Path Planning

By Abdelrahman Mostafa

1 main information

1.1 What is Path Planning?

Path planning is finding the best way for a robot or a vehicle to go from one place to another without hitting anything. Path planning is important for robots and vehicles that can work by themselves. Path planning can be used for different things, like self-driving cars, robot arms, or drones.

1.2 How to do Path Planning?

here are many ways to do path planning, depending on how the space, the robot or vehicle, the task, and the situation are described. Some of the common ways are:

- Grid-based search:
  This way divides the space into squares, and gives each square a score based on how far it is from the goal and if it has anything in it. Then, it uses search methods like A*, D*, or Dijkstra's (for more info go to section 2) to

find the path with the lowest score from the start square to the goal square. Grid-based search is easy and always works, but it may not be fast or exact for spaces that are not squares.

— Take into consideration

1. *in a grid-based search problem, an obstacle can be a cell with a very high cost, $f(n)$, or an infinite cost which means that it is impossible or very hard to move through it*
2. *By spaces that are not squares, I mean spaces that have different shapes, sizes, or dimensions than a grid of cells. For example, a space that is curved, irregular, or continuous is not a square space. A space that has many variables or dimensions, such as the configuration space of a robot arm, is also not a square space. These spaces are harder to represent and search than square spaces, because they have more complexity and diversity.*

- Sampling-based search:
  This way picks random points or shapes in the space, and connects them with lines that

do not cross anything to make a graph or a tree. Then, it uses graph search methods like breadth-first, depth-first, or best-first to find a path from the start point to the goal point. Sampling-based search is good for spaces that are not squares, but it may not always work or be the best.

- Trajectory optimization:
  This way makes the path planning problem into a math problem that tries to make the path as short, smooth, clear, and other things as possible.

2 further description

2.1 A* algorithm

2.1.1 what is A* algorithm?

This is a search algorithm that finds the shortest path between two nodes in a graph, such as a map or a maze. It uses a heuristic function (which is a way of estimating how far a node is from the goal) to guide its search. It picks the node with the lowest cost (which is the sum of the distance from the start and the heuristic value) and explores its neighbors. It repeats this process until it reaches the goal or runs out of nodes to explore. A* algorithm is smart, fast, and optimal, meaning it always finds the best path if it exists.

2.1.2 examples

To illustrate, let's say we want to find the shortest path from Cairo to Alexandria in Egypt using A* algorithm. We can use the straight-line distance as our heuristic function, which is easy to compute using the coordinates of each city. The heuristic value for Cairo would be about 180 km, which is the distance to Alexandria as the crow flies. The heuristic value for Alexandria would be zero, since it is the goal. The heuristic value for any other city would be somewhere between these two values, depending on how close it is to Alexandria. The A* algorithm would use these heuristic values to decide which city to explore next, until it reaches Alexandria or runs out of cities to explore.

We get the total cost according to the formula :

$$f(n) = g(n) + h(n)$$

as :

- $g(n)$ is the cost of the path from the start node to n
- $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal
- $f(n)$ is the total cost

2.2 D* algorithm

This is an extension of A* algorithm that can handle changes in the graph while searching. For example, if a robot is moving in an unknown environment and discovers new obstacles, it can use D* algorithm to update its path without starting from scratch. D* algorithm keeps track of the changes in the graph and repairs the path accordingly. There are different versions of D* algorithm, such as Focused D* and D* Lite, that have different ways of updating the path.

2.3 Dijkstra algorithm

2.3.1 what is Dijkstra algorithm?

This is another search algorithm that finds the shortest path between one node and all other nodes in a graph. It does not use a heuristic function, but only considers the distance from the start node. It also picks the node with the lowest cost and explores its neighbors, but it does not stop until it has visited all nodes in the graph. Dijkstra algorithm is simple, complete, and optimal, but it may be slower than A* algorithm if the goal node is far away as it will check all nodes whatever its heuristic value was.

2.3.2 examples

1. Suppose we have a graph with four nodes, A, B, C, and D, and some edges with different weights, which are the distances between the nodes. We want to find the shortest path from A to all other nodes.
2. We start by assigning a cost to each node, which is the distance from A. The cost of A is zero, since it is the start node. The cost of any other node is infinity, since we do not know how to reach them yet.
3. We also keep track of a set of visited nodes, which are the nodes that we have explored and found their shortest paths. The set is initially empty.
4. We pick the node with the lowest cost that is not in the visited set. This node is called the current node. The first current node is A.
5. We explore the neighbors of the current node and update their costs based on the current node's cost and the edge weight. For example, the cost of B is updated from infinity to 2, which is the sum of A's cost (0) and the edge weight (2). The cost of C is updated from infinity to 4, which is the sum of A's cost (0) and the edge weight (4).
6. We add the current node to the visited set and mark it as done.

7. We repeat this process until all nodes are in the visited set or there are no more nodes with finite costs. The next current node is B, since it has the lowest cost (2) among the unvisited nodes.
8. We explore B's neighbor and update its cost. The cost of D is updated from infinity to 5, which is the sum of B's cost (2) and the edge weight (3).
9. We add B to the visited set and mark it as done.
10. We repeat this process until all nodes are in the visited set or there are no more nodes with finite costs. The next current node is C, since it has the lowest cost (4) among the unvisited nodes.
11. We do not explore C's neighbor, since it is already in the visited set.
12. We add C to the visited set and mark it as done.
13. We repeat this process until all nodes are in the visited set or there are no more nodes with finite costs. The next current node is D, since it has the lowest cost (5) among the unvisited nodes.
14. We do not explore D's neighbor, since it is already in the visited set.

15. We add D to the visited set and mark it as done.
16. We stop because all nodes are in the visited set.

The final costs of each node are:
A: 0 B: 2 C: 4 D: 5

2.4 summary

- A* is an algorithm to find path with the least cost (least obstacles and distance)
- D* is like A* but can handle changes in the space while searching and update itself accordingly.
- Dijkstra is visit all nodes not literally but computationally