Robot Motion Planning - Homework II

David Akhihiero

September 22, 2024

1 Introduction

In this assignment, I implemented the visibility graph roadmap algorithm on a 2D configuration space with obstacles. I also implemented the breadth-first search (BFS), Dijkstra, and A* motion planning algorithms for searching in the visibility graph from a start node to a goal node and I compared the performance of these algorithms with each other and with the A* algorithm in the PGraph class of Peter Corke's Robotics toolbox.

2 Graph Creation

To generate the graph, I wrote a $createVisibilityGraph(new\_obstacles,\ dense)$ function. The function takes two boolean parameters; $new\_obstacles$ and $dense$ to decide if a user wants to create a new environment using the $createObstacles()$ function or reload an already created environment (either the dense environment or the sparse environment). If $new\_obstacles$ is true, the user can draw new obstacles and if false, depending on the value of $dense$ either the sparse environment or the dense environment is loaded. After the environment obstacles are loaded, the $createVisibilityGraph(new\_obstacles,\ dense)$ function connects all the nodes in the environment as long as the edge connecting two nodes does not cross an obstacle. To check if an edge crosses an obstacle, I used a function called $discretizeLine(p1,\ p2,\ stepsize)$ adapted from this example code MathWorks (2024). I modified the code so it doesn't break when the edge is vertical. The function generates a list of points along the edge bordered by $p1$ and $p2$ with the points $stepsize$ away from each other. I used the $isFree(q)$ function to check if any of those points are inside an obstacle and if none are, the edge is valid and can be added to the graph.

I also wrote an $addNode()$ function to add new nodes to this graph; namely, the start node and the goal node. The function also connects the additional nodes to all other nodes as long as the edges between nodes are valid. The results of these functions for the sparse environment and the dense environment are shown in Figure 1.

3 Breadth First Search (BFS)

Following the class example, I wrote a $BFS()$ function that plans a path from a start node to a goal node on the graph. My function included a stopping criteria for when

