the polygonal representation was used. Furthermore, the buffer size of 0.2 was chosen since the robot has an approximate diameter of 0.4. Having a smaller buffer size allows for closer paths to obstacles, and thus, shorter overall paths.

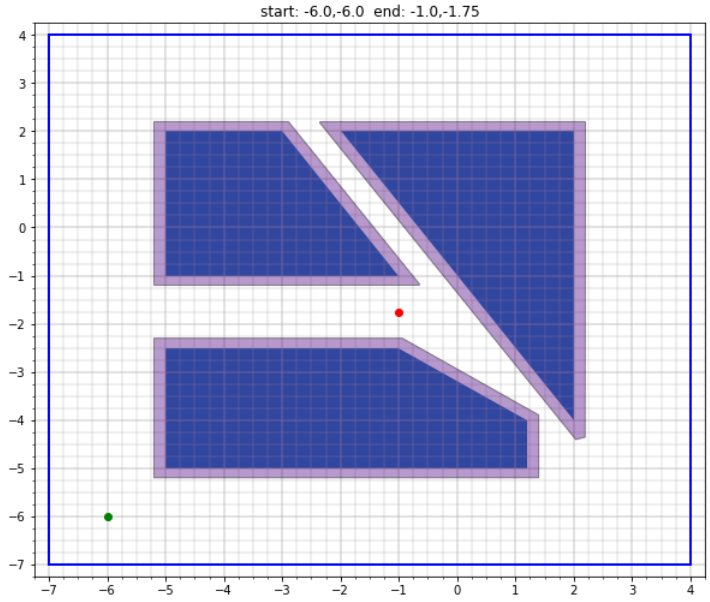


Figure 1: The above shows the original polygon in dark blue and the polygons with buffers in light purple for world 1 with a hypothetical start(green) and end(red) point

3 Question 3

In the following demonstrations of the A* algorithm, the original polygonal obstacles and its buffered versions are displayed. The path of the robot is a green line stemming from the starting green point to the goal red point. To further highlight the robot path, a yellow circle with a diameter of 0.4 is drawn to show the outside of the robot and prove that it does not collide with the actual obstacles at any point.

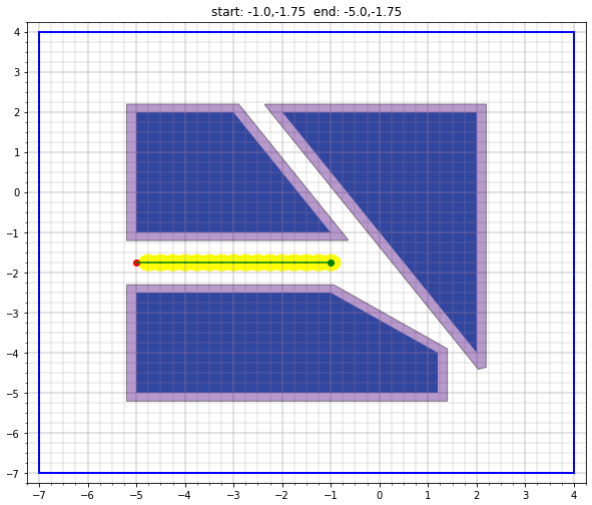


Figure 2:

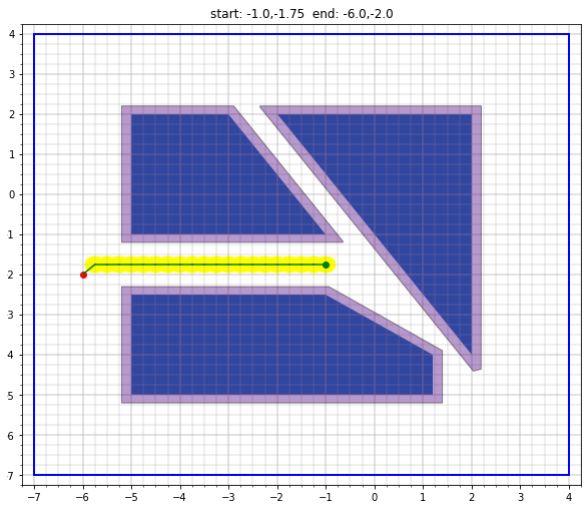


Figure 3:

Based on the above figures, it is quite apparent that the shortest path is found with the A* algorithm. For instance, The Figure 2 and Figure 3 demonstrate that the shortest path is found by following grid point by grid point traversal. Figure 2 shows a direct straight line from the start to the goal. Likewise, the same path is found for Figure 3 except the final step is a diagonal since the goal is one interval lower than that of Figure 2. We look at the other figures as well. With each traversal, we see that each step brings the robot closer to the goal as long as the current point does not overlap the buffered obstacles. For instance, take

