CSCI 395

Patrick Lin, Lameya Mostafa, Lina Tran

Professor Raj Korpan

Lab 5: Path Planning and Motion Planning

October 14, 2025

Comparison Tables

Map, Start/Goal, Algorithm, Path Cost, States Expanded (where applicable), and Planning Time.

DFS

Map	Start	Goal	Path Cost	States Expanded	Planning Time
1	0, 0	8, 8	56.325902	56	0.0015323162078857422s
2	321, 148	106, 202	106011.439401	104283	3.8592565059661865s
3	0, 0	18,18	291.771645	223	0.006867647171020508s
4	1, 1	500, 501	4957.860674	70114	1.5643296241760254s
5	1, 1	24, 24	567.058008	557	0.016300201416015625s
6	5, 79	200, 130	9932.014706	9290	0.2992258071899414s

BFS

Map	Start	Goal	Path Cost	States Expanded	Planning Time

1	0, 0	8, 8	14.242641	64	0.0018568038940429688s
2	321, 148	106, 202	25.455844	361	0.011128664016723633s
3	0, 0	18,18	26.627417	344	0.009998559951782227s
4	1, 1	500, 501	4604.920415	111663	3.165646553039551s
5	1, 1	24, 24	33.698485	573	0.016805410385131836s
6	5, 79	200, 130	264.132034	14684	0.41685032844543457s

IDDFS

Map	Start	Goal	Path Cost	States Expanded	Planning Time
1	0, 0	8, 8	21.798990	37	0.008877038955688477s
2	321, 148	106, 202	69.053824	444	0.23755288124084473s
3	0, 0	18,18	61.154329	317	0.10806894302368164s
4	1, 1	500, 501	4957.860674	56761	2530.7819125652313s
5	1, 1	24, 24	105.580736	532	0.38658881187438965s
6	5, 79	200, 130	6335.510237	11360	1129.4090323448181s

Thinking Question: For map1.txt, how do the paths generated by BFS and DFS differ? Explain why this difference occurs based on how each algorithm explores the search space.

In terms of computational time, both algorithms perform relatively similarly. However, BFS demonstrates a clear advantage by producing a path with lower cost, despite expanding only a few more states than DFS. The path generated by BFS is noticeably more optimal compared to that of DFS. This difference arises from the way each algorithm explores the search space: DFS follows one branch to its deepest point before backtracking, often resulting in longer or less efficient paths, while BFS explores all possible paths level by level, ensuring the shortest route to a given point is found. As a result, BFS yields a more optimal path than DFS.

A*
Testing with epsilon 1, 10, 20 on maps 1, 2, 3

Map	Epsilon	Path Cost	States Expanded	Planning Time
1	1	14.242641	52	0.0015659332275390625s
1	10	15.071068	21	0.0007150173187255859s
1	20	15.071068	21	0.0006778240203857422s
2	1	251.450793	17812	0.5656468868255615s
2	10	257.308658	1225	0.054204463958740234s
2	20	256.722871	1592	0.07567930221557617s
3	1	26.627417	68	0.002390623092651367s
3	10	28.384776	24	0.0009186267852783203s
3	20	28.384776	24	0.0016438961029052734s

Thinking Question: How does increasing the value of ϵ affect the trade-off between path optimality and search speed? Explain why this behavior is observed.

As seen in the table, increasing epsilon (ε) in A* makes the search faster but slightly less optimal. With ε = 1 on Map 2, I expanded over 17,000 states and got a near-optimal path, taking 0.565 seconds. Increasing ε to 10 or 20 reduced the states expanded to around 1,200–1,500 and cut planning time to under 0.08 seconds, but the path cost increased slightly.

I see that higher ε makes A* act greedier, prioritizing nodes closer to the goal and ignoring some lower-cost paths. So, I trade path optimality for speed: small ε gives a better path but slower search, while large ε is faster but less precise.

General Table with eps 1

Map	Start	Goal	Path Cost	States Expanded	Planning Time
1	0, 0	8, 8	14.242641	52	0.0017642974853515625s
2	321, 148	106, 202	251.450793	17812	0.5826704502105713s
3	0, 0	18,18	26.627417	68	0.002192258834838867s
4	1, 1	500, 501	4604.920415	95085	2.6745760440826416s
5	1, 1	24, 24	33.698485	84	0.003564119338989258s
6	5, 79	200, 130	264.132034	8117	0.2520291805267334s

RRT

Map	Start	Goal	Path Cost	States Expanded	Planning Time	bias/eta
1	0, 0	8, 8	15.965089 5653485	50	0.0119705 200195312 5 s	Bias 0.05 eta 1
2	321, 148	106, 202	307.23056 40617844	2822	20.444247 484207153 s	bias 0.1 eta 5
3	0, 0	18,18	32.381476 86688772	67	0.0121450 424194335 94 s	bias 0.05 eta 1.5
4 (didn't reach goal)	1, 1	500, 501	2.0	2	24.262885 808944702 s	bias 0.1 eta 10
5	1, 1	24, 24	40.461482 861348564	86	0.0094492 435455322 27s	bias 0.05 eta 2
6	5, 79	200, 130	334.94604 685176455	885	1.0324058 532714844 s	bias 0.1 eta 5

Thinking Question: What is the purpose of goal biasing in RRT? How does it affect the planner's speed and the shape of the tree?

Goal biasing in RRT helps the planner focus some of its samples toward the goal rather than purely random exploration. For example, on Map 1 with bias 0.05, the planner quickly found a path in 0.012 s, expanding only 50 states. On Map 4, even with a higher bias of 0.1 and large eta, it didn't reach the goal, showing that bias alone can't guarantee success in very large or complex maps. I notice that increasing the goal bias tends to make the tree grow more directly toward the goal, which can speed up planning and reduce expanded states. However, too much bias might limit exploration, causing the planner to miss alternative paths around obstacles. So, goal bias is a trade-off: higher bias improves speed toward the goal but may reduce coverage of the free space.

PRM

Map	Start	Goal	Path Cost	States Expanded	Planning Time	num_samples/neighbors
1	0, 0	8, 8	16.828427	52	0.017093181610107422s	s 100 n 5
2	321, 148	106, 202	252.622775	318	2.806387424468994s	s 500 n 10
3	0, 0	18,18	28.079835	201	0.07317137718200684s	s 200 n 8
4 (no path found)	1, 1	500, 501	0.000000	1	5.003888368606567s	s 1000 n 15
5	1, 1	24, 24	36.003369	202	0.09363102912902832s	s 200 n 8
6	5, 79	200, 130	302.451896	750	1.307307481765747s	s 800 n 12

Thinking Question: PRM has two key parameters: num_samples and num_neighbors. How does increasing each of these affect the quality of the roadmap and the overall planning time

From the table and graphs below it can be seen that increasing num_samples generally improves the quality of the roadmap because more points give the planner better coverage of free space. For example, on Map 2, using 500 samples instead of 100 (like Map 1) allowed PRM to find a path with fewer states expanded (318 vs 52) and a reasonable path cost (252.6). Similarly, higher num_neighbors increases connectivity between nodes, making it more likely that the planner can find a valid path through complex spaces, like Map 6 with 12 neighbors.

However, both higher num_samples and more neighbors increase planning time. For instance, Map 4 had 1000 samples and 15 neighbors, and although it still didn't find a path, the planning took 5s, much longer than maps with fewer samples or neighbors. So, increasing these parameters improves roadmap completeness and path quality but at the cost of longer computation times. It's a trade-off between efficiency and success rate.

Potential Fields

Map	Start	Goal	Path Cost	States Expanded	Planning Time
1	0, 0	8, 8			
2	321, 148	106, 202			
3	0, 0	18,18			
4	1, 1	500, 501			
5	1, 1	24, 24			
6	5, 79	200, 130			

Visibility Graph

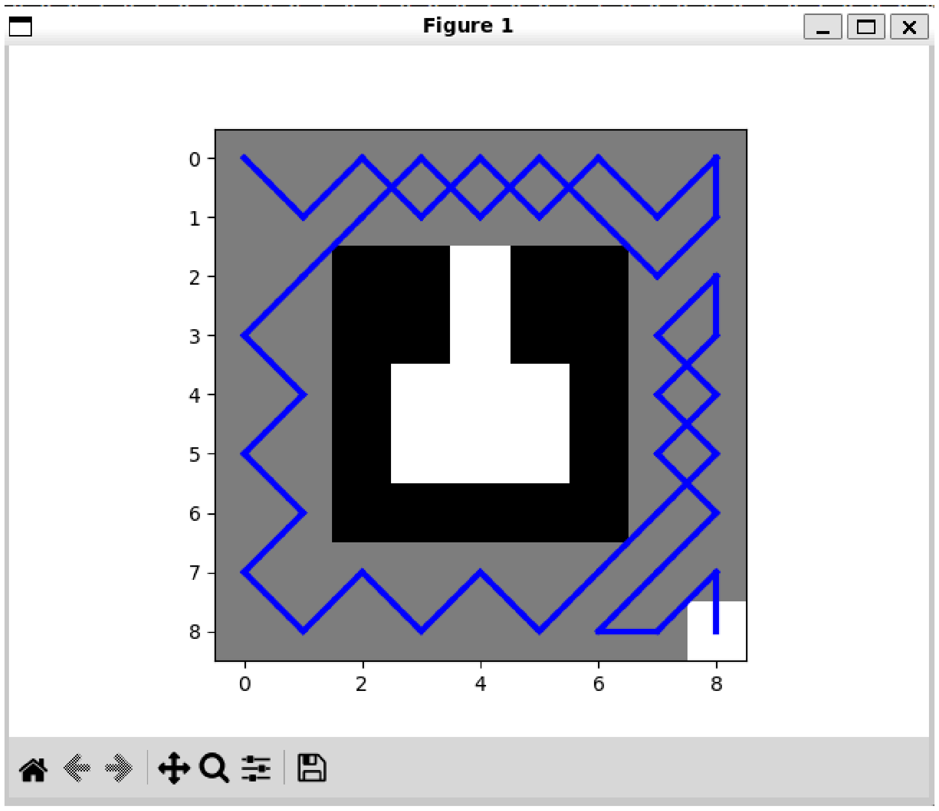
Map	Start	Goal	Path Cost	States Expanded	Planning Time
1	0, 0	8, 8			
2	321, 148	106, 202			
3	0, 0	18,18			
4	1, 1	500, 501			
5	1, 1	24, 24			
6	5, 79	200, 130			

