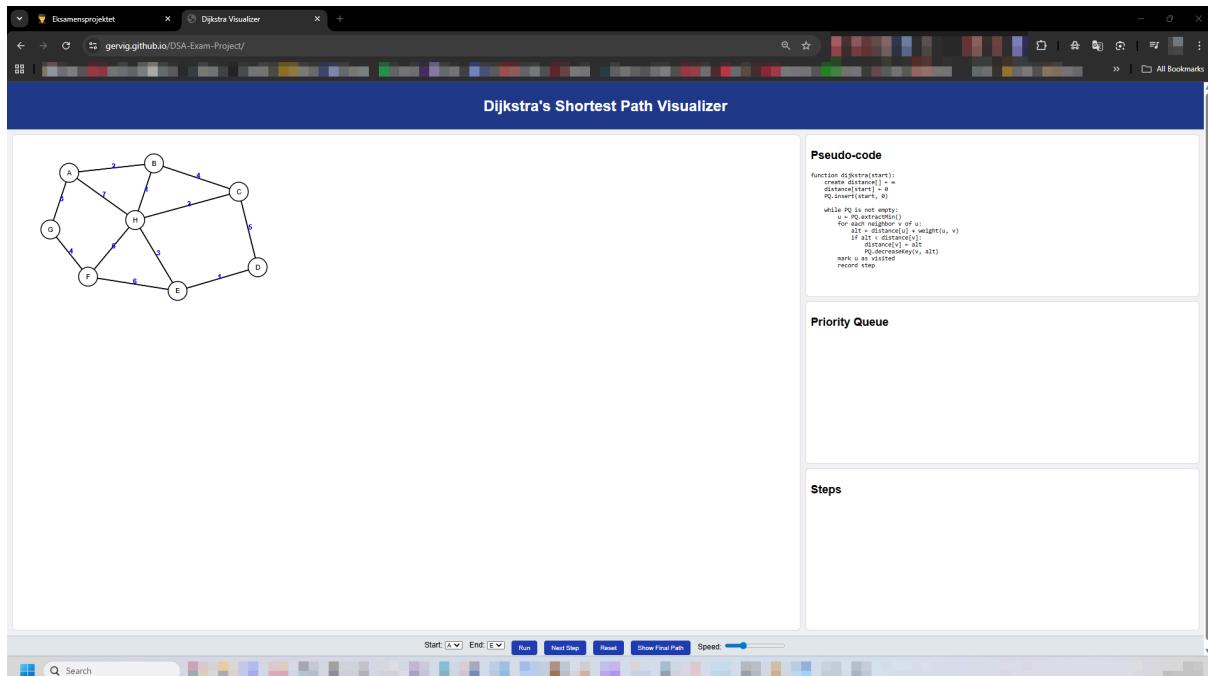


# Dijkstra's Shortest Path Visualizer



Made by **Casper Alexander Gervig**

Github: <https://github.com/Gervig/DSA-Exam-Project>

Deployed version: <https://gervig.github.io/DSA-Exam-Project/>

## Short explanation of the project:

This project is an interactive visual tool that shows how the Dijkstra algorithm finds the shortest route between two points. You can choose two nodes, and the program animates the steps the algorithm goes through, explaining each decision it makes. It's basically a way to watch the computer 'think' as it figures out the best path.

## Detailed explanation of the project:

### Dijkstra's algorithm

The algorithm I chose for this project was **Dijkstra's algorithm**. The idea is that given a graph / map, it has to find the shortest path from one node to another. It has two assumptions that need to be true in order for it to work. First, it has to be "**weighted**", meaning the cost or distances between the nodes needs to be known. Second, the weights cannot be **negative numbers**.

When we start we only know the distance to our start node, which is 0 (we are starting there). We set the distance to all other nodes to **infinity**, until we **update / relax** them. Relaxation means checking whether the current path to a neighbor is shorter than the previously known path.

But in order to **keep track of the path** we've taken to get to the end node, we need to keep track of the **previous node** we visited if and when we update our estimates.

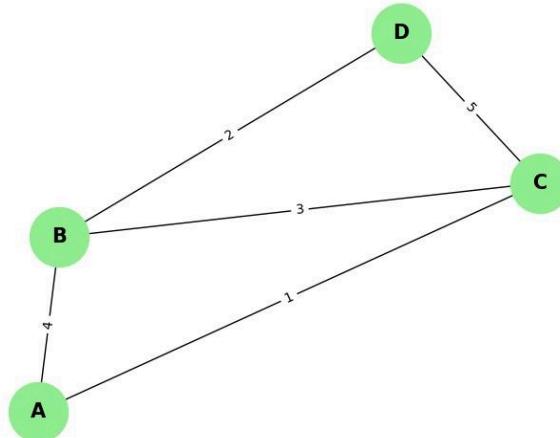
After we update our estimates, we have to choose the **next node to explore**. The rule is among all the unexplored nodes, we choose the one with the shortest path.