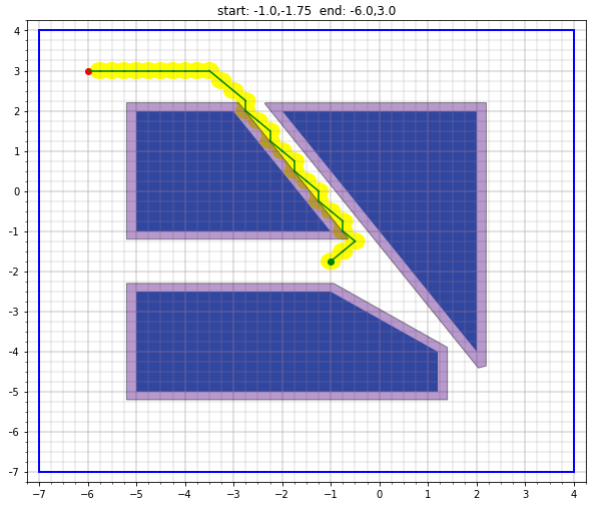


Figure 4:

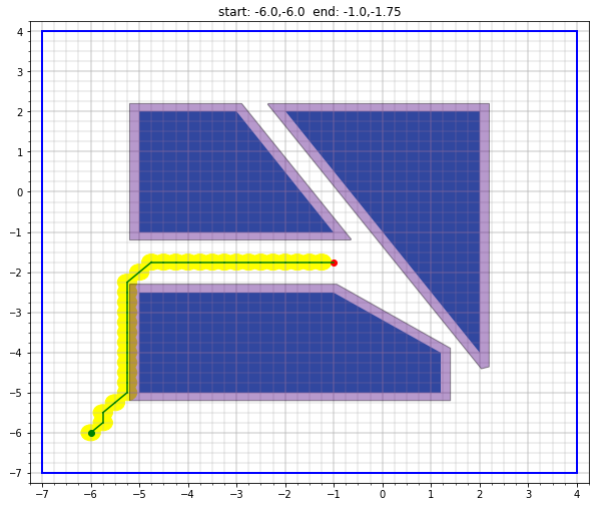


Figure 5:

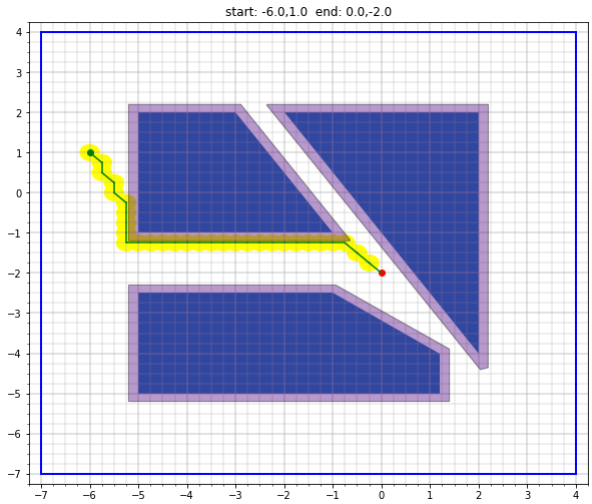


Figure 6:

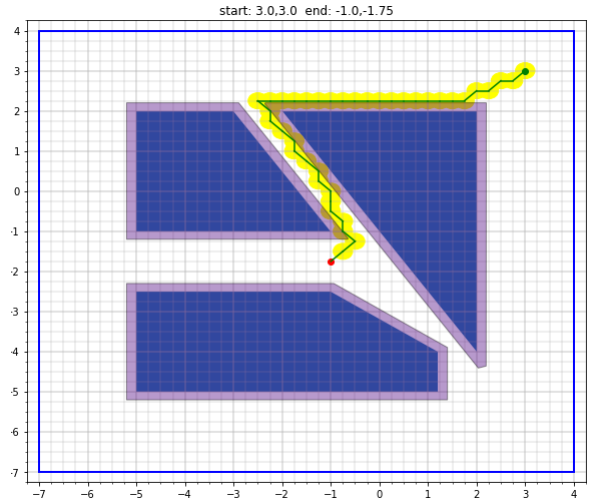


Figure 7:

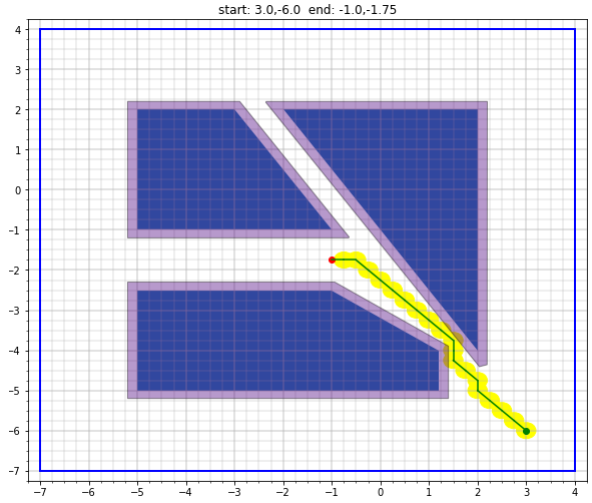


Figure 8:

