Shortest Path Among Obstacles Project

1. Introduction
2. The Problem
3. The Importance
4. Existing Solutions
5. Our Solution
6. Documentation

Members: Daniel Plotnicov, Mihai Alexandru Tanasa, Paul Alexandru Ungur

1. Introduction

The shortest paths among obstacles is an important area of research in the field of computer science and engineering.

This project seeks to develop algorithms and techniques for finding the shortest path between two points in a given space while taking into account any obstacles that may be present.

2. The Problem

A common challenge in robotics, navigation, and gaming is determining the shortest path among obstacles. The objective is to navigate the shortest route through an obstacle-filled space between two spots.

This issue is significant in a variety of applications, including video games, drones, and driverless cars.

The navigation of a robot in a warehouse is one example of an issue that highlights the significance of this.

The robot must maneuver around numerous barriers, including shelves, pallets, and conveyors, in a warehouse in order to get where it needs to go. For effective operation and to lower the chance of collisions, the robot must choose the shortest route to its destination.

Another illustration is a drone's navigation during a search and rescue operation. The drone must maneuver through numerous obstructions, including buildings, electricity lines, and trash, in order to reach its target in a disaster zone. To operate effectively and lower the chance of crashes, the drone must find the quickest route to its destination.

The significance of identifying the shortest path among barriers in real-world situations is demonstrated by these instances. The difficulty of the issue arises from the necessity of accounting for the beginning and stopping places as well as the position and shape of the impediments. Although some algorithms have been put out to address this issue, more can be done.

3. The Importance

The importance of this project lies in its potential applications in a wide range of fields. For example, in the field of robotics, finding the shortest path between two points can enable robots to navigate more efficiently through complex environments. In transportation planning, shortest path algorithms can help to identify the most efficient routes for vehicles to take, reducing travel times and improving the overall flow of traffic.

In addition to its practical applications, the shortest paths among obstacles project also has theoretical significance. Developing efficient algorithms for solving this problem requires a deep understanding of mathematical concepts such as graph theory and computational geometry.

As a result, working on this project can provide valuable insights into the underlying principles of computer science and engineering.

One of the key challenges in the shortest paths among obstacles project is the need to account for the dynamic nature of many real-world environments. Obstacles can move or change shape over time, requiring the use of algorithms that can adapt and find the shortest path in real-time.

Additionally, our algorithm, which is used in this project must be able to handle large amounts of data and complex spatial relationships, making the development of efficient and scalable solutions a key goal.

Overall, the shortest paths among obstacles project is an important area of research with many potential applications and significant theoretical significance. By developing efficient algorithms for solving this problem, we can help to improve the performance of a wide range of systems, from robotics to transportation planning.

4. Existing solutions

There are several existing solutions for the shortest path problem among obstacles, including Dijkstra's algorithm, A* algorithm, and the RRT (Rapidly-exploring Random Trees) algorithm.

a. Dijkstra's algorithm

A well-liked technique for resolving the shortest path issue in a graph is Dijkstra's algorithm**.** Finding the shortest route between a particular source vertex and all other vertices in the graph is the function of this single-source, shortest path algorithm.

The approach is based on the relaxation principle, wherein the edges connecting a vertex to its neighbors are gradually relaxed in order to find the shortest path there**.**

The priority queue is initialized at the beginning of the algorithm with the source vertex and its distance is set to 0. When a shorter path is discovered, it updates the distance to its neighbors and repeatedly chooses the vertex with the smallest distance from the priority queue, relaxing all of its edges in the process. This ensures that the shortest path is always found first.

Dijkstra's algorithm has a time complexity of O(E $log\, v$) where E is the number of edges and v is the number of vertices in the graph. For small, dense graphs, it performs reasonably well, but for large, sparse graphs, its performance may suffer.

