**Dijkstra’s Algorithm**

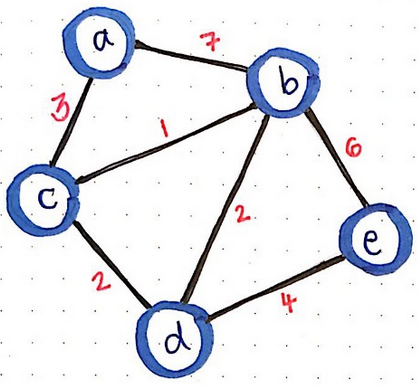
Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks, provided that the nodes are reachable from the starting node. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

The algorithm exists in many variants; Dijkstra's original variant found the shortest path between two nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

Here's a description of the algorithm:

1. Every time that we set out to visit a new node, we will choose the node with the smallest known distance/cost to visit first.
2. Once we’ve moved to the node we’re going to visit, we will check each of its neighboring nodes.
3. For each neighboring node, we’ll calculate the distance/cost for the neighboring nodes by summing the cost of the edges that lead to the node we’re checking *from* *the starting vertex*.
4. Finally, if the distance/cost to a node is *less than* a known distance, we’ll update the shortest distance that we have on file for that vertex.

Consider the following:



When we start Dijkstra’s algorithm, we do not even know if all of the other vertices are reachable, so the shortest distance from **A** to every other node is INFINITY. However, we do know the shortest distance of one node: **A**. Since we are already at **A**, our start vertex, the shortest distance is ZERO.

|  |  |  |
| --- | --- | --- |
| Vertex | Shortest Distance from **A** | Previous Vertex |
| A | 0 |  |
| B | infinity |  |
| C | Infinity |  |
| D | Infinity |  |
| E | infinity |  |

Let us keep track of which nodes we have or have not visited. We can do this with two array structures: a **visited** array and an **unvisited** array.

Visited = [] Unvisited = [a, b, c, d, e]

Next let us examine the neighbouring nodes of **A** and calculate the distance to them. The distance to node **B** is the cost of **A** plus the cost to get to node **B**, which is in this case 0+7=7. Likewise, the distance to node **C** is the cost of **A** plus the cost to get to node **C**, which is 0+3=3. Our table now looks like:

|  |  |  |
| --- | --- | --- |
| Vertex | Shortest Distance from **A** | Previous Vertex |
| A | 0 |  |
| B | 7 | A |
| C | 3 | A |
| D | Infinity |  |
| E | infinity |  |

Visited = [a] Unvisited = [b, c, d, e]

Let us repeat the same procedure as before: check the unvisited neighbours of node **C** and calculate their shortest paths from our origin node **A**. The two neighbours of node **C** which have not been visited yet are nodes **B** and **D**. The distance to node **B** is the cost of **A** plus the cost to get from node **C** to **B**, which in this case is 3+1=4. The distance to node **D** is the cost of **A** plus the cost to get from node **C** to **D**, which in this case is 3+2=5.

Let us compare these two distances to the values in our table. The distance from A to D was infinity, but through C it becomes 5. The distance from A to B through C is 4. This is shorter than the previous distance from A to B! So we now update our shortest paths.

|  |  |  |
| --- | --- | --- |
| Vertex | Shortest Distance from **A** | Previous Vertex |
| A | 0 |  |
| B | 4 | C |
| C | 3 | A |
| D | 5 | C |
| E | infinity |  |

Visited = [a, c] Unvisited = [b, d, e]

Now let us check the unvisited neighbours of **B** which is **D** and **E**. Let us calculate the distance to from A to D through B, which is 4+2=6. This is not less than the current shortest distance, so we don’t update this value. Let us calculate the distance from **A** to **E** through **B**, which is 4+6=10. Remember the shortest path from **A** to **B** is through **C**. Our table becomes:

|  |  |  |
| --- | --- | --- |
| Vertex | Shortest Distance from **A** | Previous Vertex |
| A | 0 |  |
| B | 4 | C |
| C | 3 | A |
| D | 5 | C |
| E | 10 | B |

Visited = [a, c, b] Unvisited = [d, e]

The next node we check is node **D**. Only one node of **D**’s neighbours is unvisited, which is node **E**. The distance from **A** to **E** through **D** is the distance from **A** to **D** plus the distance from **D** to **E**, which is 5+4=9. This is less than the previously calculated distance, so we have to update our table as follows:

|  |  |  |
| --- | --- | --- |
| Vertex | Shortest Distance from **A** | Previous Vertex |
| A | 0 |  |
| B | 4 | C |
| C | 3 | A |
| D | 5 | C |
| E | 9 | D |

Visited = [a, c, b, d] Unvisited = [e]

None of e’s neighbours are unvisited, so we are finished.

We run the algorithm only once, unless the graph changes.

The way to look up any shortest path in this table is by retracing our steps and following the previous vertex of any node. Let us say we want to find the shortest path from **A** to **E**. Using a stack data structure, we will start with node **E** and push it onto our stack. Then we will look at node **E**’s previous vertex, which happens to be **B**. We will push **B** onto our stack. Then we will look at **B**’s previous vertex, which happens to be node **C**. We will push **C** onto our stack. Then we look at **C**’s previous vertex which is **A** – our starting vertex. We push **A** onto the stack. Once we trace our steps all the way back to our starting vertex we can pop() each vertex off the stack, which in this case will result in the following order: **A-C-B-E**.

Dijkstra’s algorithm is used in many fields including computer networks (routing systems), google maps (the shortest possible path from one location to another), Biology (the network model in spreading of an infectious disease), etc…