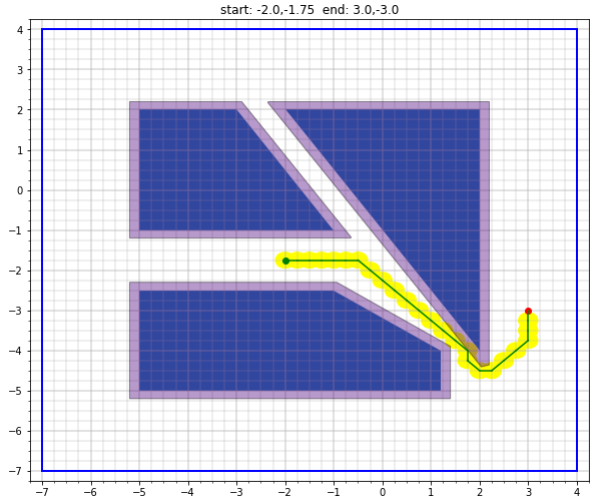


Figure 9:

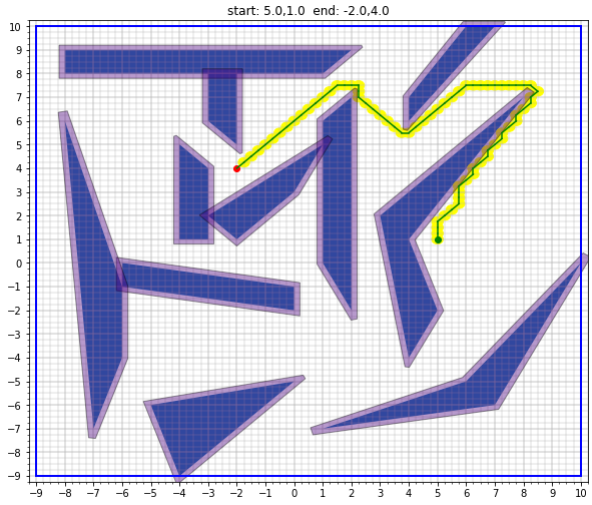


Figure 10:

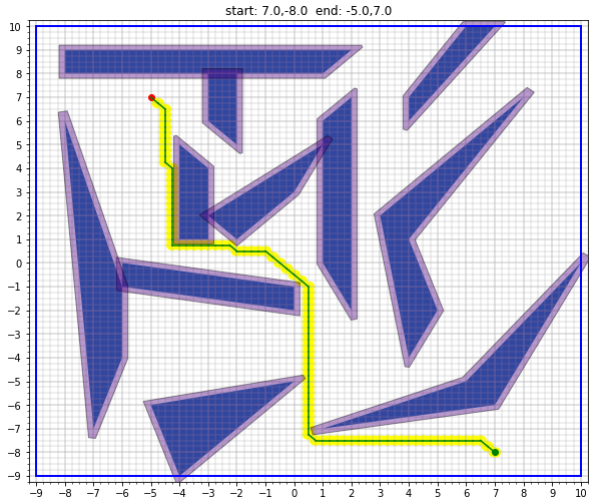


Figure 11:

Figure 10. The determined path jigsaws back and forth along the diagonal of the first obstacle the robot meets. This is due to the attempt to keep the robot as close to the border of the obstacle without entering the buffered zone around the obstacles.

4 Question 4

The implementation of FDA* is done such that the successors of the selected next position is chosen until the lowest (cost + heuristic) value is no longer visible by the first node in this continuous check. The reason for running FDA* is to allow for even shorter paths from the start to the goal. In comparison, A* is locked to traversing between adjacent nodes only. However, FDA* allows jumping from one node to another one much further away as long as there is a line of sight. Essentially, FDA* causes the robot to moves along the straight line distance between the two nodes rather than moving along the x and y axis.

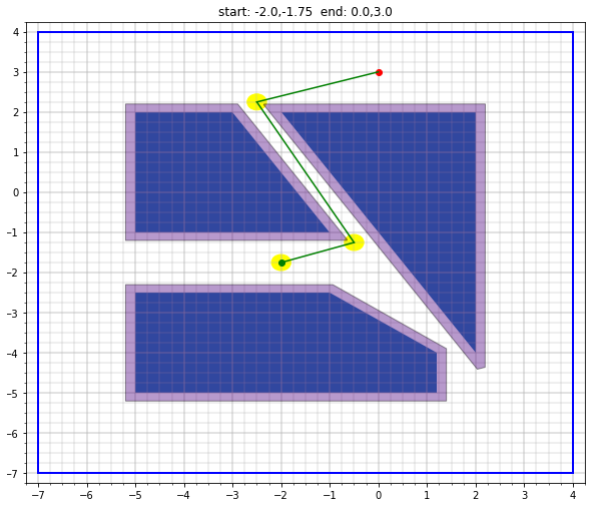


Figure 12:

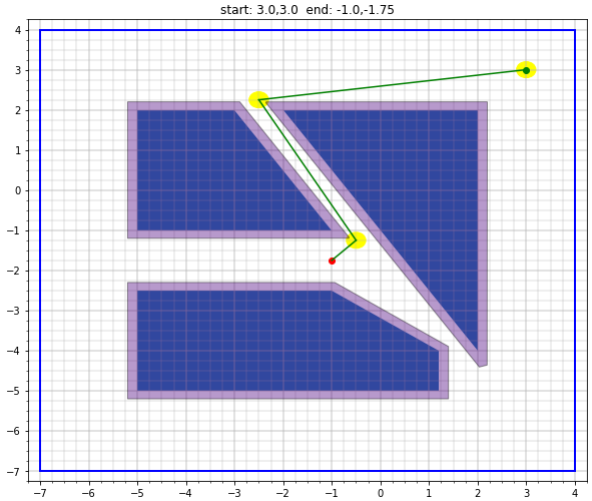


Figure 13:

