A Walkthrough of Dijkstra's Algorithm (in JavaScript!)

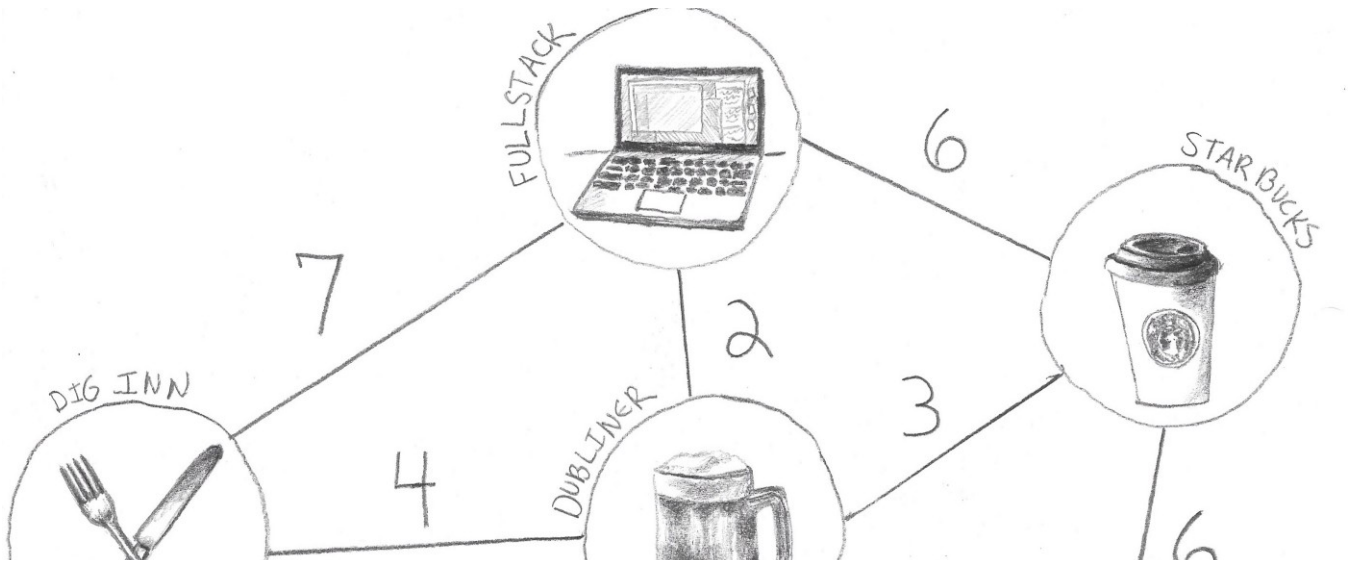
Adrienne Johnson
Nov 20, 2018 · 7 min read

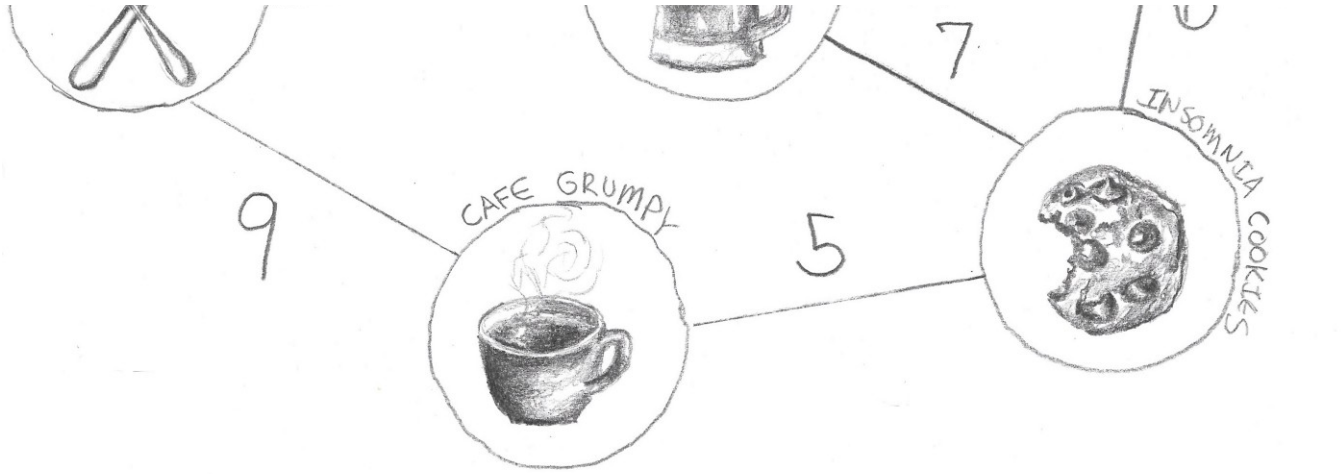
So many things I use every day used to seem like magic, served up for my convenience and delight by programming gods on Mt. Google, Mt. Spotify, and other peaks. Now, as a software engineering student, I feel more and more empowered to understand how things *actually* work.

