Path Planning for Point and Disk Robots

Aniruddh Kumar Sharma

April 2025

1 Introduction

This report explains an implementation of path planning algorithms for navigating through environments with polygon obstacles. The solution uses visibility graphs and Dijkstra's algorithm to find the shortest path from a start point to an end point. Two versions are presented: one for a point robot (zero radius) and another for a unit disk robot (with non-zero radius).

2 Problem Formulation

Given:

- A 2D environment with polygon obstacles
- Start point and goal point
- Robot (either a point or a disk with radius)

The task is to find the shortest collision-free path from the start to the goal.

3 Data Structures

The implementation uses the following data structures:

- `Point` = `Tuple[float, float]` - Represents a 2D point
- `Polygon` = `List[Point]` - Represents a polygon as a list of vertices
- `VisibilityGraph` = `Dict[int, Dict[int, float]]` - Adjacency list with distances
- `PathResult` = `Tuple[List[Point], float]` - The resulting path and its length

4 Visibility Graph Approach

Both implementations use the visibility graph approach:

1. Collect all vertices of obstacles along with start and goal points
2. Create a visibility graph where:
   - Nodes are the vertices of obstacles plus start and goal points
   - Edges connect vertices that can "see" each other (line of sight not blocked by any obstacle)
   - Edge weights are Euclidean distances between vertices
3. Apply Dijkstra's algorithm to find the shortest path in the visibility graph

5 Algorithm Explanation

The path planning algorithm works by exploiting the geometric property that the shortest path from start to goal in the presence of polygon obstacles will pass through obstacle vertices. This leads to the visibility graph approach:

5.1 Key Insights

- The shortest path will consist of straight-line segments connecting the start point, some subset of obstacle vertices, and the goal point.
- These straight-line segments must not intersect with any obstacle (except at the vertices).
- Only vertices that are "visible" to each other (have a direct line of sight) can be connected in the path and add weight as the euclidean distance on the path.

5.2 Main Steps

1. **Vertex Collection**: All obstacle vertices, plus start and goal points, are collected and indexed for efficient reference.
2. **Visibility Testing**: For each pair of vertices, a line-of-sight test determines whether they can "see" each other without intersecting any obstacle.
3. **Graph Construction**: A weighted graph is built where nodes are vertices and edges connect visible vertices with weights equal to Euclidean distances.
4. **Shortest Path Search**: Dijkstra's algorithm finds the minimum distance path from start to goal through the visibility graph.

5.3 Robot with Radius

For a robot with non-zero radius (modeled as a disk):

- The robot's center must stay at least one radius away from all obstacles.
- This is achieved by "inflating" each obstacle by the robot's radius.
- The path planning then proceeds as if for a point robot but using these enlarged obstacles.

The advantage of this approach is that it finds the mathematically optimal path, unlike sampling-based methods which only provide approximate solutions.

6 Core Implementation Details

6.1 Initialization

The path finder initializes by indexing all points and identifying obstacle edges:

Algorithm 1 Initialize Path Finder

```
procedure InitializePathFinder(obstacles, start, end)
    Store start and end points with indices 0 and 1
    index ← 2
    for each obstacle in obstacles do
        for each point in obstacle do
            Add point to all_points list
            Map point to index in point_to_index dictionary
            index ← index + 1
        end for
    end for
    Initialize edges_of_obstacles as empty set
    for each obstacle in obstacles do
        for each pair of adjacent points (p1, p2) in obstacle do
            Add (min(index[p1], index[p2]), max(index[p1], index[p2])) to edges_of_obstacles
        end for
    end for
end procedure
```

6.2 Line Intersection Detection

Determining whether two line segments intersect is crucial for visibility testing:

Algorithm 2 Check if Lines Cross

```
procedure DoLinesCross(point1, point2, obstacle_point1, obstacle_point2)
    Calculate orientation o1 = orientation(point1, point2, obstacle_point1)
    Calculate orientation o2 = orientation(point1, point2, obstacle_point2)
    Calculate orientation o3 = orientation(obstacle_point1, obstacle_point2, point1)
    Calculate orientation o4 = orientation(obstacle_point1, obstacle_point2, point2)
    if o1 ≠ o2 AND o3 ≠ o4 then
        return true
    end if
    if o1 = 0 AND point_is_on_segment(point1, obstacle_point1, point2) then
        return true
    end if
                                        ▷ Similar checks for other collinear cases
    return false
end procedure
```

6.3 Visibility Testing

Testing if two points have a clear line of sight:

6.4 Visibility Graph Construction

Building the graph of visible connections:

6.5 Shortest Path Finding with Dijkstra's Algorithm

Finding the shortest path using Dijkstra's algorithm:

Algorithm 3 Check Visibility Between Points

```
procedure CanPointsSeeEachOther(index1, index2)
    point1 ← all_points[index1]
    point2 ← all_points[index2]
    if (min(index1, index2), max(index1, index2)) ∈ edges_of_obstacles then
        return true
    end if
    for each obstacle in obstacles do
        for each edge (p1, p2) in obstacle do
            if point1 or point2 is vertex of this edge then
                continue
            end if
            if DoLinesCross(point1, point2, p1, p2) then
                return false
            end if
        end for
    end for
    return true
end procedure
```

Algorithm 4 Construct Visibility Graph

```
procedure MakeVisibilityGraph
    Initialize graph as empty adjacency list
    for i = 0 to number of points - 1 do
        for j = i + 1 to number of points - 1 do
            if CanPointsSeeEachOther(i, j) then
                Calculate Euclidean distance d between points i and j
                Add edge i→j with weight d to graph
                Add edge j→i with weight d to graph
            end if
        end for
    end for
    return graph
end procedure
```

Algorithm 5 Find Shortest Path

```
procedure FindShortestPath
    graph ← MakeVisibilityGraph()
    distances[start] ← 0, all others ← ∞
    pq ← priority queue with (0, start)
    came_from[v] ← null for all vertices
    while pq is not empty do
        (current_dist, current) ← pq.pop_minimum()
        if current = goal then
            break
        end if
        if current_dist ¿ distances[current] then
            continue
        end if
        for each neighbor, weight of current in graph do
            new_dist ← current_dist + weight
            if new_dist ¡ distances[neighbor] then
                distances[neighbor] ← new_dist
                came_from[neighbor] ← current
                pq.push((new_dist, neighbor))
            end if
        end for
    end while
    if distances[goal] = ∞ then
        return empty path, ∞
    end if
    Reconstruct path by backtracking from goal using came_from
    return path, distances[goal]
end procedure
```

7 Unit Disk Robot Extension

7.1 Obstacle Inflation

For robots with non-zero radius, obstacles must be ”inflated”:

Algorithm 6 Inflate Obstacle

```
procedure MakeBigger(shape, radius)
    Initialize bigger_shape as empty list
    for i = 0 to length of shape - 1 do
        p1 ← shape[i]
        p2 ← shape[(i+1) mod length of shape]
        Calculate edge vector (dx, dy) from p1 to p2
        Calculate edge length L = √(dx² + dy²)
        if L ¿ 0 then
            Calculate outward normal vector (-dy/L, dx/L)
            Move p1 by radius in normal direction
            Add moved point to bigger_shape
        else
            Add p1 to bigger_shape
        end if
    end for
    return bigger_shape
end procedure
```

7.2 Robot Path Finder Implementation

The complete algorithm for a disk robot:

Algorithm 7 Find Path for Disk Robot

```
procedure FindRobotPath(obstacles, start, end, radius)
    inflated_obstacles ← empty list
    for each obstacle in obstacles do
        inflated ← MakeBigger(obstacle, radius)
        Add inflated to inflated_obstacles
    end for
    finder ← ShortestPathFinder(inflated_obstacles, start, end)
    path, distance ← finder.FindShortestPath()
    return path, distance
end procedure
```

8 Algorithm Complexity

The time and space complexity of the implementation are as follows:

- **Time Complexity**: $O(n^3)$ where n is the total number of vertices
  - Visibility graph construction: $O(n^2 \cdot e) = O(n^3)$
  - Visibility testing: $O(e) = O(n)$ per vertex pair
  - Dijkstra's algorithm: $O(n^2 \log n)$ in worst case
- **Space Complexity**: $O(n^2)$
  - Visibility graph storage: $O(n^2)$
  - Other data structures: $O(n)$

Optimization opportunities include reduced visibility graphs, spatial partitioning, tangent graphs for disk robots, and A* search instead of Dijkstra's algorithm.

9 Example Usage

Below is a simple example demonstrating the usage:

Algorithm 8 Example Usage

```
Define obstacle polygons:
shapes = [[(1,1), (1,3), (3,3), (3,1)], [(5,2), (7,4), (9,2), (7,0)]]
start = (0,0)
end = (10,3)
path, distance = FindRobotPath(shapes, start, end, 1.0)
Output path and total distance
```