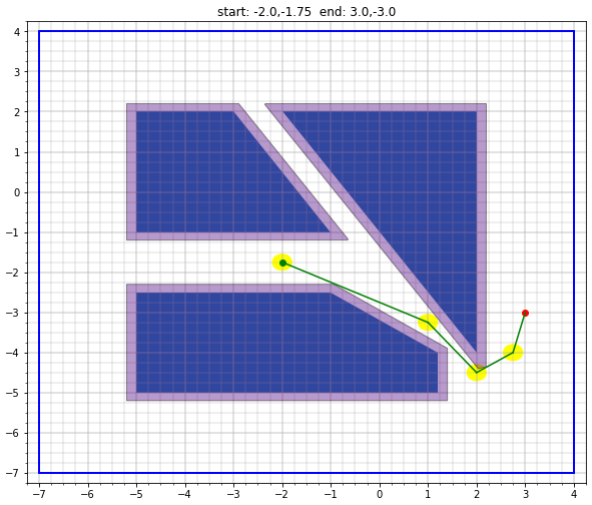


Figure 14:

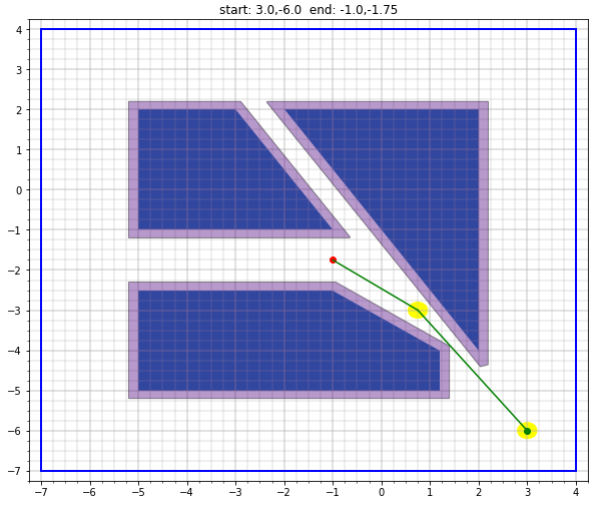


Figure 15:

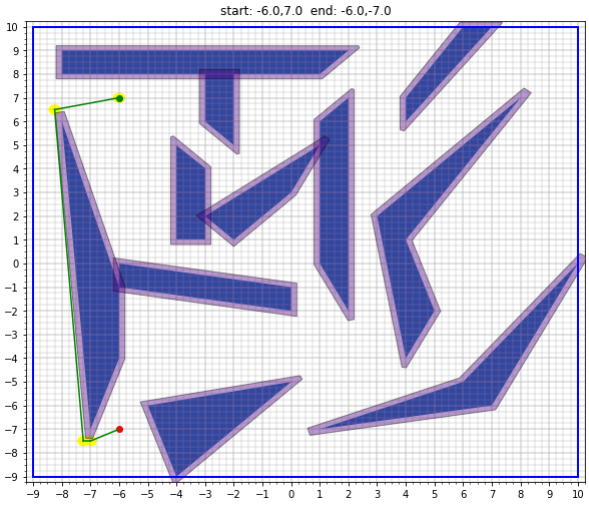


Figure 16:

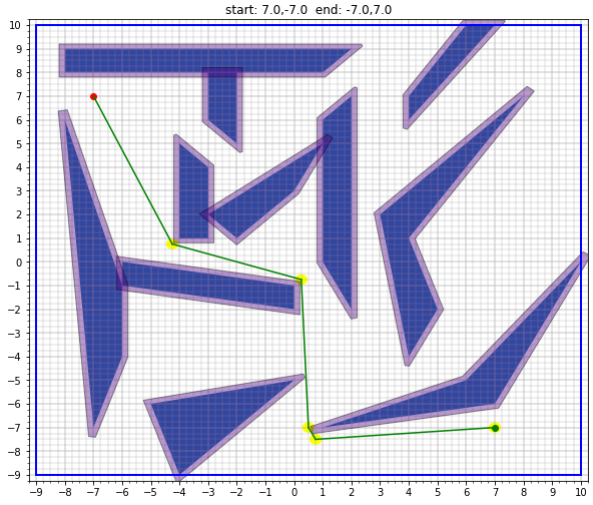


Figure 17:

5 Question 5

In order to optimize the A* algorithm and the FDA* algorithm, we can optimize it in terms of time and space. In terms of time, the way to optimize the algorithm is to have an open list to add nodes that are searched from the neighbors of successor nodes. This would ensure that these nodes are checked first. This is done through the fringe list in our algorithm. In the algorithm, the fringe is appended with the successors of the current node if it is not in the closed list and meets the condition of being a possible potential node to traverse. The fringe sorts it by the lowest key $h + g$ and pops the lowest node to check. The other method of optimizing in terms of speed is to change the heuristic. Testing the heuristic of $h_1 = \sqrt{2} \cdot min\{|s^x - s^x_{goal}|, |s^y - s^y_{goal}|\} + max\{|s^x - s^x_{goal}|, |s^y - s^y_{goal}|\} - min\{|s^x - s^x_{goal}|, |s^y - s^y_{goal}|\}$. On average, the heuristic tested on 10 start/goal pairs yielded a run time of approximately 33 seconds. Testing other heuristics, we used the straight line distance from the node to the goal and timed the run time. The average run time for that heuristic yielded value of approximately 39 seconds with not much change in the found shortest path. As such, we can conclude that the straight line distance heuristic is a slightly underestimating heuristic.

In the code, we added a tie break to sort the fringe list by the g value prior to sorting by the f value (h+g). By doing so, we expect the time to decrease for running the algorithm because A* will prioritize nodes that

are closer to the goal node. Experimentally testing this, we have that the A* algorithm run with the tie break is 1 second faster than without on world 2 and faster by 8 seconds on world 3.

In order to optimize it in terms of space, the algorithm generates vertices in terms of x and y values as they are needed and checks equality of nodes by checking if the x and y values are equal. The original method is to pre generate the grid of nodes prior to conducting the A* and FDA* algorithms and checking equality based on memory location. The former method saves a lot more space than the latter.

6 Question 6

From running the A* algorithm and the FDA* algorithm, the below table of experimental times is found. As expected, the A* algorithm is faster, although not by too much. This is due to the fact that A* goes step by step and checks immediate cost and heuristic while FDA* needs to check all successors of successors and needs more computation time from checking line of sight. Ultimately, the more nodes that need to be checked in order to reach the goal, the greater the gap between FDA* and A* runtimes because FDA* still needs to check line of sight for each potential node before traversal.

World	A* Run Time	FDA* Run Time
world 1	1.51600027084	1.82300019264
world 2	10.4030001163	11.7110002041
world 3	44.2639994621	57.2510001659
world 4	34.7589998245	42.2749998569

7 Question 7

For our heuristic in this example, we are using the Euclidean straight line distance between two points. This is equivalent to $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. A heuristic is consistent if the cost from the current vertex to a neighboring vertex, plus the estimated cost from the neighboring vertex to the goal is less than or equal to the estimated cost from the current node to the goal. In other words, given current vertex s, and neighboring vertex s', a consistent heuristic would be one where cost(s,s') + heur(s', goal) $\geq$ h(s,goal). The Euclidean straight line distance heuristic is consistent due to the triangle inequality. Since the cost(s,s') is also calculated with the Euclidean straight line distance, a triangle is created, which is shown in Figure 18. Since h(s) is the hypotenuse of said triangle, it must be less than or equal to the other edges of the triangle. Thus, our heuristic is consistent.

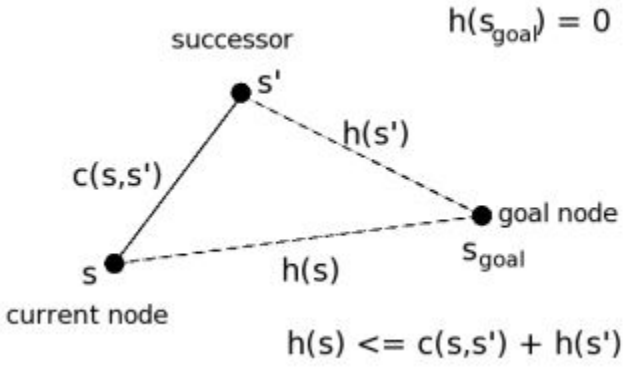


Figure 18: Triangle Inequality for Consistent Heuristic

In Figure 19 below, we have an example search problem with the start node at (4,0) and the goal node at (1,2).

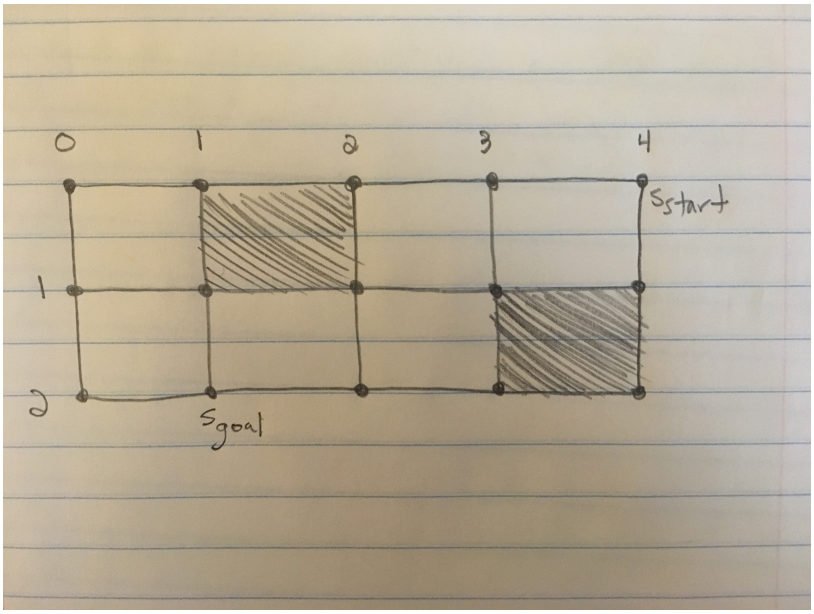


Figure 19: Example Search Problem

In the table below, we have the h values (straight line distance) for every coordinate to the goal node.