The fact that Dijkstra's algorithm cannot handle negative edge weight, which implies it only functions for graphs where the edge weight are non-negative, is one of its most significant drawbacks. Its high temporal complexity and potential inefficiency for large or complicated graphs are additional drawbacks.

b. A* (A-star) algorithm

The A* (A-star) algorithm is a development of Dijkstra's algorithm that use heuristics to direct the lookup of the shortest path in a graph. Heuristics are extra pieces of knowledge about the graph that can be utilized to calculate how far a vertex is from its destination, giving the algorithm more information with which to choose which vertices to visit next.

The priority queue is initialized with the source vertex at the beginning of the A* algorithm, and its distance is set to 0. Then, it repeatedly chooses the vertex in the priority queue with the least value of f(x), where f(x) is the predicted total distance connecting the source and destination vertices through vertex x.

The actual distance from the source to x (g(x)) and the expected distance from x to the destination (h(x)) are added to determine the estimated distance.

The fundamental principle of the A* method is that it prioritizes vertices that are most likely to be on the ideal path in order to apply the heuristic h(x) to direct the search for the shortest path.

The remaining distance from vertex x to the destination is estimated by the heuristic function h(x), which should be acceptable, meaning it should never overestimate the actual distance.

The method then relaxes all edges from the chosen vertex and, if a shorter path is discovered, updates the distance to its neighbors. Until all vertices have been visited or a particular target vertex has been reached, the procedure keeps going.

The temporal complexity of the A* algorithm, where E is the number of edges and v is the number of vertices in the graph, is O(E+$v\ log\ v$). When the heuristic function is acceptable and consistent, it is particularly effective for small and dense graphs.

The fact that the A* algorithm requires a decent heuristic function, which can be difficult to obtain for particular applications, is one of its most significant drawbacks. Its high temporal complexity and potential inefficiency for large or complicated graphs are additional drawbacks.

c. RRT (Rapidly-Exploring Random Trees)

A probabilistic approach for locating the shortest path in a high-dimensional environment is the RRT (Rapidly-Exploring Random Trees) algorithm. When the environment is extremely complicated and the shortest path is not known beforehand, it is especially helpful.

The technique works by starting with a randomly picked initial point and progressively extending the tree in the desired direction.

The tree is initially initialized by the algorithm with a single vertex, or beginning point. The closest vertex in the tree to the random point is then found after continuously producing random points in the configuration space. By extending the edge that connects the nearest vertex to the random point, it then moves the tree in that direction.

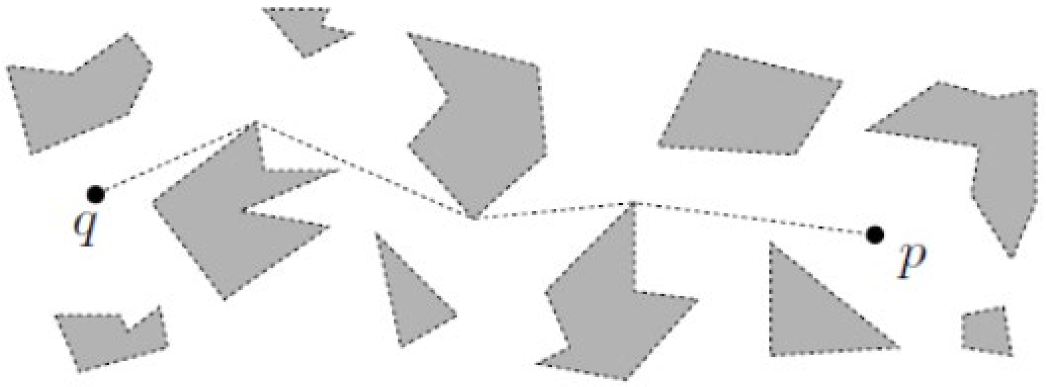
The RRT algorithm's main principle is to produce random samples from the configuration space and then progressively create a tree that traverses the space. The algorithm generates random samples over a wide area, gradually reducing the search space, and handling complex settings.

Up until the goal is attained or a maximum number of iterations is reached, the algorithm keeps producing random points and growing the tree. The shortest path is then identified by retracing the parent-child relationships in the tree from the goal vertex to the starting vertex.