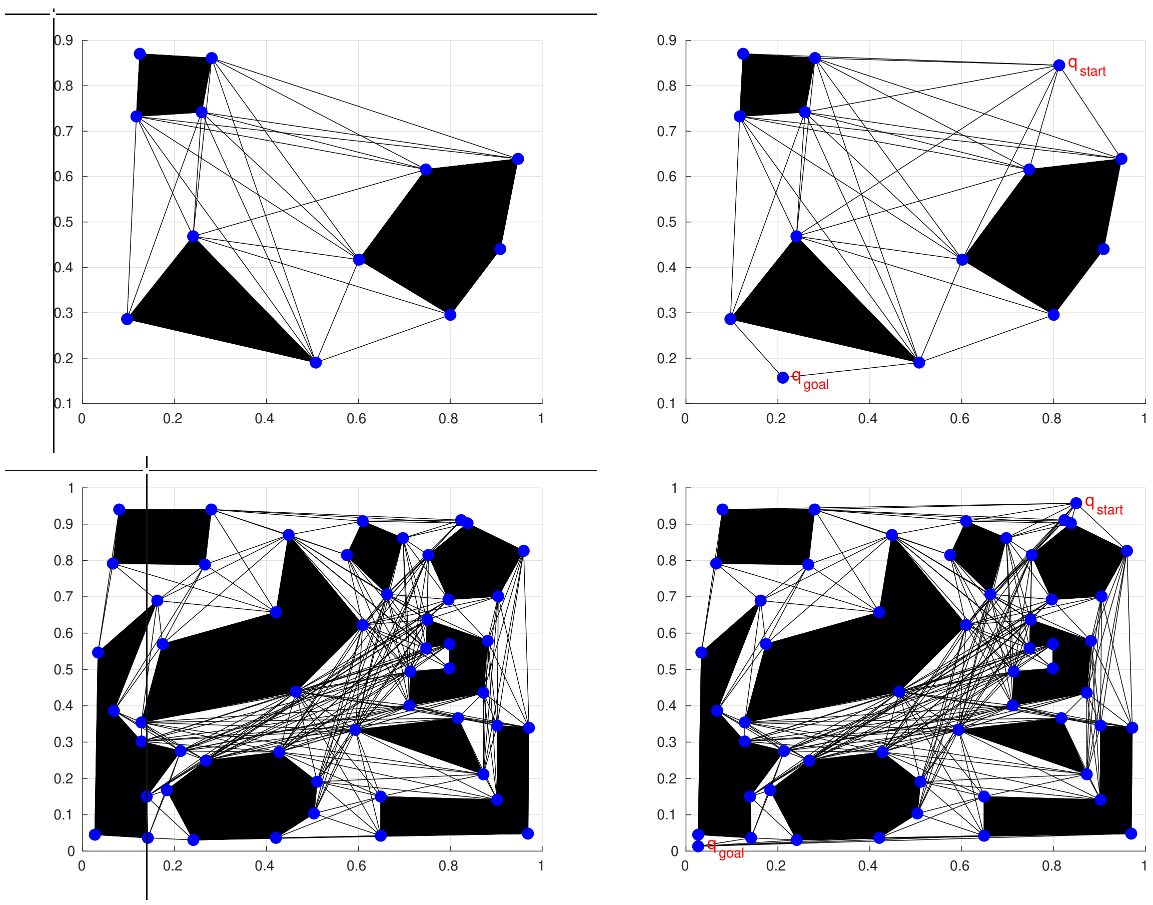

Figure 1: Visibility Graph Generation for Sparse and Dense Environments

the goal is found. The function returns a boolean indicating if a path was found, a list containing the IDs of the nodes along the path, and the total cost (in length) along the path.

4 Dijkstra

I also wrote a *Dijkstra*() function that plans a path from a start node to a goal node on the graph. This function also included a stopping criteria for when the goal is found. The function returns a boolean indicating if a path was found, a list containing the IDs of the nodes along the path, and the total cost (in length) along the path. I wrote two copies of this function; one function used an array for priority queue (PQ) and sorts the array whenever a new node is added and the second function used a min-binary heap for the PQ. I wrote a min-binary heap class following the algorithm in this Wikipedia page Wikipedia (2024).

4.1 Using a Heap-based Priority Queue (PQ)

I wrote a *BinaryHeap* class that uses the min-heap algorithm to implement a PQ where the node with the smallest cost is at the root of the tree. The nodes of the heap are objects of a *Node* class that I also wrote.

The $Node$ class has properties; *cost*, *idx*, *back_pt* and *actual_cost* where

- *cost* is the cost of the node with which the heap is organized.
- *idx* is the ID of that node in the visibility graph.
- *back_pt* is the ID of the back pointer to that node (or the parent of the node).
- *actual_cost* is the path length from the start node of the graph to that node.

The algorithms for min-heap insertion and extraction are outlined in Algorithms 1 and 2.

```
Input: Min-Heap array $PQ$, node to insert $node$
Output: Updated Min-Heap array $PQ$
$PQ[length(PQ) + 1] \leftarrow node$;
// Insert $node$ at the end of the heap
$i \leftarrow length(PQ)$;
// Index of the newly inserted element
while $i > 1$ and $PQ[\lfloor i/2 \rfloor].cost > PQ[i].cost$ do
    // $\lfloor i/2 \rfloor$ is the index of the parent
    swap $PQ[i]$ with $PQ[\lfloor i/2 \rfloor]$;
    // Swap with parent if heap property is violated
    $i \leftarrow \lfloor i/2 \rfloor$;
    // Move up to the parent node
end
return $PQ$;
```