Take, for example, the path-finding prowess of Google Maps, software that has gotten me pretty much every place I've been in the past decade. To begin understanding this one, I'll walk through a basic implementation of Dijkstra's Algorithm, a solution that helped Google Maps get where it is today.

The Map

Before the algorithm can calculate anything, it needs some possible paths to traverse, so first we need a map. Let's say I'm at Fullstack Academy in New York, and I want to know the shortest possible path to Cafe Grumpy. The junctures between here and there might look something like the diagram below, with the time it takes to walk from one point to another marked along each path. For easier reference, I'm using landmarks instead of street intersections.





Map of nearby streets that might lead to Cafe Grumpy (not to scale...).

On my phone, this constellation would look like a map, but under the hood, it is a data structure called a **weighted graph**. The locations on the map are the **nodes** of the graph, the paths between them are the **edges**, and the time to get from any one node to the other is the **weight** of each edge.

The graph can be represented by an **adjacency list**. Each node in the adjacency list points to an array of neighboring nodes, or in other words, the endpoint of every edge extending from that node. In a weighted graph, the adjacency list carries a second piece of information: the weight of each edge, or the cost of getting to that particular node.

In Javascript, we could build a weighted graph like this:

```
class Graph {
  constructor() {
    this.nodes = [];
    this.adjacencyList = {};
  }
```

To add a node to the graph (a new location on our map), we push it into the collection of node values, which will help us iterate through them later, and we add a new entry in the adjacency list, setting its value to an empty array.

```
  addNode(node) {
    this.nodes.push(node);
    this.adjacencyList[node] = [];
  }
```

When we add a neighbor, or edge, to that node, we want to add that edge to our adjacency list, so we push the neighboring node along with the time it takes to get there into that array.

```
  addEdge(node1, node2, weight) {
    this.adjacencyList[node1].push({node:node2, weight: weight});
    this.adjacencyList[node2].push({node:node1, weight: weight});
  }
}
```

Now the structure is in place to create our map of nearby coffee shops!

```
let map = new Graph();
map.addNode("Fullstack");
map.addNode("Starbucks");
...

map.addEdge("Fullstack", "Starbucks", 6);
map.addEdge("Fullstack", "Dig Inn", 7);
...
```

The Approach

In this simple example, it would be easy enough to scan the diagram and add some numbers to figure it out, but if I wanted to venture out of my neighborhood to a coffeeshop in, say, Midtown, the permutations of paths there would be much harder to calculate myself.

That's where Dijkstra's Algorithm comes in. Before I get into any code, let's get a basic idea of how it'll work:

1. Move to a node that we haven't visited, choosing the fastest node to get to first.
2. At that node, check how long it will take us to get to each of its neighboring nodes. Add the neighbor's weight to the time it took to get to the node we're currently on. Keep in mind that we're calculating the time to reach those nodes *before* we visit them.
3. Check whether that calculated time is faster than the previously known shortest time to get to that node. If it *is* faster, update our records to reflect the new shortest

time. We'll also add this node to our **line of nodes** to visit next. That line will be **arranged in order of shortest calculated time** to reach.

By calculating and continually updating the shortest time to reach each node on the graph, the algorithm compiles the shortest path to the endpoint.

Let's try those steps out. Of all the nodes on the map, we only know the time it takes to get to one — Fullstack — because it takes no time at all. It's where we're starting! From there, we check the neighboring nodes. We see that Dubliner has a weight of 2, and so we add that to the time it took to get where we are, which is 0. We do the same for Starbucks and Dig Inn. Since these times are shorter than any time we previously knew, we record the time it takes to get to these nodes as 2, 6, and 7, respectively, and we add them to our line of nodes to visit next. We also update our record of the last node we were on when we got to those now-shortest paths, adding an entry for each one and pointing them toward Fullstack. This is how we'll trace our path back to where we started.