So the algorithm can be boiled down to three steps:

1. Update estimates
2. Track which node we visited previously
3. Choose the next node to explore

### Graphs

The **graph** consists of **nodes** (circles you "travel" between) each node is drawn as a circle marked with a letter, **edges** (the lines between the nodes) each edge is drawn as a line between each node and **weights** (the cost of "traveling" between two given nodes) shown as a number on the edges / lines.



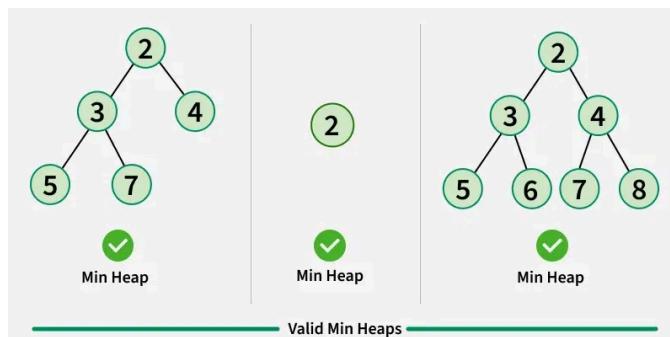
## Binary heap (min heap)

The algorithm needs a way to prioritize the closest unvisited node, so that it can explore it next. A visited node is one whose shortest distance from the start node has been finalised. In this project I chose to use a **binary heap** data structure to function as the **priority queue**. There are 2 types of binary heaps, a **min heap** (the value of the root node is the smallest) and a max heap (the value of the root node is the largest), the rules for the root nodes also apply for all subtrees. For this project I used a min heap, because it gives fast access to the minimum node. In the visualization the nodes are highlighted orange when they are being “explored”. The priority queue contains the current shortest known paths to nodes and updates them. A binary heap is shaped like a binary tree, but behaves a little different. The value of any node must be equal to or less than the value of its child nodes (left and right), we call this the **invariant** (a property that must always be true).

When **adding** new nodes to the heap, first the shape of the tree must be preserved, all levels of the tree must be filled except possibly the last one. We add at the end of the tree (left first). The newly added node might now be greater than its parent, so we swap it with the parent until the node is no longer greater than its parent. In the worst case we need to swap it all the way to the root.

When **removing** a node, the only node that is allowed to be removed is the root in this data structure. Again first we fix the shape, we take a node from the bottom and put it as the new root. Now we make swaps from the bottom down, swap the new root with the smallest child until we hit the bottom or the node is less than or equal to both children.

Both these methods give the adding and removing from the min heap a **time complexity** of  $O(\log n)$ , because at worst we need to make as many swaps as there are levels in the tree.



## Big-O

Using a binary heap priority queue gives this implementation of Dijkstra's algorithm a time complexity of  $O(V + E \log V)$ .

**V** = number of **vertices** (nodes)

**E** = number of **edges**

Each extract-min operation takes  $O(\log V)$ .

Each decrease-key/insert takes  $O(\log V)$ .

## Pseudo-code

```
function dijkstra(start):
    create distance[ ] ← ∞
    distance[start] ← 0
    PQ.insert(start, 0)

    while PQ is not empty:
        u ← PQ.extractMin()
        for each neighbor v of u:
            alt ← distance[u] + weight(u, v)
            if alt < distance[v]:
                distance[v] ← alt
                PQ.decreaseKey(v, alt)
        mark u as visited
        record step
```

## Articles and videos that served as inspiration:

### Videos:

Dijkstra's Algorithm:

[https://www.youtube.com/watch?v=EFg3u\\_E6eHU](https://www.youtube.com/watch?v=EFg3u_E6eHU)

Binary Heap:

<https://www.youtube.com/watch?v=AE5l0xAcPZs>

### Articles:

Dijkstra's Algorithm:

<https://www.geeksforgeeks.org/dsa/dijkstras-shortest-path-algorithm-greedy-algo-7/>

[https://en.wikipedia.org/wiki/Dijkstra%27s\\_algorithm](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)

Binary Heap:

<https://www.geeksforgeeks.org/dsa/binary-heap/>

Graph theory:

[https://www.tutorialspoint.com/graph\\_theory/graph\\_theory\\_weighted\\_graphs.htm](https://www.tutorialspoint.com/graph_theory/graph_theory_weighted_graphs.htm)