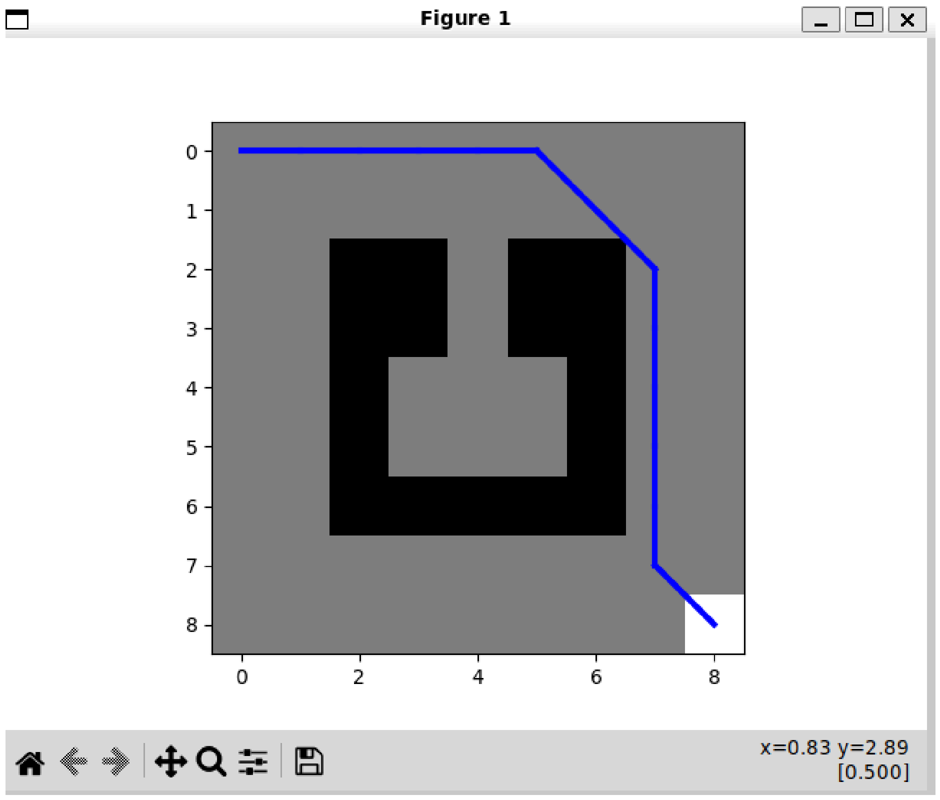
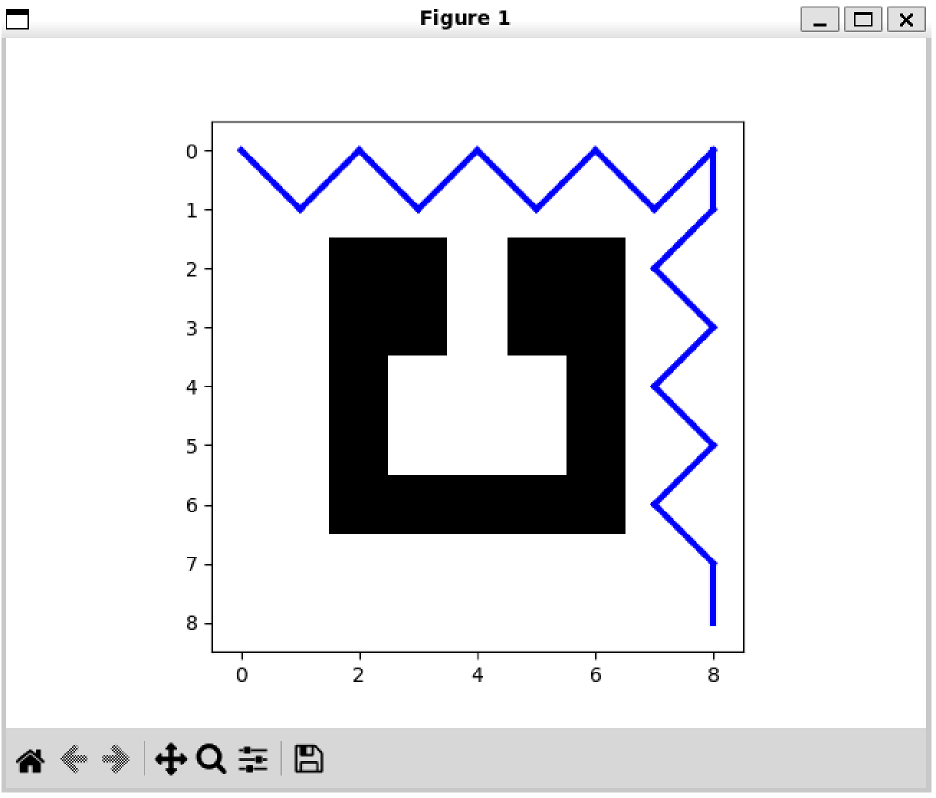
Bug2

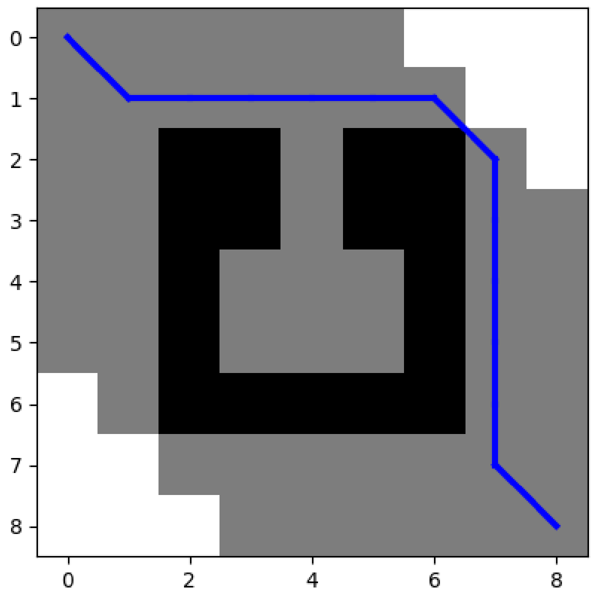
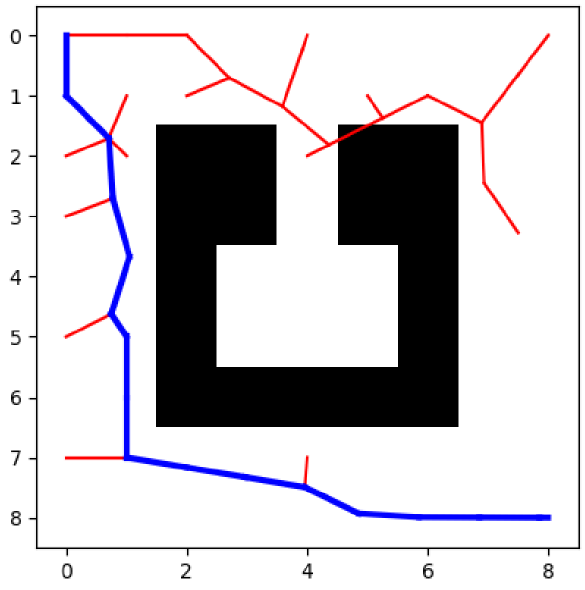
Map	Start	Goal	Path Cost	States Expanded	Planning Time
1	0, 0	8, 8			
2	321, 148	106, 202			
3	0, 0	18,18			
4	1, 1	500, 501			
5	1, 1	24, 24			
6	5, 79	200, 130			