We move to another node, the one first up in our line. Right now that's the Dubliner. Now, here's where the algorithm starts to reveal its glory. If we check the neighboring nodes, we see that the time it takes to get to Starbucks is 5. We got that by adding the time it took us to to the Dubliner (2) plus the weight of the edge between the Dubliner and Starbucks (3). Now let's check our records. The shortest distance we have on file for Starbucks is 6. Wait, that's longer! We just found a faster route to Starbucks! Now we can update two sets of records: First, we update the shortest known time to Starbucks. Then we update our path trace, changing our entry for Starbucks to point to Dubliner — the node we were on when we found the shortest time to reach it.

The Line

A key part of this approach is maintaining an ordered line of nodes to visit next. To set up that part of the algorithm, we'll create a **priority queue**. A regular queue is first-in, first-out. A priority queue, however, doesn't just place elements at the end of the line. It places them in order based on the **weight** each element has. Think of a regular queue as a grocery store line and a priority queue as the intake of an emergency room.

```
class PriorityQueue {
  constructor() {
    this.collection = [];
  }
```

We'll add a method for adding an element to the line. The element we pass into it will be an array, with the node's value at index 0 and its weight at index 1. Each time we add a node, we'll iterate through our line and insert the node before the first element we find that has a larger weight than our node. In the case of our pathfinding, **lower** weight = **shorter** time = **higher** priority.

```
    enqueue(element){
      if (this.isEmpty()){
        this.collection.push(element);
      } else {
        let added = false;
        for (let i = 1; i <= this.collection.length; i++){
          if (element[1] < this.collection[i-1][1]){
            this.collection.splice(i-1, 0, element);
            added = true;
            break;
          }
        }
        if (!added){
            this.collection.push(element);
        }
      }
    };
```

We'll also add quick methods to remove an element and check if the queue is empty:

```
    dequeue() {
      let value = this.collection.shift();
      return value;
    };

    isEmpty() {
      return (this.collection.length === 0)
    };
  }
```

And now we can put the algorithm together — **finally!** — as a method on our graph.

The Algorithm

We'll start by creating objects to hold our record of the shortest known times and the steps we took. And we'll initialize our priority queue:

```
findPathWithDijkstra(startNode, endNode) {
  let times = {};
  let backtrace = {};
  let pq = new PriorityQueue();
```

At the start, all we know is the shortest time it takes to get to our start node. The shortest time it takes to reach the others could be anything, so we'll initialize those times to infinity.

```
  times[startNode] = 0;

  this.nodes.forEach(node => {
    if (node !== startNode) {
      times[node] = Infinity
    }
  });
```

We add our starting node to the priority queue to get things kicked off.

```
  pq.enqueue([startNode, 0]);
```

And now we can really get rolling! We access the first element in the queue and start checking its neighbors, which we find using the adjacency list we made at the very beginning. We add the neighbor's weight to the time it took to get to the node we're on.

```
  while (!pq.isEmpty()) {
    let shortestStep = pq.dequeue();
    let currentNode = shortestStep[0];

    this.adjacencyList[currentNode].forEach(neighbor => {
      let time = times[currentNode] + neighbor.weight;
```

Then we check if the calculated time is less than the time we currently have on file for this neighbor. If it is, then we update our times, we add this step to our backtrace, and we add the neighbor to our priority queue!

```
      if (time < times[neighbor.node]) {
        times[neighbor.node] = time;
```

```
        backtrace[neighbor.node] = currentNode;
        pq.enqueue([neighbor.node, time]);
      }
    });
  }
```

Now we can leave the algorithm to do its work. Once it reaches the end of our queue, all we have to do is look through our backtrace to find the steps that will lead us to Cafe Grumpy. We can look up Cafe Grumpy in our times object to find out just how long it will take to get there, knowing that it's the quickest route.

```
  let path = [endNode];
  let lastStep = endNode;

  while(lastStep !== startNode) {
    path.unshift(backtrace[lastStep])
    lastStep = backtrace[lastStep]
  }

  return `Path is ${path} and time is ${times[endNode]}`
}
```

There you have it! The shortest path is to take Dubliner to Insomnia Cookies to Cafe Grumpy, in 14 minutes.

Smart and elegant, right? There are of course ways to optimize this approach. Imagine there were paths extending from Fullstack toward the top of our map, in the *opposite* direction of Cafe Grumpy. It's less likely that checking any of those nodes would uncover a quicker route, but Dijkstra's Algorithm would look anyway, and that could be a waste of time. Though depending on traffic…

But optimizing and factoring in those considerations is a topic for another day. For now, I think I'll grab some coffee and maybe pick up a cookie on my way.

Programming JavaScript Algorithms Pathfinder Data Structures