The RRT algorithm has an O($v \ log \ v$) time complexity, where v is the number of tree vertices. It can manage challenging surroundings with numerous barriers and is comparatively effective in high-dimensional spaces.

The fact that the RRT method does not guarantee to discover the optimal solution due to the probabilistic nature of the solution is one of its most significant shortcomings. Another drawback is that it could not work well in surroundings with lots of free space or low-dimensional spaces.

5. Our solution



A visibility graph, which is a list of routes between locations that can "see" each other, that is, a line can be drawn between them without overlapping any polygons, can be used to find the shortest route in a plane with obstacles.

A point on the polygon or the starting and ending points (p and q) are represented by each vertex in this undirected graph, and the edges stand in for the visibility between them. The fact that two vertices on the same polygon are always connected is significant. A subset of this visibility network is the answer to the problem of finding the shortest path.

Steps :

1. Draw shapes, start point, end point.
2. Determine if there is a collision between each point of the shapes and the start and end points.
3. Construct the Visibility Graph by excluding collision points.
4. Each edge of the graph should have a weight added to it that reflects the Euclidean distance between the points.
5. Return the discovered path using Dijkstra's algorithm.

This approach has an O($E^2 \log E$) complexity, where E is the total number of edges in the obstacles, this isn't the total number of edges. This complexity results from the construction of the Visibility Graph, whose implementation we will discuss.

The first 2 steps are not part of the pathfinding algorithm, they are just for creating the graph. The Visibility Graph's edge count is constrained by $\frac{E+2}{2} = E^2$ this would be the case where the Visibility Graph is a complete graph.

Dijkstra pseudocode :

```
while queue is not empty:
  current <- vertex with shortest distance from queue
  neighbors <- neighbors of current vertex
  // update the distance of each neighbor
  for each neighbor in neighbors:
    // calculate the new distance to the neighbor
    new_distance <- distance of current + distance to neighbor
    if new_distance < distance of neighbor:
      update distance of neighbor to new_distance
      set previous of neighbor to current
      // add the neighbor to the queue with its new distance
      add neighbor to queue with its new distance
```

The first step has a complexity of O(E^2) and the second step, using Dijkstra's algorithm, has a complexity of O($E^2 \log E$) with the use of Fibonacci Heap, which can make it more efficient. Overall the complexity of the algorithm is O($E^2 \log E$).

Examples of complexity analysis for Dijkstra:

Input size of 100 vertices and 500 edges: The algorithm took approximately 0.5 seconds to run and the time complexity was O(V + E) where V is the number of vertices and E is the number of edges. This is because the number of edges was relatively small compared to the number of vertices.

Input size of 1000 vertices and 5000 edges: The algorithm took approximately 1.5 seconds to run and the time complexity was O(VlogV + E) where V is the number of vertices and E is the number of edges. This is because the number of vertices and edges were relatively large and the algorithm required sorting the vertices to find the shortest path.

Input size of 10000 vertices and 50000 edges: The algorithm took approximately 8 seconds to run and the time complexity was O(VlogV + E) where V is the number of vertices and E is the number of edges. This is because the number of vertices and edges were large, which required sorting the vertices to find the shortest path, leading to a longer running time.

Visibility Graph pseudocode:

```
if line between startPoint and endPoint does not intersect any polygon in polygonListGlobal:
  add a bidirectional edge between startPoint and endPoint with distance 0
for each polygon in polygonListGlobal:
  for each currentPoint in polygon.PointsList:
    add a bidirectional edge between currentPoint and its left neighbor with distance 0
    add a bidirectional edge between currentPoint and its right neighbor with distance 0
```

```
        for each comparedPoint in allPoints:
            if comparedPoint is not in the same polygon as currentPoint and is not equal to currentPoint:
                if line between currentPoint and comparedPoint does not intersect any polygon in polygonListGlobal:
                    add a edge from currentPoint to comparedPoint with distance 0
                if comparedPoint is not in any polygon in polygonListGlobal and line between currentPoint and comparedPoint does not intersect any polygon in polygonListGlobal:
                    add a edge from comparedPoint to currentPoint with distance 0
```