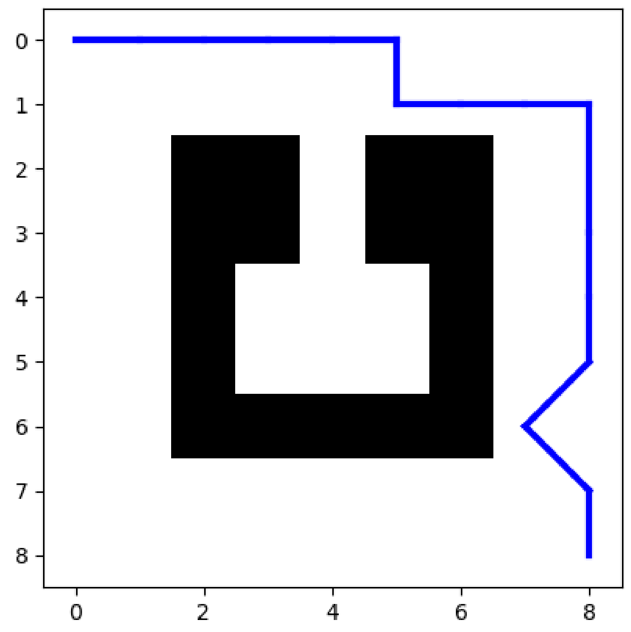
Path Plots

Map 1

Algorithm	Path
DFS	 Figure 1 0 1 2 3 4 5 6 7 8 0 2 4 6 8

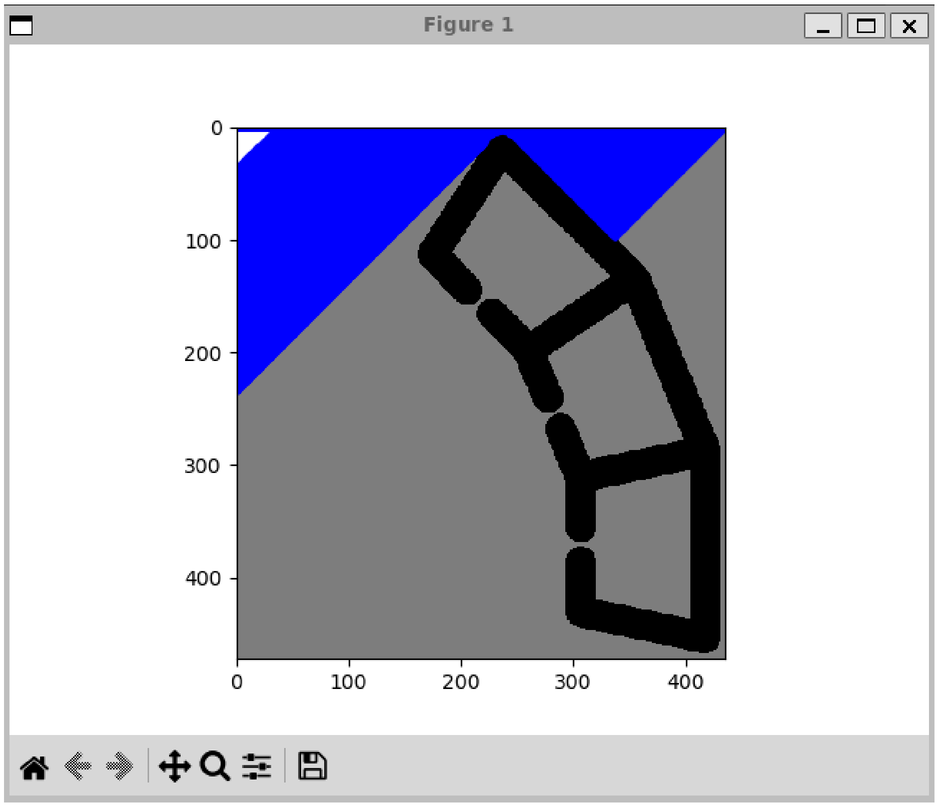
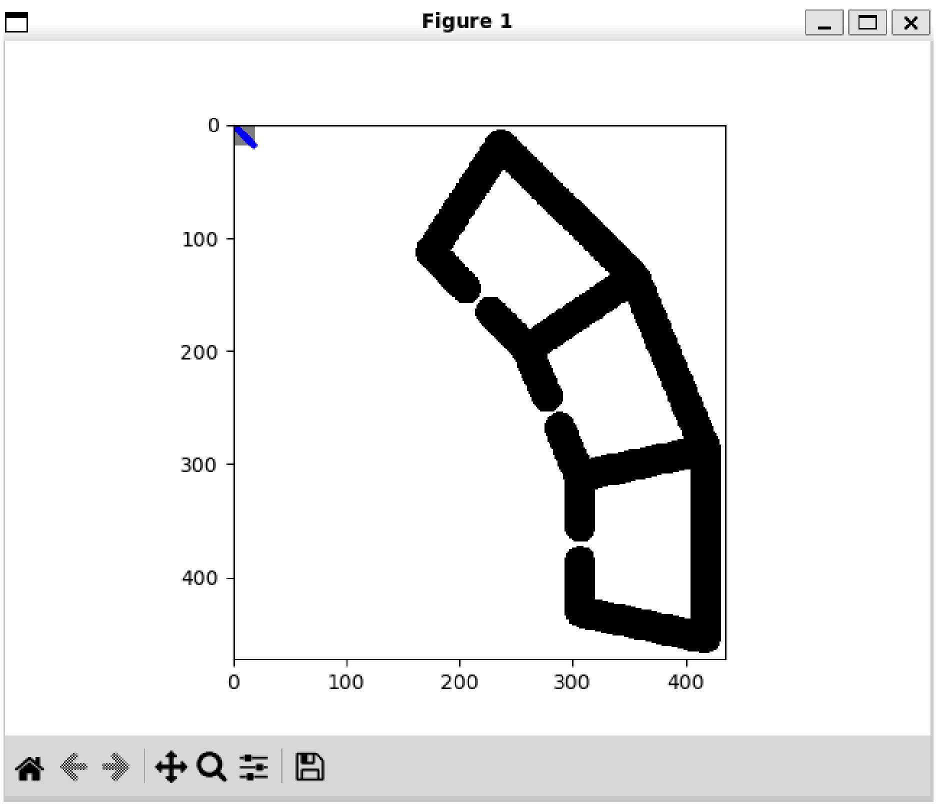
BFS	 Figure 1 0 1 2 3 4 5 6 7 8 0 2 4 6 8 x=0.83 y=2.89 [0.500]
IDDFS	 Figure 1 0 1 2 3 4 5 6 7 8 0 2 4 6 8

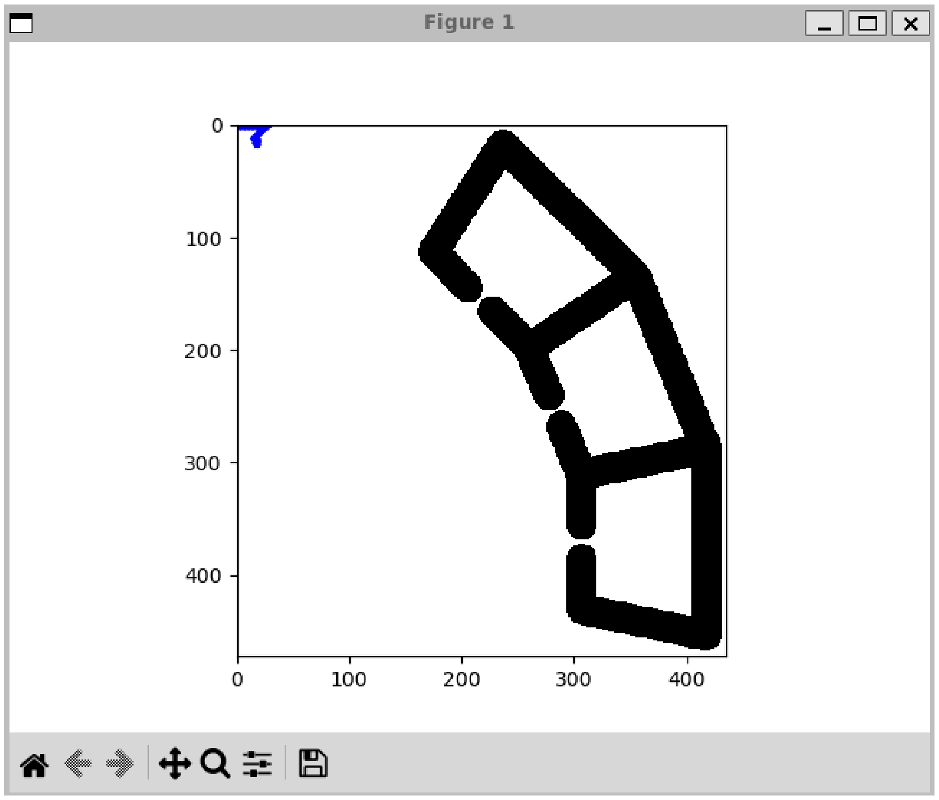
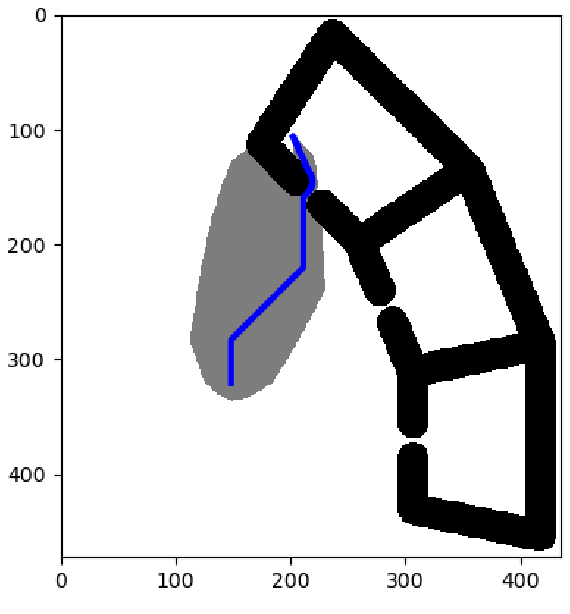
A*	
RRT	

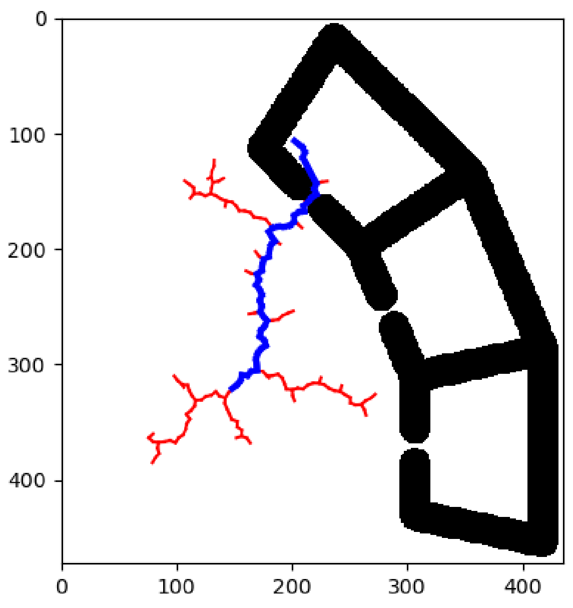
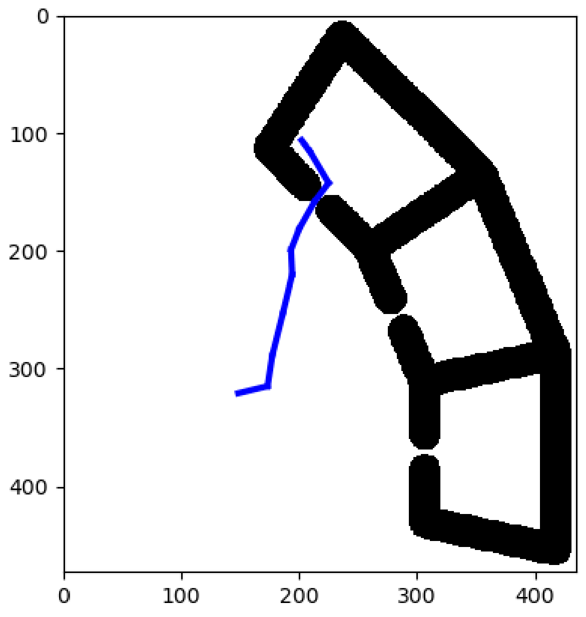
PRM	
Potential Fields	
Visibility Graph	
Bug2	

