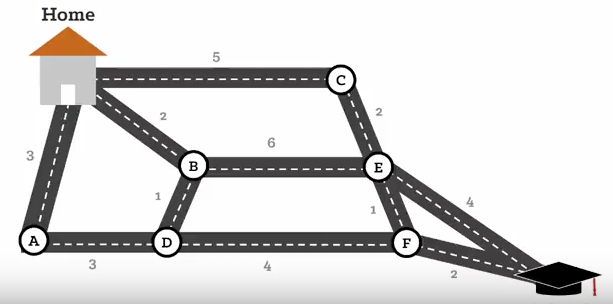
Research – Mobile Robotics and Mapping Algorithms

For most robots, simple movement isn’t too complicated. If I wanted a robot to move forward and then come back to me, I could theoretically split it up into the following instructions:

1. Forward for 5 seconds
2. Rotate 180 degrees
3. Forward for 5 seconds
4. Rotate 180 degrees

However, getting a robot to follow a complex path between two points is a lot more difficult. Theoretically, somebody could walk the path between point A and point B and pass on each individual instruction to a robot, like we did in the example above. Not only is this time consuming and expensive to do, but it’s also impossible in some situations. What would we do if every time we selected an address in Google Maps we had to wait days for someone to come down and map out the path to us? Fortunately, that’s where mapping algorithms come into play.

Mapping algorithms solve the question “What’s the shortest distance from point A to point B?” As you might imagine, the algorithms needed to answer this question are complex and can be difficult to understand. I didn’t have any programming experience or familiarity with mapping algorithms before this assignment but learning about how they worked was extremely interesting. The best and most widely applied algorithm that programmers use to solve this question is called “Dijkstra's shortest path algorithm”.



Dijkstra’s algorithm is called a “greedy” algorithm, which in this context means that at every logical step the algorithm will choose the shortest path available. To understand Dijkstra’s shortest path algorithm, imagine paths as graphs with nodes and edges (also known as paths). Above you can see the visualization that helped me understand this concept the best. The shortest path from home to school would be from Home -> B -> D -> F -> School, and this can be calculated by using Dijkstra’s algorithm. Above, the letters are the nodes and the numbers are the edges. These edges are known as “weighted edges”, which mean that they also have their own numeric value.

So how does this work? To understand what’s going on in the code, we need to understand the variables being used:

v: nodes  
u: nodes that are neighbors to node v  
s: source node (the starting place)  
dist: an array of distances; in this context, this means that the array would store a list of distances  
Q: a queue of nodes; in this context, a queue is essentially a list with special properties  
S: a set of nodes; when Dijkstra’s algorithm finishes, this set will contain the nodes that a robot would travel to in order to walk the minimum distance between point A and B.

These are the only variables that need to be used to find the shortest distance between A and B. However, some special initializations need to be made in the beginning of the program for this algorithm to work. An initialization is essentially setting the starting state of each variable. The first set of initializations that must be executed are for the **distance**s that are being stored in our dist array. However, we don’t know any distances between nodes at this point. The only distance we know is from the source node *s* to itself, which is 0. So this distance for *s* is stored in the *dist* array. We still need to set the distance for the remaining nodes however, so those are set to ∞ since we don’t know what the distance between the nodes are yet.

On initialization, *Q* will be filled with every node that can be travelled to. As each node is visited, it’ll be removed in *Q* so that by the end of the algorithm *Q* should be empty. *S* is initially empty.

Now that all our variables have been initialized, we can begin the algorithm. In the first step, a node *v* is chosen but it’ll only be chosen if *Q* is not empty and if this node *v* is not already in *S.* The node *v* that will be selected is the one with the smallest weighted edge. What that means in our context is that the v that is selected will have the shortest distance between the starting point and any of its neighboring nodes. However, since this is the first time “running” the algorithm, the starting node will be chosen because we set it equal to 0 and everything else equal to ∞. This node will be added to set *S* since this set is for nodes that have been visited. The algorithm continues to do this until eventually every node has been checked and at the end, we’ll have a list in set *S* that shows us the optimal path to take.

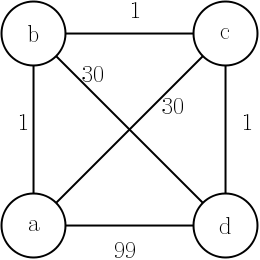
How does this relate to robotics? This is actually the exact algorithm that Google Maps and other navigation software uses to plot a path from Point A to Point B! AIs on self-driving cars of the future would use this exact algorithm to get from place to place.

Finding the shortest path can make a huge financial difference for a robot. Time spent away from the optimal path is time and money wasted, so calculating the shortest path is extremely useful for everyone involved. But can this algorithm only be applied to robotic movement? Is there another way we can use this to cut costs and make the most efficient robots?

It turns out that yes, there are other applications as well. Any information that the robot sends out follows these same principles to an extent. If the robot is complex enough, it could even have its own router. When information is ready to be sent out from one router to be accepted by another, Dijkstra’s algorithm can be used to determine which router the information should be sent to. It checks each of the “neighboring” routers that it can connect to and it sends data out to the one closest to it. That router will then select its closest neighbor and so on until you have the shortest possible path between our router A and the destination B.

Does Dijkstra’s algorithm always work? It does in some cases. When there are only positive edges and each node doesn’t need to be visited, Dijkstra’s shortest path algorithm will always find the shortest path from point A to point B. However, there’s a famous question in computer science called “The Travelling Salesman” problem. Google defines the traveling salesman problem as “a mathematical problem in which one tries to find the shortest route that passes through each of a set of points once and only once”. This question was first proposed with the following example: if a salesman needs to go to a certain list of cities, how can he calculate the most efficient path to take so that the salesman returns to his original position? This situation could also be applied to robots. For instance, let’s say Amazon releases a large drone that can deliver packages across multiple cities. How does the drone calculate the order that the cities will be visited in?

Unfortunately, Dijkstra’s algorithm does not work in this situation. It only gives us the shortest path. Even if it’s adjusted to take the shortest path while also visiting every node, it won’t come up with the optimal path because it’s a “greedy algorithm”. As defined early, a greedy algorithm only takes the best logical step at that point in time, not what’s the most logical overall. Below is an example of why Dijkstra’s algorithm couldn’t be applied.



In this example, we need to go to each node while keeping our distance travelled to a minimum. Our human brains can deduce that if we must start at a then the optimal path would be a -> b -> d -> c -> a, which has an edge weigh total of 62. However, if we modify Dijkstra’s algorithm to also visit every node, it’ll take a path that’s longer than the optimal one and here’s why: In the first iteration, it’ll take the path from a to b (total distance: 1). But since it’s always looking for the shortest possible distance to take, it’ll go from b -> c next (total distance: 2), then c -> d (total distance: 3), then finally d -> a next (total distance: 102). This is of course greater than the optimal distance of 62, even though we executed Dijkstra’s algorithm perfectly.

In conclusion, Dijkstra’s algorithm gives us a powerful way to map out the shortest possible path between point A and point B. Its versatility and simplicity mean it’s the go-to mapping algorithm for all GPS systems like Google Maps today. However, there are some famous situations it can’t be used, such as in the traveling salesman problem. Despite these shortcomings it’s the single best algorithm to represent what a mapping algorithm should do in the world of robotics and it will continue to pave the way that future inventions will operate.