Coordinates	H-Val
(0,0)	2.24
(1,0)	2
(2,0)	2.24
(3,0)	2.83
(4,0)	3.61
(0,1)	1.41
(1,1)	1
(2,1)	1.41
(3,1)	2.24
(4,1)	3.16
(0,2)	1
(1,2)	0
(2,2)	1
(3,2)	2
(4,2)	3

Example Trace:

The start vertex is (4,0) and the goal is (1,2).

Set (4,0) g-val to 0. Set its parent to itself. Add it to the openList.

closed = []
openList = [[(4,0), f-val = 3.61]]

Pop (4,0), expand it and check if it is the goal. If not, add it to closed. Check if there is a line of sight between (4,0) and its neighbors. If there is, set each neighbor's parent as (4,0), and update the g-val as the straight line distance between (4,0) and itself. In this case, all neighbors have (4,0) as their parent. Add the neighbors to openList, and sort it based off f-val (g-val+h-val).

closed = [[(4,0),f-val=3.61]]
openList = [[(3,1),f-val=3.65], [(3,0),f-val=3.83], [(4,1), f-val=4.16]]

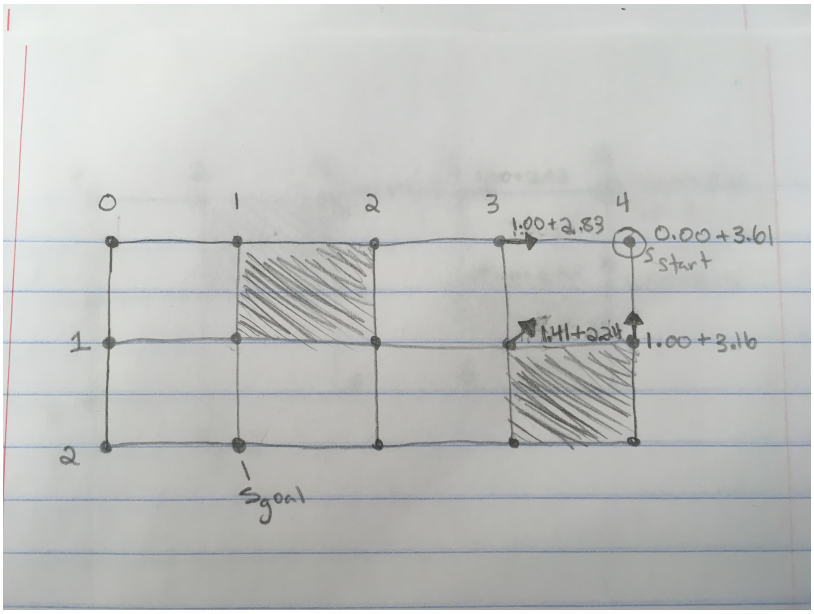


Figure 20: Example Search Problem: After Expanding (4,0)

Pop (3,1), expand it, and check if it is the goal. If not, add it to closed. For each neighbor, check if there is a line of sight between the parent of (3,1), which is (4,0), and itself. Update g-values and parents accordingly as described above, and add each neighbor to the openList.

closed = [[(4,0),f-val=3.61],[(3,1),f-val=3.65)]]
openList = [[(2,1),f-val=3.65],[(3,0),f-val=3.83],[(2,2),f-val=3.83],[(4,1),f-val=4.16],[(2,0),f-val=4.23],[(3,2),f-val=4.41],[(4,2),f-val=5.00]]

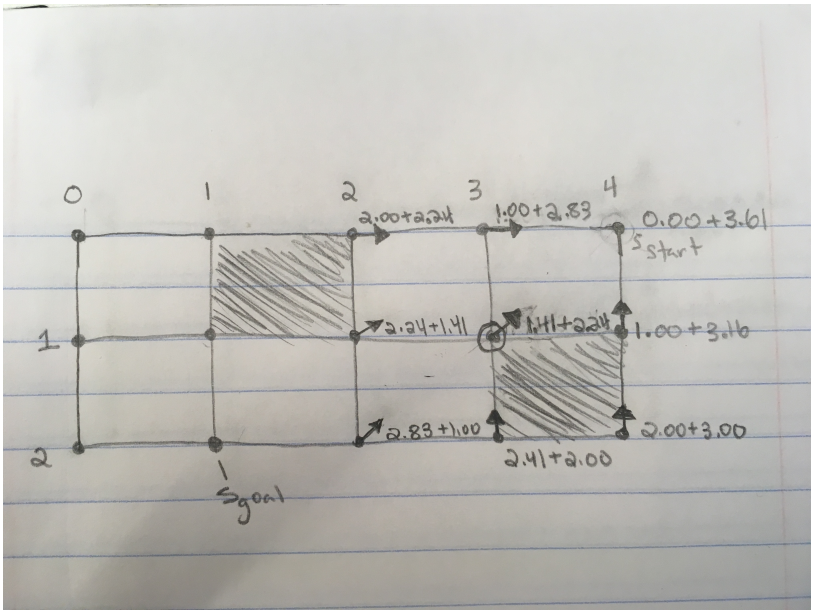


Figure 21: Example Search Problem: After Expanding (3,1)