Map 2

Algorithm	**Path**

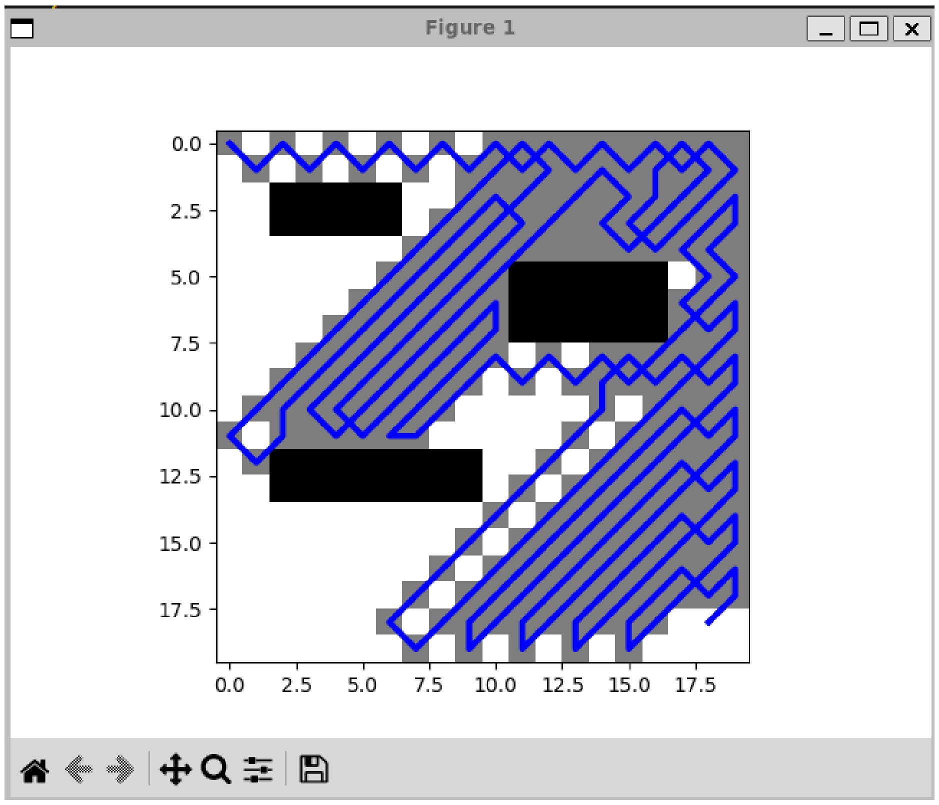
DFS	Figure 1 0 100 200 300 400 0 100 200 300 400
BFS	Figure 1 0 100 200 300 400 0 100 200 300 400

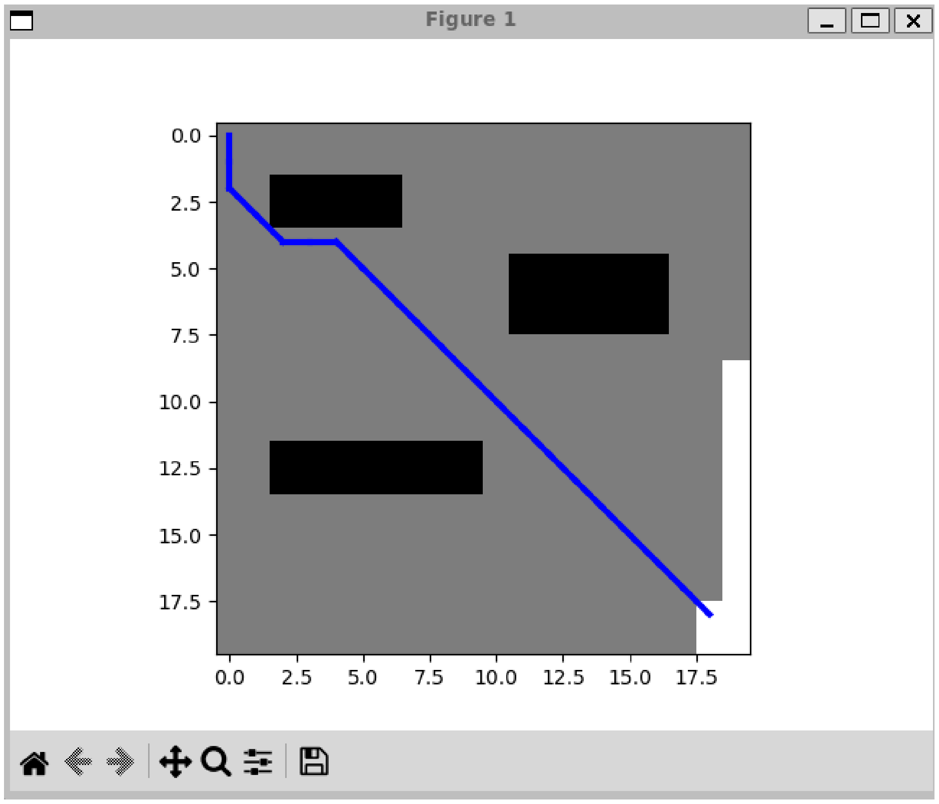
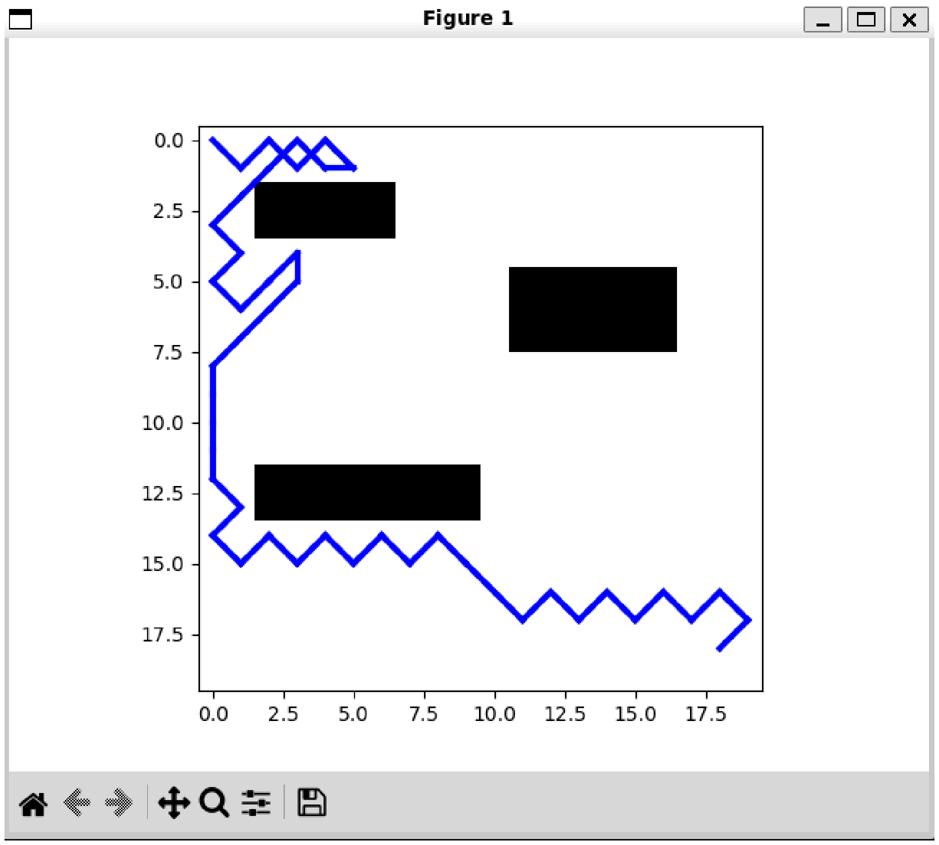
IDDFS	Figure 1 0 100 200 300 400 0 100 200 300 400
A*	0 100 200 300 400 0 100 200 300 400

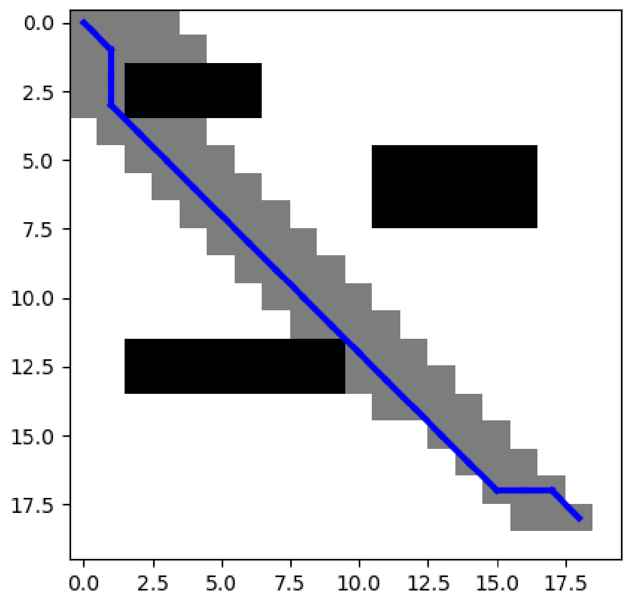
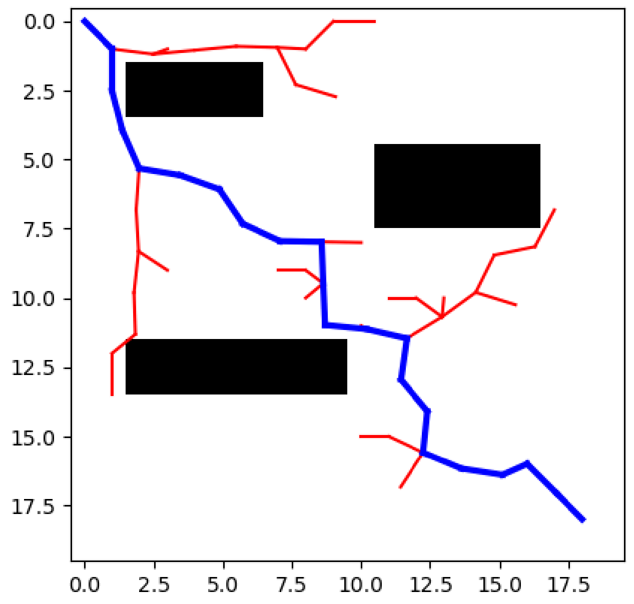
RRT	
PRM	
Potential Fields	