Algorithm 1: Min-Heap Insertion

```
Input: Min-Heap array $PQ$
Output: The minimum element, updated Min-Heap
        array $PQ$
min $\leftarrow PQ[1]$;
// Store the minimum element (root) to return
$PQ[1] \leftarrow PQ[length(PQ)]$;
// Move the last element to the root
$length(PQ) \leftarrow length(PQ) - 1$;
// Decrease the size of the heap
$i \leftarrow 1$;
// Index of the root element
while $2i \leq length(PQ)$ do
    $j \leftarrow 2i$;
    // $2i$ is the index of the left child
    if $j < length(PQ)$ and
      $PQ[j].cost > PQ[j + 1].cost$ then
        $j \leftarrow j + 1$;
        // $2i + 1$ is the index of the right child
    end
    if $PQ[i].cost \leq PQ[j].cost$ then
        break;
        // Heap property is satisfied
    end
    swap $PQ[i]$ with $PQ[j]$;
    // Swap with the smaller child based on $node.cost$
    $i \leftarrow j$;
    // Move down to the child node
end
return $min$;
// Return the extracted minimum element
```

Algorithm 2: Min-Heap Extraction

Figure 2: Min-Heap Insertion and Extraction Algorithms

5 A*

I also wrote a $MyAstar()$ function that plans a path from a start node to a goal node on the graph using the A* algorithm. This function also included a stopping criteria for when the goal is found. The heuristic for this function was Euclidean distance. The function returns a boolean indicating if a path was found, a list containing the IDs of the nodes along the path, and the total cost (in length) along the path. I wrote two copies of this function; one function used an array for priority queue (PQ) and sorts the array whenever a new node is added and the second function used a min-binary heap for the PQ.

6 Comparison between Algorithms

To compare these algorithms, I used each algorithm to plan a path from a start configuration to a goal configuration on a sparse graph and a dense graph and measured the average time each algorithm took across 10000 trials and the length of the path returned. I also measured the time and path length of the A* algorithm of the PGraph class. The results for the sparse and dense environments are shown in Tables 1 and 2 and the paths with these algorithms are shown in Figures 3 and 4.

Table 1: Path Length and Computation Time in Sparse Environment

Algorithm	Path Length	Computation Time (s)
BFS	1.141918	0.000129
Dijkstra	1.012888	0.000361
Dijkstra (Min Binary Heap)	1.012888	0.001107
A* (Implemented)	1.012888	0.000254
A* (Implemented with Min Binary Heap)	1.012888	0.000525
A* (PGraph)	1.012888	0.000719

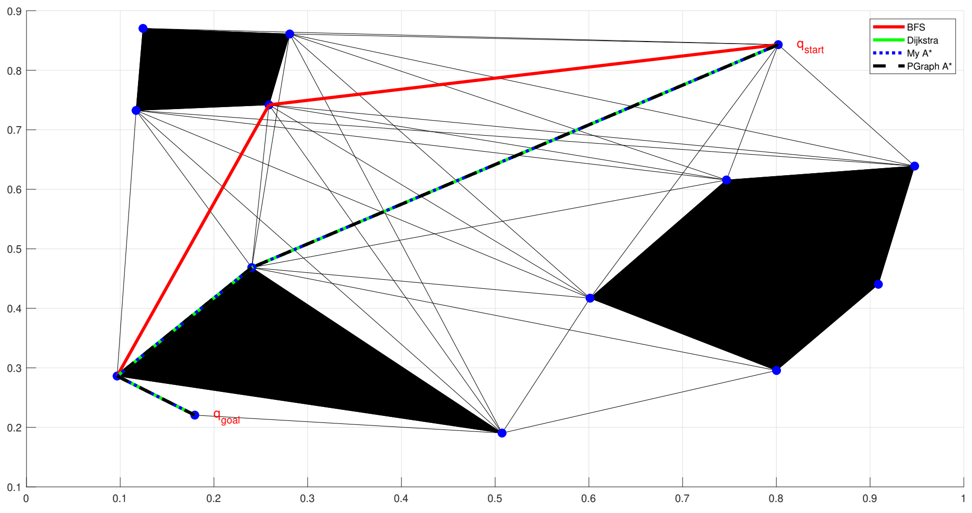

Figure 3: Paths in the visibility graph from a start node to a goal for a sparse environment for all algorithms