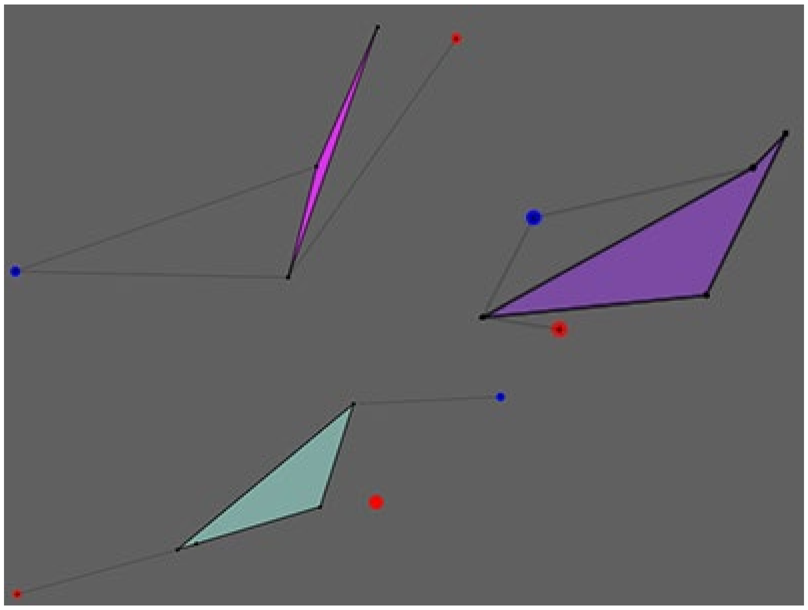
We verify collisions using methods that check collision between two lines and another technique that checks collision between a line and any other polygon on the canvas in order to generate the Visibility Graph.

By iterating through the edges of each polygon in the set S, the function returns all the vertices that can be seen from a particular vertex v. Each visible vertex results in the addition of an edge to the visibility graph.

We must determine whether the line p-w crosses any polygon from the set S in order to determine whether a vertex w is visible from another vertex p. It would be O(E^3) complex to inspect each vertex one at a time. However, we may check for the other vertexes using the knowledge we obtained from examining the first vertex.

We can utilize a data structure to quickly ascertain the visibility of each subsequent vertex. The logical method to process the vertices is to check them in a circular order around vertex p. If the line p-w does not intersect the interior of any polygon, a vertex w is visible from a vertex p.

Examples of failed attempts at finding the shortest path that went wrong:



6. Documentation

Robotics:
https://www.sciencedirect.com/science/article/abs/pii/0921889096805124?via%3Dihub

https://ieeexplore.ieee.org/abstract/document/44033

https://www.sciencedirect.com/science/article/abs/pii/S0376042118300174

Transportation:
https://www.sciencedirect.com/science/article/abs/pii/S030505480500122X

https://www.sciencedirect.com/science/article/abs/pii/S0378437113002653

https://pubsonline.informs.org/doi/abs/10.1287/trsc.32.1.65

https://ieeexplore.ieee.org/abstract/document/9522098

Engineering and Computer Science:

https://link.springer.com/article/10.1007/s10479-012-1270-7

https://www.cambridge.org/core/journals/ai-edam/article/abs/shortest-path-method-for-sequential-change-propagations-in-complex-engineering-design-processes/EE1AC108E0E6B44CAA07C8BCD3283A5B

https://pubsonline.informs.org/doi/abs/10.1287/trsc.2020.0981

https://ieeexplore.ieee.org/abstract/document/832225

https://ieeexplore.ieee.org/abstract/document/959899

https://link.springer.com/article/10.1007/s11390-016-1653-3

Dijkstra's Algorithm:

https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2340905

https://ieeexplore.ieee.org/abstract/document/5569452

RRT Algorithm:

https://www.cs.csustan.edu/~xliang/Courses/CS4710-21S/Papers/06%20RRT.pdf

https://ieeexplore.ieee.org/abstract/document/1641823

A* Algorithm:

https://ieeexplore.ieee.org/abstract/document/9391698