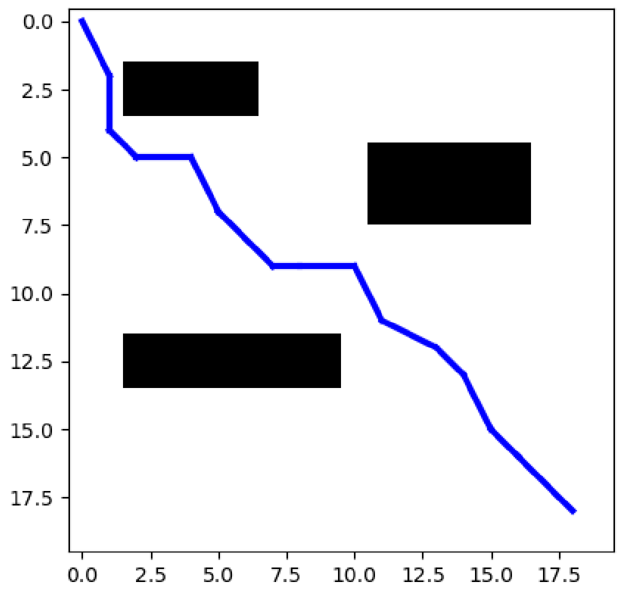
Visibility Graph	
Bug2	

Map 3

Algorithm	**Path**
DFS	 Figure 1 0.0 2.5 5.0 7.5 10.0 12.5 15.0 17.5 0.0 2.5 5.0 7.5 10.0 12.5 15.0 17.5

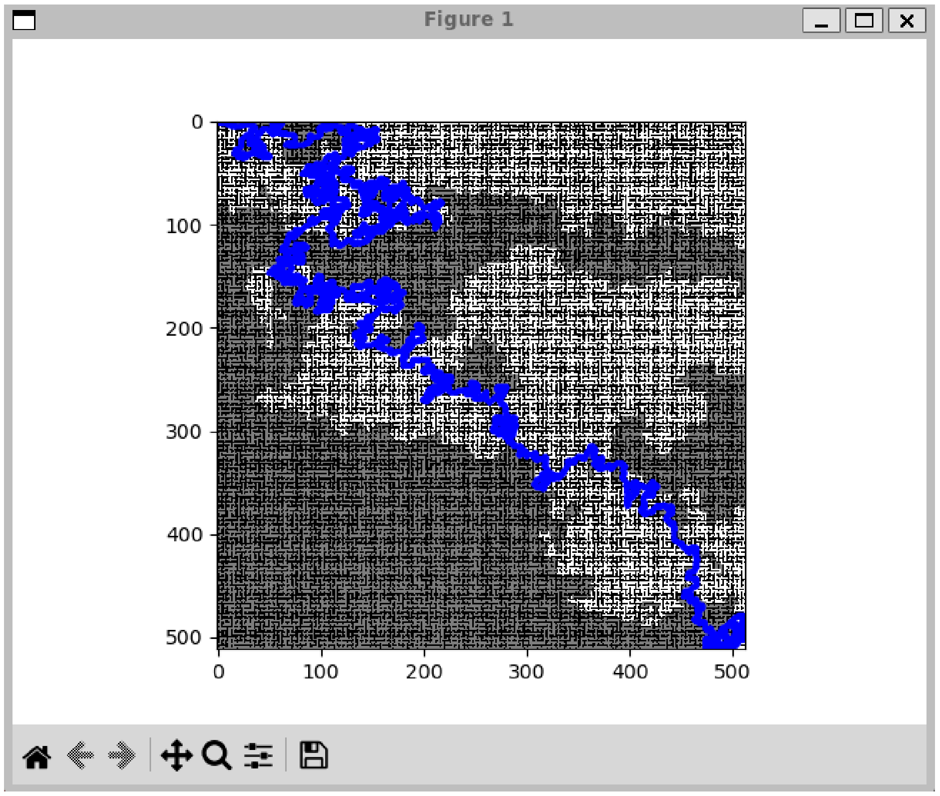
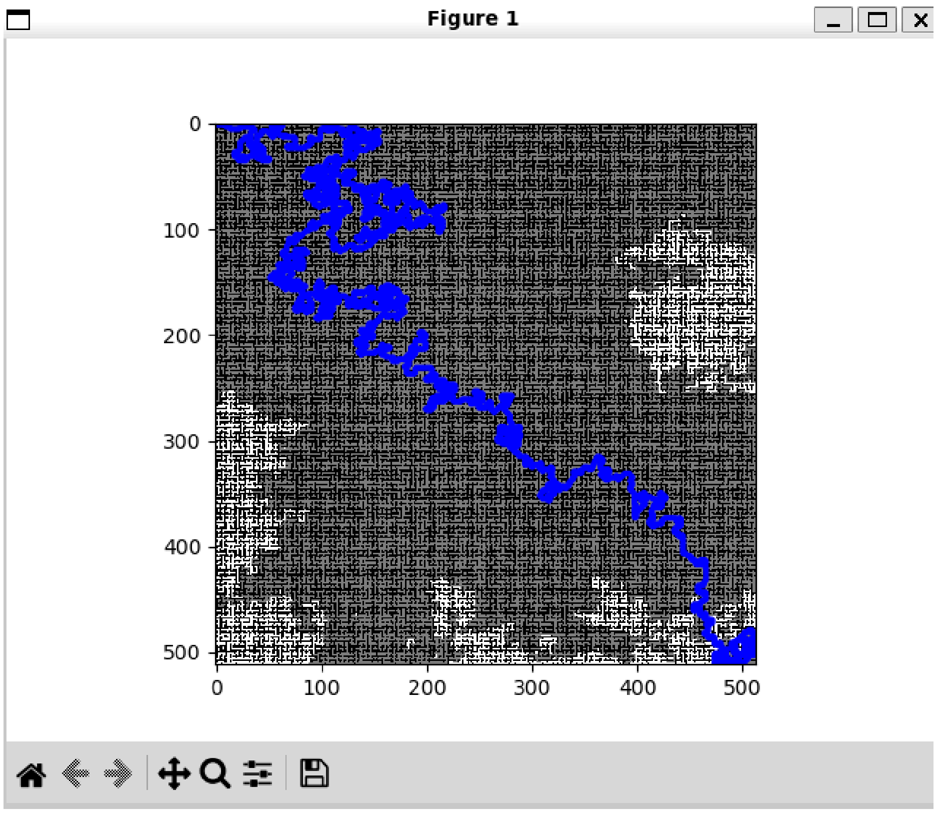
BFS	Figure 1 0.0 2.5 5.0 7.5 10.0 12.5 15.0 17.5
IDDFS	Figure 1 0.0 2.5 5.0 7.5 10.0 12.5 15.0 17.5

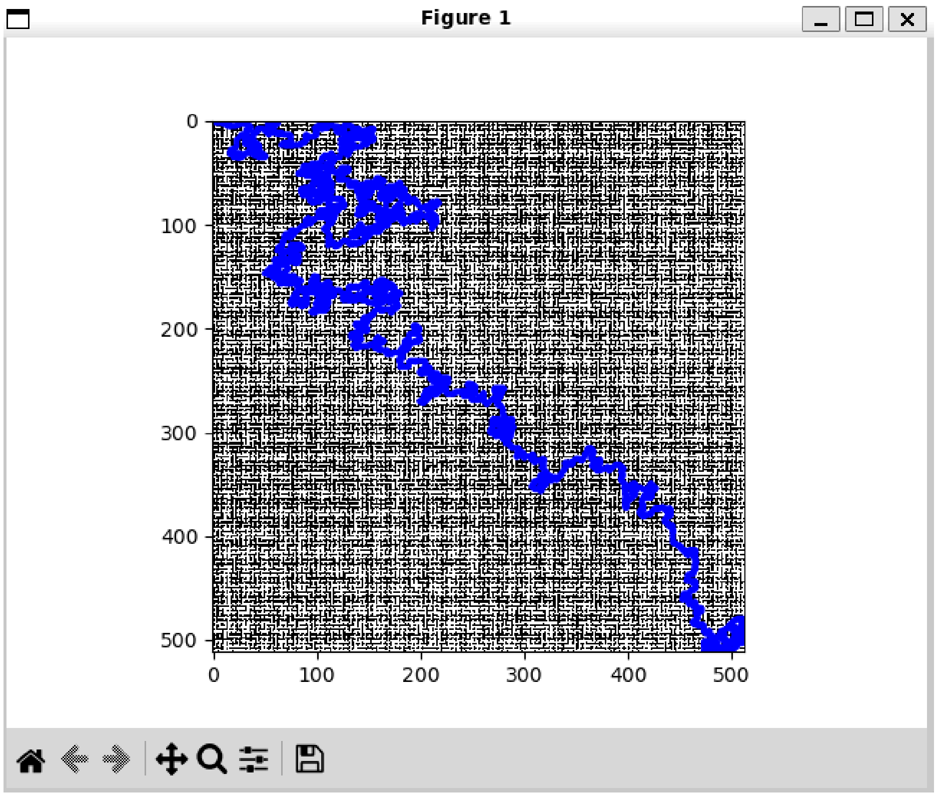
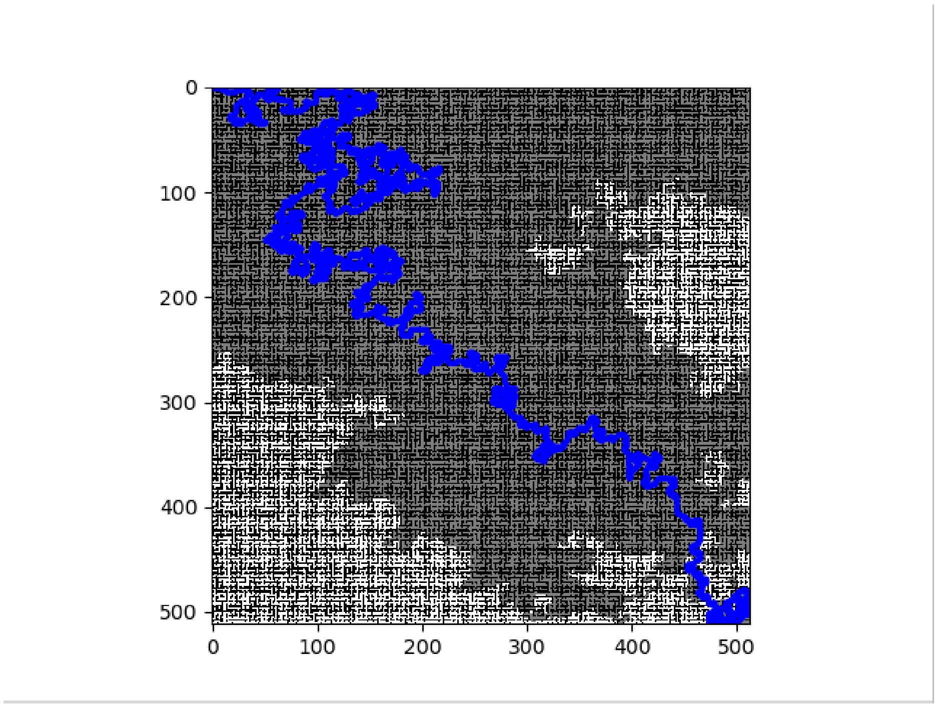
A*	
RRT	