As shown in the results, although BFS reaches the goal in the fewest number of nodes, the path found by BFS is not optimal (it takes a longer path to the goal). Dijkstra and A* converge on the same path with the same length and since Dijkstra necessarily finds the shortest path, the heuristic used for my A* algorithm (Euclidean distance) must have been optimistic since it finds the same path. The results also show that BFS is much faster than all the other algorithms and this makes sense since there is no sorting in BFS and sorting is an expensive process. The result also shows that A* is faster than Dijkstra which is also expected since A* is an informed search directed at the goal while Dijkstra is not. Comparing the paths of my A* algorithm with that of PGraph confirms that my algorithm is accurate since it finds the same path but my algorithm appears faster than PGraph's. This may be due to my sorting method (I used the *sortrows*() function which may be faster than whatever PGraph is doing. Surprisingly, the computation time for my A* and Dijkstra's algorithm that uses a binary heap is greater. Theoretically, this approach should be faster so my implementation is probably less than optimal. Improving the code may give much better results.

Table 2: Path Length and Computation Time in Dense Environment

Algorithm	Path Length	Computation Time (s)
BFS	2.029231	0.000660
Dijkstra	1.187952	0.003310
Dijkstra (Min Binary Heap)	1.187952	0.009396
A* (Implemented)	1.187952	0.002220
A* (Implemented with Min Binary Heap)	1.187952	0.003529
A* (PGraph)	1.187952	0.002657

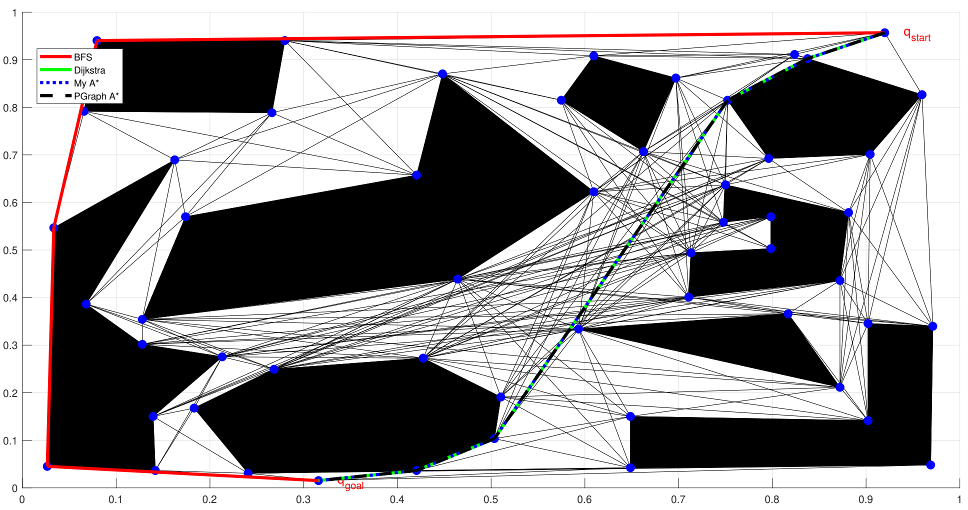

Figure 4: Paths in the visibility graph from a start node to a goal for a dense environment for all algorithms

For the dense environment, the results show similar trends as in the sparse environment but this time, the computation time for my A* implementation is close to that of PGraph's but the computation time using a binary heap is still greater. As expected, the time to compute a path is larger as the graph size increases.

References

MathWorks (2024). Function to discretize a line between two points. `https://www.mathworks.com/matlabcentral/answers/121488-function-to-discretize-a-line-between-two-points`. Accessed: 2024-09-20.

Wikipedia (2024). Binary heap — wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Binary_heap`. Accessed: 2024-09-20.