Saurav Prateek's

Shortest Path Algorithm

A Mathematical Approach

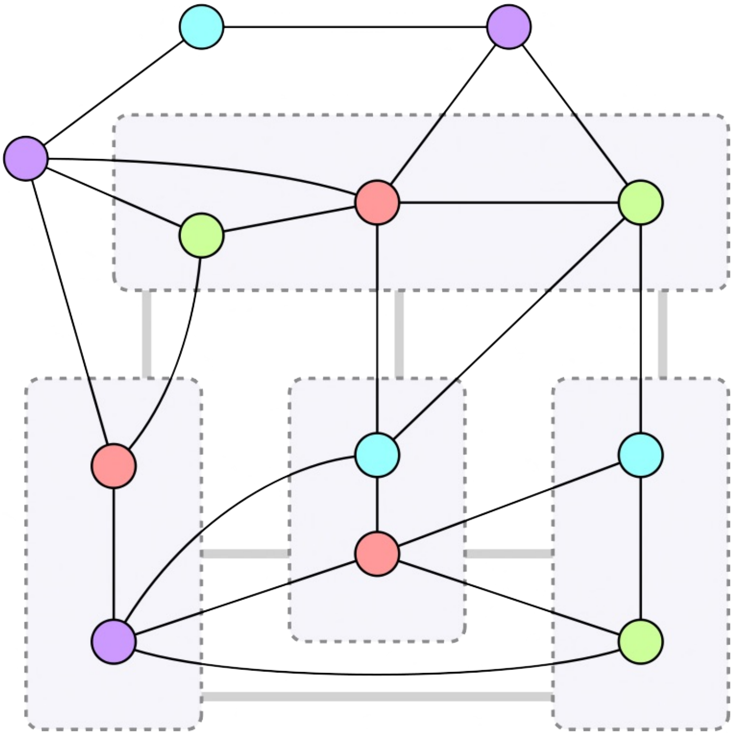
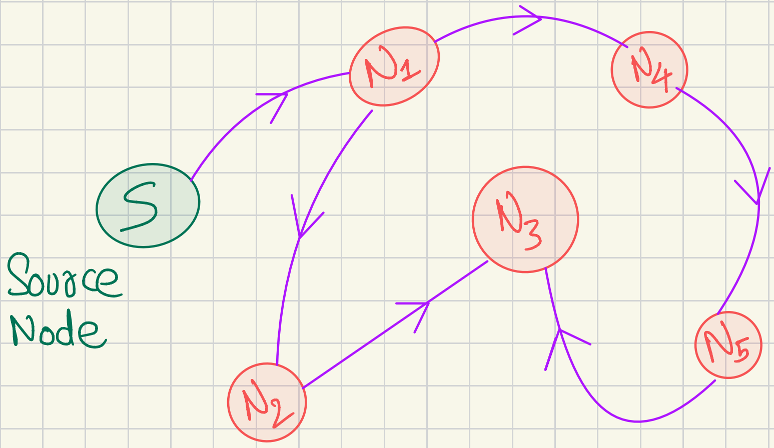


Table of Contents

Problem Statement 1

We have been given a directed-weighted graph with a source node. We need to find the shortest distance from the source node to all the other nodes present in the graph.



Brute Force Approach 2

A Brute Force approach would be to compute all the possible paths from a source node **s** to a destination node **d**. Calculate the sum of distances for every path and pick the smallest one.

But there can be a lot of possible paths to check, out of which many of them might not be worth checking in the first place.

Here, we will look into how we can avoid checking the non optimal paths and compute the shortest path efficiently

Assumptions

We will assume that we have a weighted directed graph **G(V, E)** where

$V \rightarrow$ Set of Vertices
$E \rightarrow$ Set of Edges

And,

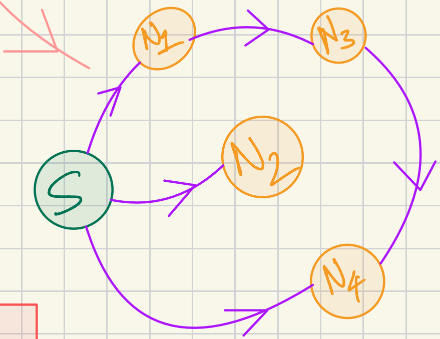
Sp(u,v) $\rightarrow$ shortest path from node **u** to node **v**

Where, **{u,v} E V**

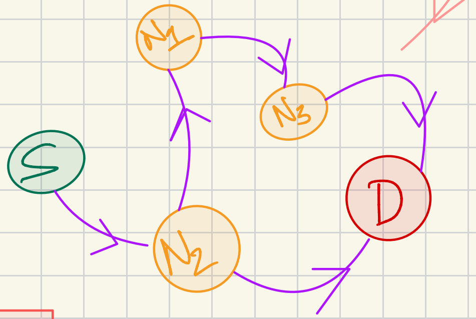
We will probably solve two variants of the shortest path problem.

1) Single Source shortest path: We have a single source **s** and we compute the shortest path from **s** to every other node present in the graph.

2) Single Pair Shortest Path: We have a source node **s** and a destination node **d** and we compute the shortest path from node **s** to **d**.