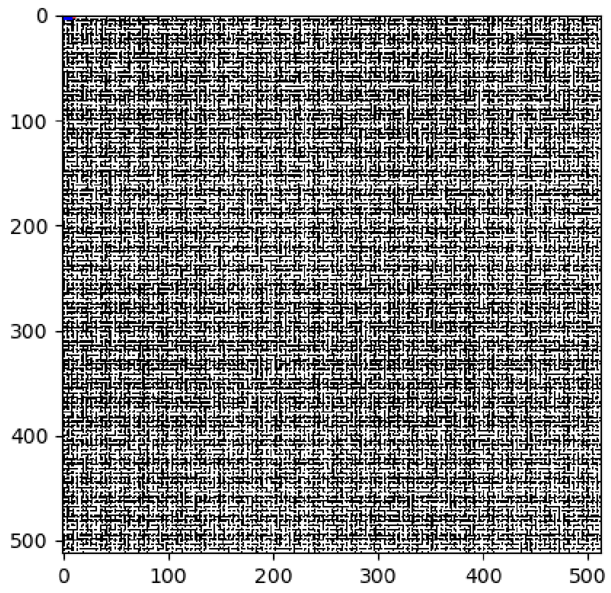
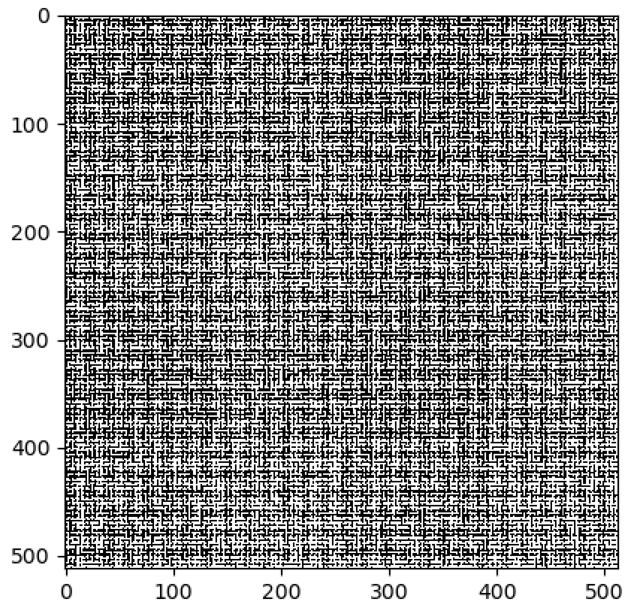
PRM	
Potential Fields	
Visibility Graph	
Bug2	

Map 4

Algorithm	**Path**

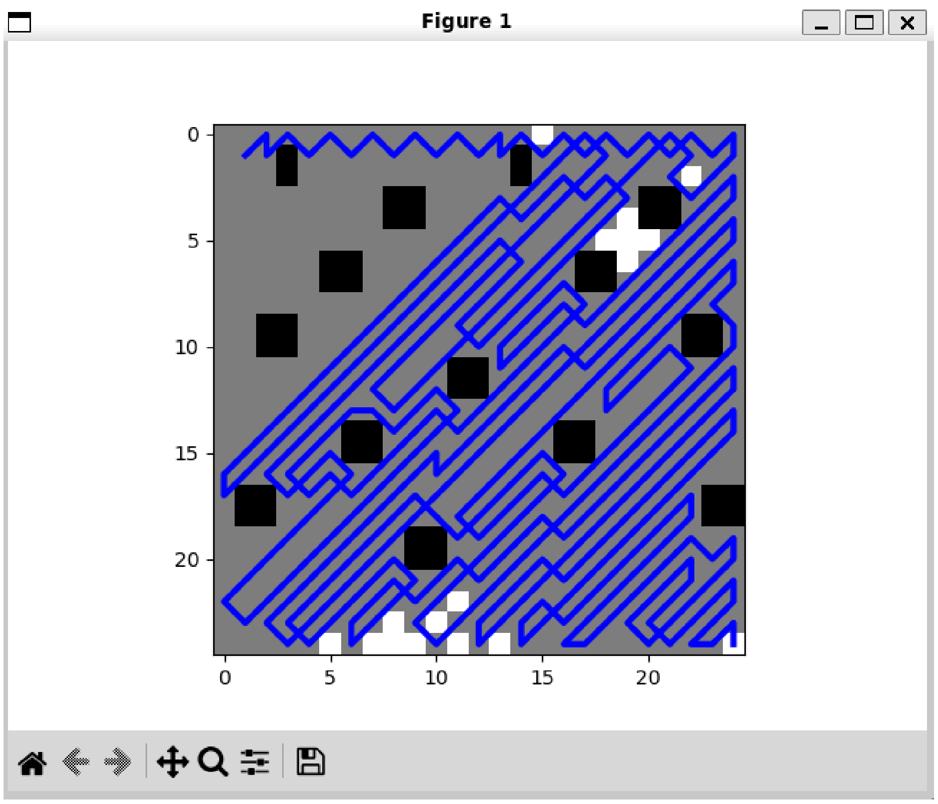
DFS	Figure 1 0 100 200 300 400 500 0 100 200 300 400 500
BFS	Figure 1 0 100 200 300 400 500 0 100 200 300 400 500

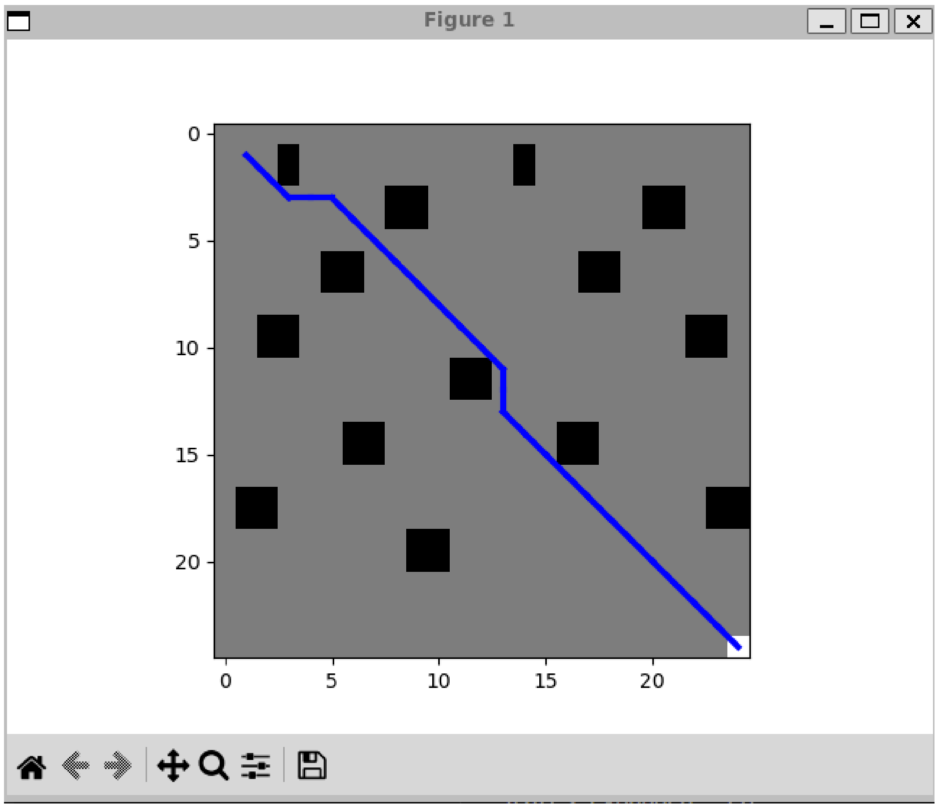
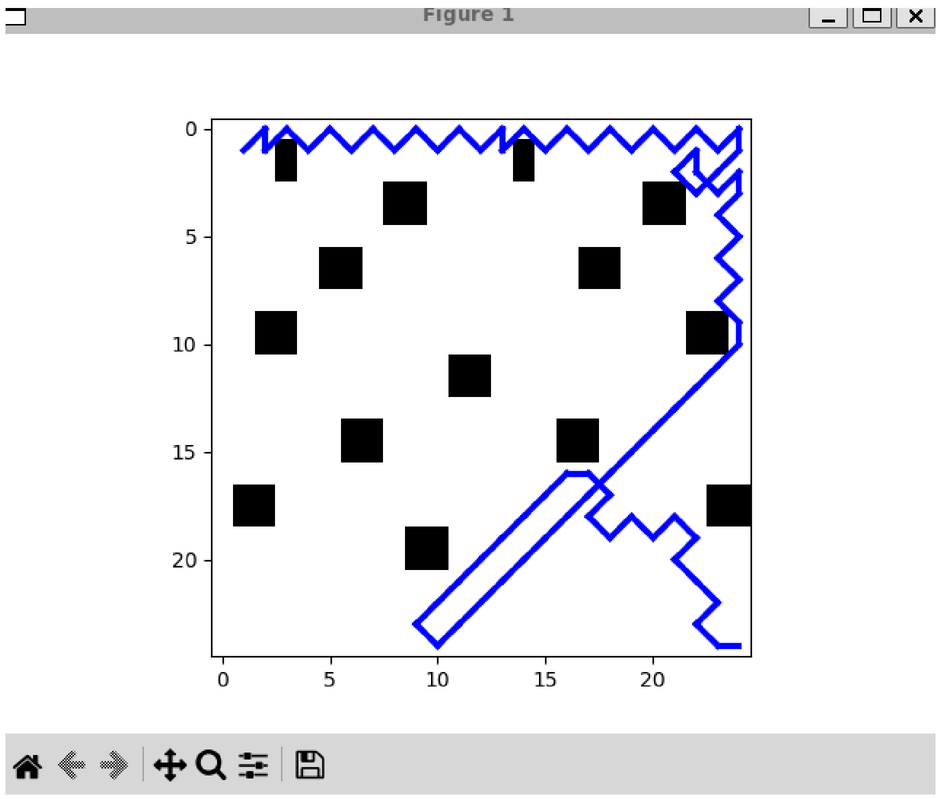
IDDFS	Figure 1 0 100 200 300 400 500 0 100 200 300 400 500
A*	0 100 200 300 400 500 0 100 200 300 400 500