Pop (2,1), expand it and check if it is the goal. If not, add it to closed. For each neighbor of (2,1), check if there is a line of sight between the parent of (2,1), which is (4,0), and itself. Update g-values and parents accordingly as described above, and add each neighbor to the openList.

closed = [[(4,0),f-val=3.61],[(3,1),f-val=3.65)],[(2,1),f-val=3.65]]
openList = [[(1,2),f-val=3.61],[(3,0),f-val=3.83],[(2,2),f-val=3.83],[(4,1),f-val=4.16],[(2,0),f-val=4.24],[(1,1),f-val=4.24],[(3,2),f-val=4.41],[(4,2),f-val=5.00],[(0,1),f-val=5.00]]

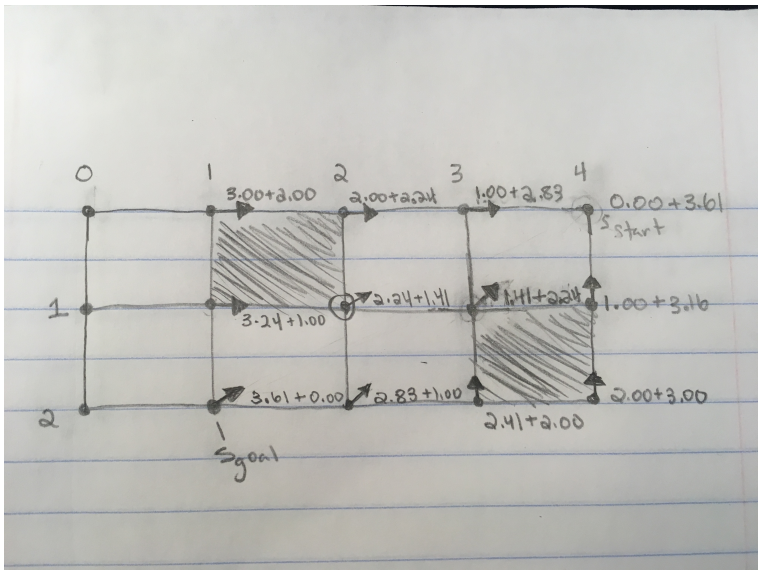


Figure 22: Example Search Problem: After Expanding (2,1)

Pop (1,2) from the openList and expand it. In this case, we have expanded a node that has a lower f-value, than the f-value's of previously popped nodes. For example, (1,2) has an f-value of 3.61, while both (3,1) and (2,1) were expanded with f-values of 3.65. Here, we check if (1,2) is the goal, which it is. The parent of (1,2) is the start node, (4,0). Thus, we can draw a straight line from (4,0) to (1,2), with a cost of 3.61.

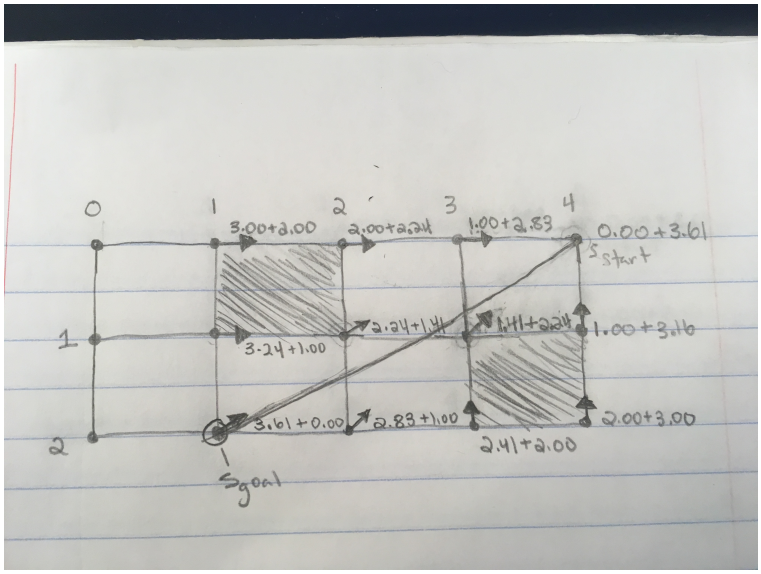


Figure 23: Example Search Problem: Final Path

8 Question 8

Computing the visibility graph is done through connecting all the vertices of the buffered polygons as long as they are not contained within another buffered polygon or outside of the boundary of the world plane (with buffers included). A list of all the possible vertices including the start node and the goal node is created to generate possible edges. The edges between each node in the list is checked with line of sight to determine if a possible path exists between the two nodes. From there, the A* algorithm is conducted on the visibility path to find the shortest path from the start node to the goal node. Displayed below, the red lines are all the paths in the visibility graph and the green line is the shortest path found by using A* on the visibility graph.