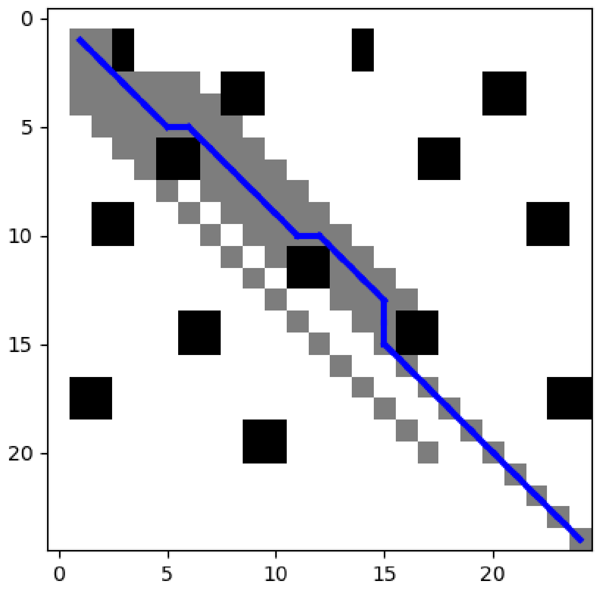
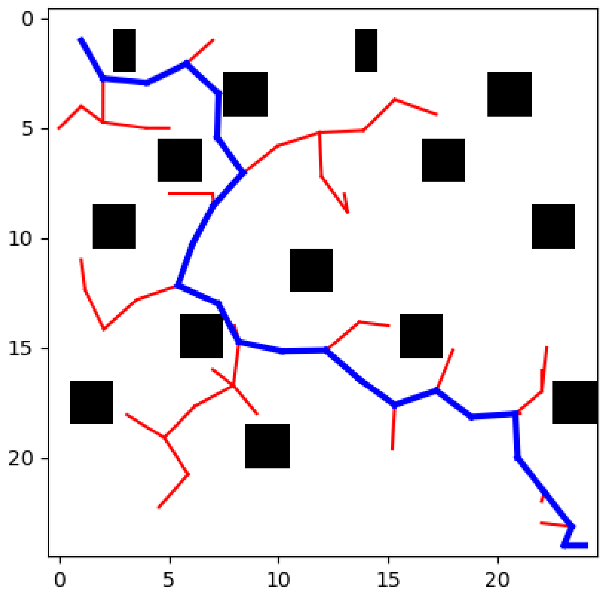
RRT	
PRM	
Potential Fields	

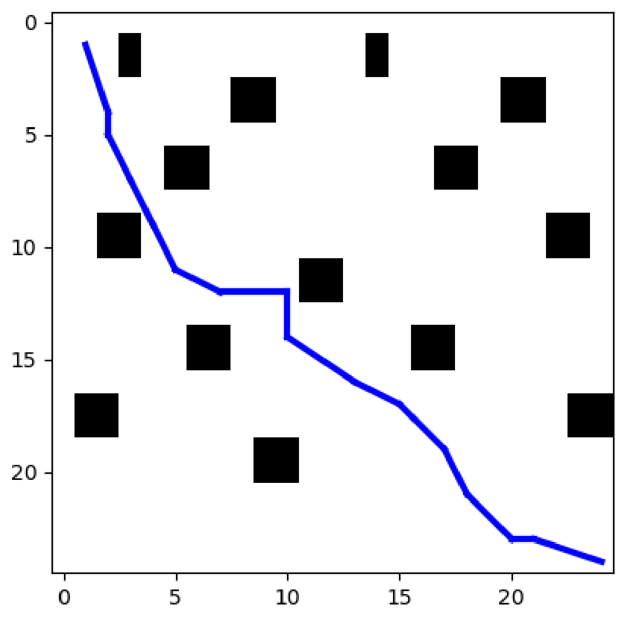
Visibility Graph	
Bug2	

Map 5

Algorithm	**Path**
DFS	 Figure 1 0 5 10 15 20 0 5 10 15 20

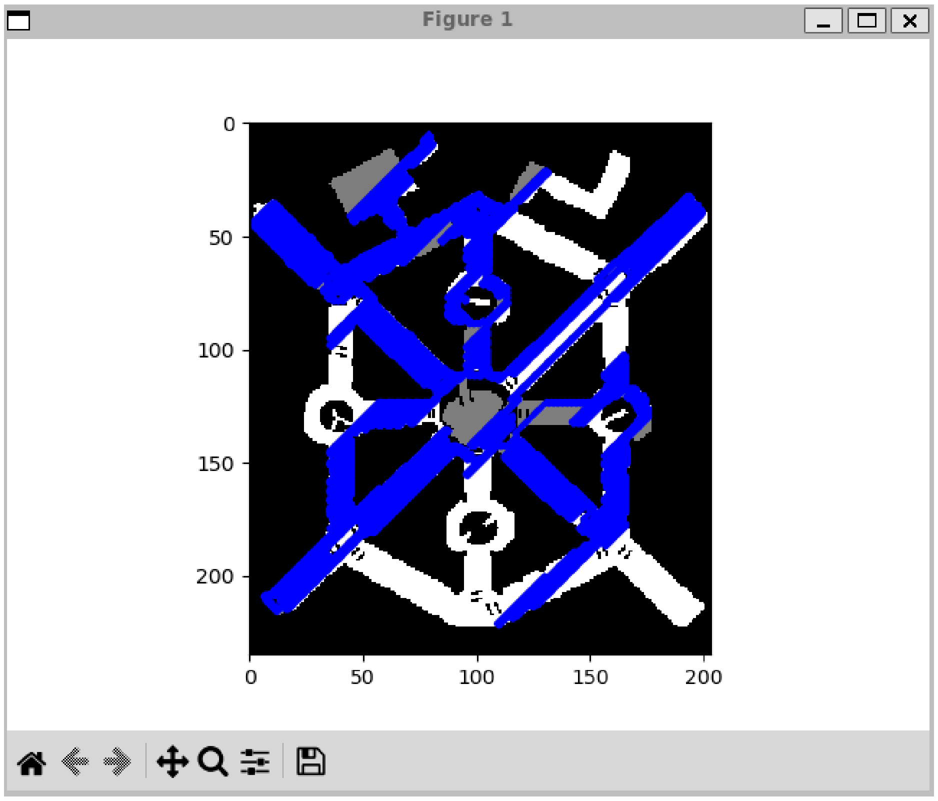
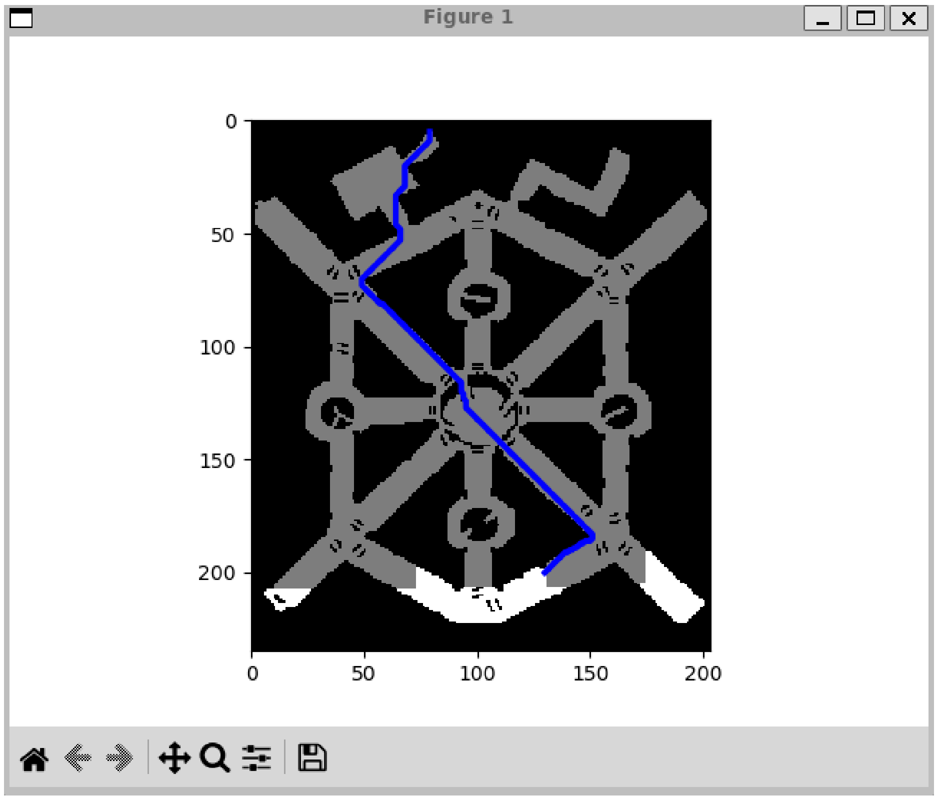
BFS	 Figure 1 0 5 10 15 20 0 5 10 15 20
IDDFS	 Figure 1 0 5 10 15 20 0 5 10 15 20