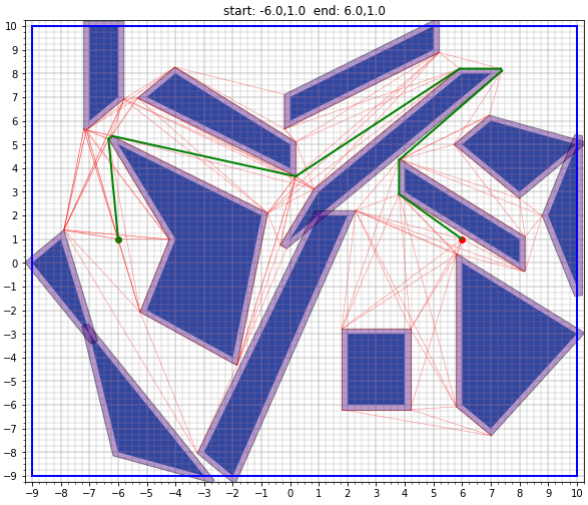


Figure 24: Using World 5

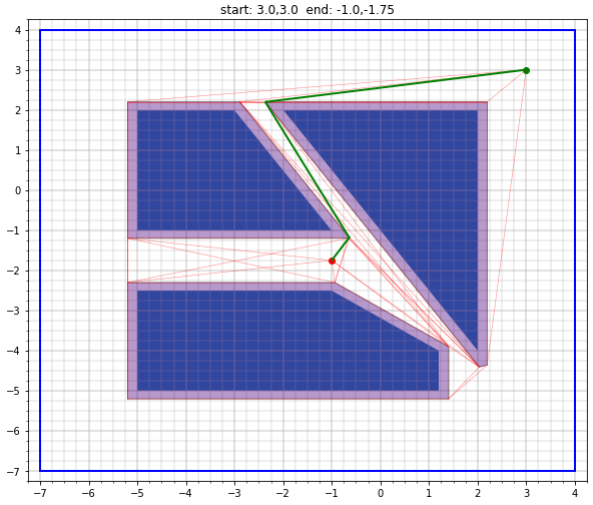


Figure 25: Using World 1

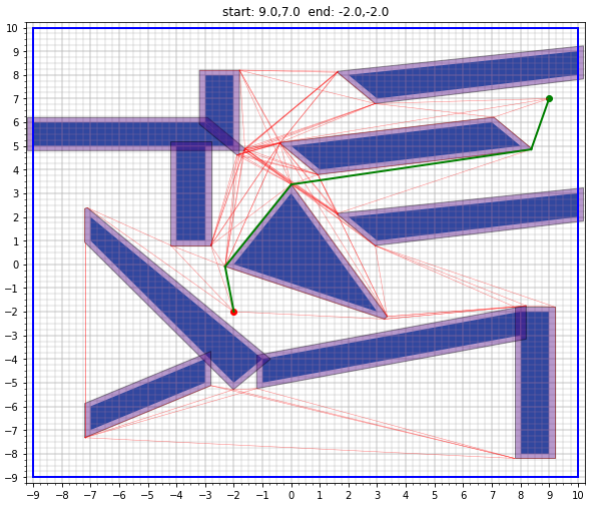


Figure 26: Using World 3

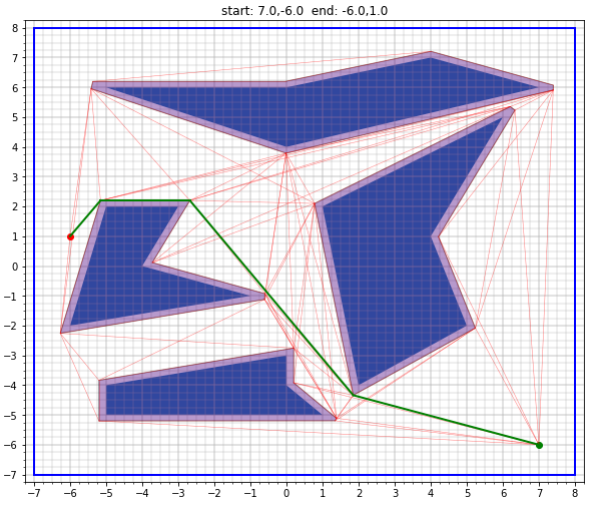


Figure 27: Using World 4

9 Question 9

Below is a table of the experimentally determined times for running A* on the visibility graph and for running FDA*. The result is as expected that the A* on visibility graph is much faster than that of running FDA*. This is due to the fact that the amount of nodes needed to be checked by A* on the visibility graph is a lot lower than with the FDA* which has to check each successor of each node it traverses. In comparison, the A* algorithm only needs to check in the tens amount of nodes while FDA* needs to check in the hundreds.

World	A* on Visibility Graph	FDA*
world 1	0.583000183105	1.74499988556
world 2	1.52599954605	10.253000021
world 3	5.10899996758	46.6860001087
world 4	6.46099996567	36.2369999886