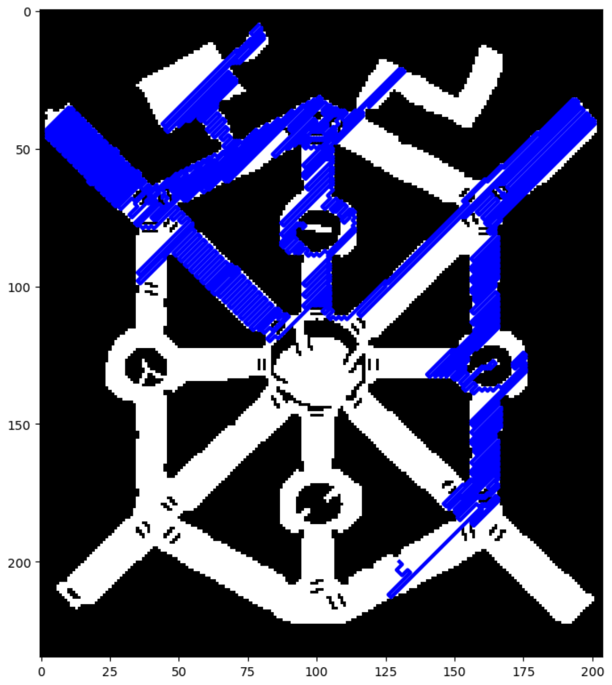
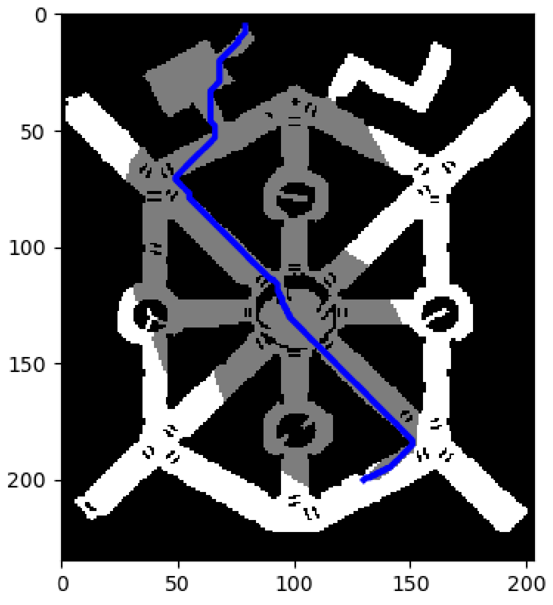
A*	
RRT	

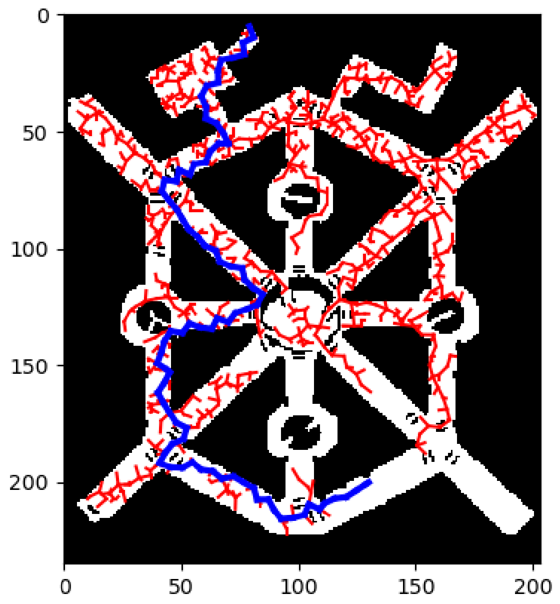
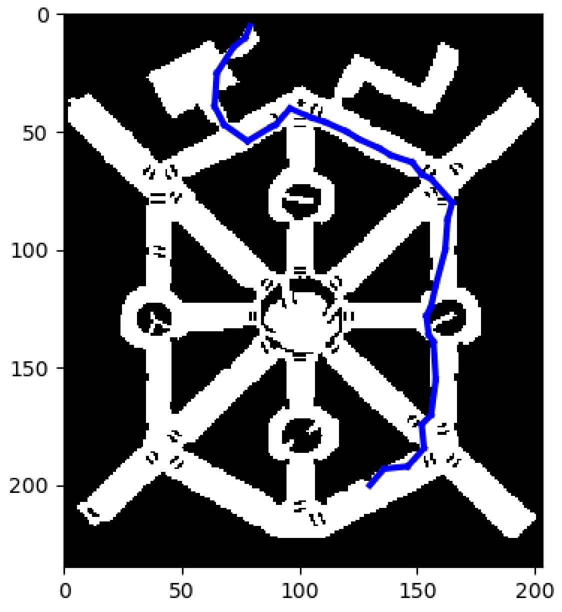
PRM	
Potential Fields	
Visibility Graph	
Bug2	

Map 6

Algorithm	**Path**

DFS	Figure 1 0 50 100 150 200 0 50 100 150 200
BFS	Figure 1 0 50 100 150 200 0 50 100 150 200

IDDFS	0 50 100 150 200 0 25 50 75 100 125 150 175 200
A*	0 50 100 150 200 0 50 100 150 200

RRT	
PRM	
Potential Fields	
Visibility Graph	

Bug2	

Analysis and Discussion

Which algorithms failed to find a solution on certain maps, and why?

RRT failed to reach the goal on map 4 (the large maze) because its random sampling and incremental growth couldn't find a collision-free path through the narrow corridors within the maximum iterations. PRM also failed on map 4 because even with 1000 samples and 15 neighbors, the roadmap didn't include a path connecting start and goal in the maze.

How did performance change between different map types (e.g., maze vs. open space)?

On open or simple maps (maps 1, 3, 5), all algorithms found paths quickly, with few states expanded. On large or complex maps (maps 2, 4, 6), planning time and states expanded increased, especially for sampling-based methods, due to more obstacles and longer paths.

Which algorithm was consistently the fastest? Which was the most optimal?

A* was consistently the fastest in most maps, particularly with higher epsilon values, and also produced near-optimal paths in simple maps. PRM and RRT were slower, with RRT being particularly inefficient in complex mazes. PRM produced fairly good paths when a roadmap existed, but could fail if sampling missed key areas.

Discuss the trade-offs you observed across all your tests.

DFS: DFS traded off path cost, number of states expanded, and planning time in exchange for simply finding a valid path. While it efficiently located a feasible route, its performance degraded significantly on large or complex maps compared to smaller or more open ones.

BFS: BFS consistently produced the optimal path from the start to the goal. However, this came at the cost of increased planning time and, in some cases, a higher number of states visited. In return, it achieved a lower overall path cost and a more efficient route.

IDDFS: Produced a path with optimality between that of DFS and BFS, effectively trading increased planning time for a lower path cost. However, on larger maps, IDDFS's performance declined compared to both DFS and BFS due to the significant overhead caused by repeated explorations at increasing depth limits.

A*: Fast and optimal on simple maps; performance degrades on very large maps if epsilon is low.

RRT: Good for high-dimensional or open spaces, but may fail in narrow or maze-like environments; planning time and path quality depend on bias and eta.
PRM: Generates good paths if enough samples and neighbors are used, but high sampling increases planning time; fails if roadmap doesn't connect start and goal.

Final Reflection Question: Based on your comprehensive results, create a "planner selection guide." For a given type of environment (e.g., an open field, a narrow maze, a cluttered warehouse, a time-critical emergency) and a primary goal (e.g., shortest path, fastest computation), which planning algorithm would you recommend and why? Justify your choices with data from your comparison table.

Tiny Map (map1)

Small Map (map3, map5)

Large Map with Some Obstacles (map2)

Large Map with Lots of Obstacles (map4, map6)

Methods: Briefly describe how you implemented each of the nine algorithms.

DFS: We implemented DFS using a stack. The stack was initialized with the start location and its corresponding parent (which is None for the starting point). The algorithm then entered a loop that continued running as long as the stack was not empty. Inside the loop, we would get the last location added to the stack. Then, we checked whether the current location was valid - that is, within the map boundaries, not an obstacle, and not already visited. If the location was invalid, we moved on to the next element in the stack.

If the location was valid, we incremented the state counter and checked whether it matched the goal condition using the goal_criterion function. If the goal was reached, we generated the final path by calling the tracePlan helper function, which backtracked from the goal through its parent nodes to reconstruct the route. Otherwise, we marked the current location as visited and added all possible neighboring locations (including diagonals) to the stack. Each newly added location was stored along with its parent in a dictionary to keep track of the path taken.

BFS: We implemented Breadth-First Search (BFS) using a queue, following a process similar to DFS with a few key differences. Instead of retrieving the most recently added location (as in DFS), BFS retrieves the oldest location added to the queue. Additionally, before adding the current location and its child nodes to the tracking dictionary, we first check whether each child is already present in the dictionary. If a child node is already recorded, it means a shorter path to that location has already been found, so there is no need to overwrite it. In contrast, the DFS implementation did not include this check, as its primary goal was simply to find a path rather than the most efficient one.

IDDFS: For the IDDFS implementation, we first created a helper function similar to our DFS implementation, with a few key modifications. The helper function, dfs, accepted an additional parameter called max_depth. When adding elements to the stack, we included the current location, its parent, and the current depth. If the current depth exceeded max_depth, the algorithm would stop expanding that branch and would not add the node's children to the stack.

In the main iddfs function, we used a loop that repeatedly called the helper function until it returned a valid plan. If no valid plan was found, the max_depth value was incremented, the self.visited set (which tracked visited locations) was reset, and the search was retried with the increased depth limit.

A*: for the A* I used a priority queue where nodes are expanded based on f = g + epsilon * h, with g as the path cost so far and h as the heuristic (Euclidean distance to goal). I kept track of visited nodes in a grid and reconstructed the path by backtracking from the goal once it was reached.

RRT:. In each iteration, I sampled a random point (with goal bias) and extended the nearest tree node toward it by a fixed distance eta. New nodes were added only if the edge was collision-free. I repeated this until the goal was reached or the maximum iterations were exhausted.

PRM: I sampled a fixed number of collision-free points in the environment. Each point was connected to its k nearest neighbors if the edge was valid, forming a roadmap graph. I then searched this graph using Dijkstra's algorithm to find the shortest path from start to goal.

(also received help from chatgpt in implementing these algorithms)

Results: Present your comprehensive comparison table and all the required path plots.

(shown in the table above)

Analysis: Provide a detailed discussion of your findings as described in Part 6.

(answered in the analysis and discussion section above)

Challenges: Describe any significant problems you encountered and how you solved them.

One challenge we had was working with NumPy arrays, since their indexing and shapes behave differently from standard Python lists. Another challenge was translating the route from the planner into the correct format for visualization. The original plan output sometimes needed reshaping or concatenating differently, so we had to use functions like np.concatenate or reshape to return a [2 x n] array that the visualizer could properly display.

Division of Labor: Clearly state who was responsible for which parts of the lab.
Patrick: Part 1, Part 2
Lameya: Part 3, Part 4
Lina: Part 5, Part 6

Reflection Questions: Provide thoughtful answers to all the "Thinking Questions" and the "Final Reflection Question" from the lab.

(answered above)