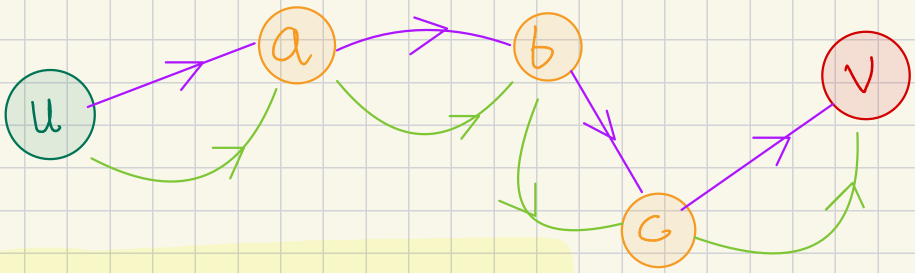
1



2

Proof: Sub-paths of a shortest path are also shortest paths

We have a weighted-directed graph **G(V, E)**. We have to prove that for a shortest path existing between the two nodes, all the sub-paths connecting intermediate nodes are also the shortest paths.

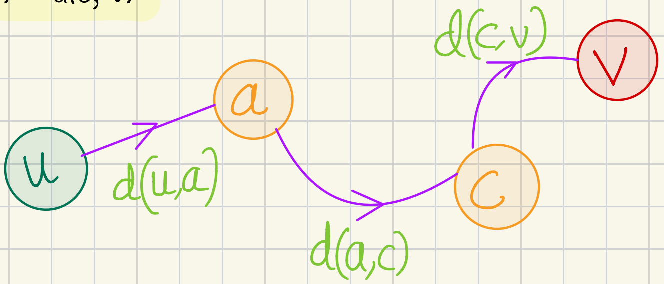


Sp(u, v) = Sp(u -> a -> b -> c -> v)
Sp(a, c) = Sp(a -> b -> c)

Let's take the above diagram as a reference to prove the statement. Let **Sp(u, v)** be the shortest path between node **u** and **v**. Also let **d(a,c)** be the sub-path distance between the intermediate nodes **a** and **c**.

So,

Sp(u, v) = d(u, a) + d(a, c) + d(c, v)



If our statement is correct then the sub-path **d(a, c)** should be the shortest path as well.

Let's contradict the statement and assume that there is an even shorter path **d`(a, c)** between node **a** and **c**.

Accordingly we will have a new path say **Sp`(u, v)** between node **u** and **v**.

$$Sp'(u, v) = d(u, a) + d'(a, c) + d(c, v)$$

Since,

$$d'(a, c) < d(a,c)$$

$$d'(a,c) + d(u, a) + d(c, v) < d(a, c) + d(u, a) + d(c, v)$$

$$\boxed{Sp'(u, v) < Sp(u, v)}$$

But, the above condition can not be true since **Sp(u, v)** is the shortest path between node **u** and **v**. Hence there can be no possible path **Sp'(u, v)** which is even smaller.

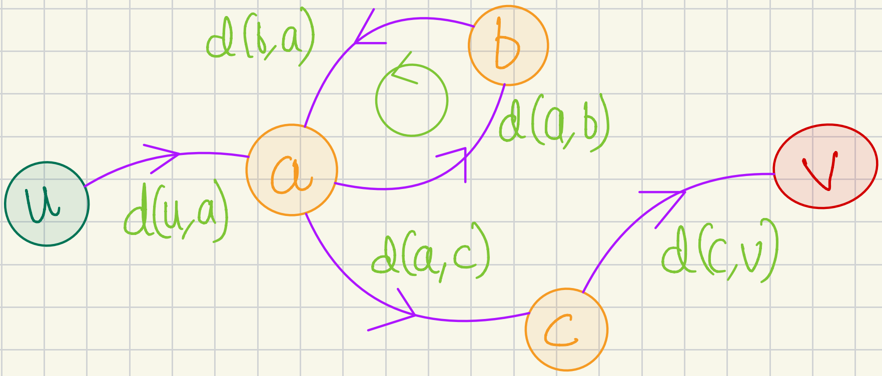
Hence, as a result there can not be any path **d'(a, c)** which is smaller than the sub-path **d(a, c)** in the shortest path between node **u** and **v**.

Proved!

Shortest Path can not have cycles

We will use the similar assumption as earlier. We have a weighted-directed graph **G(V, E)** where we have a shortest path **Sp(u, v)** between node **u** and **v**.

Let's assume that this shortest path **Sp(u, v)** contains a cycle.



From the above diagram we can assume that:

$$Sp(u, v) = d(u, a) + d(a, b) + d(b, a) + d(a, c) + d(c, v)$$

$$Sp(u, v) = d(u, a) + \{ d(a, b) + d(b, a) \} + d(a, c) + d(c, v)$$

In the previous path we have a cycle from node **a** to **b** and then back to **a**.
So, distance traversed in the cyclic path will be:

$$\rightarrow \quad d(cycle) = d(a, b) + d(b, a)$$

So,

$$Sp(u, v) = d(u, a) + \{ d(a, b) + d(b, a) \} + d(a, c) + d(c, v)$$

$$Sp(u, v) = d(u, a) + d(cycle) + d(a,c) + d(c, v)$$

$$Sp(u, v) - d(cycle) = d(u, a) + d(a, c) + d(c, v)$$

From the above derivation, we get an alternate path say:

$$Sp^{\backprime}(u, v) = d(u, a) + d(a, c) + d(c, v)$$

And,

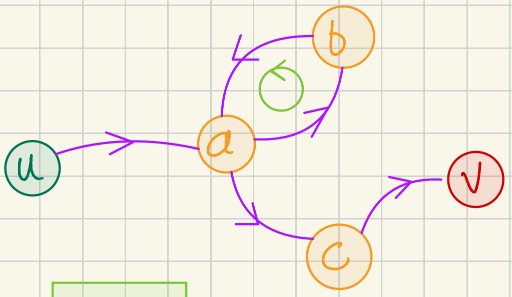
$$Sp'(u, v) = Sp(u, v) - d(cycle)$$

$$\boxed{Sp^{\backprime}(u, v) < Sp(u, v)}$$

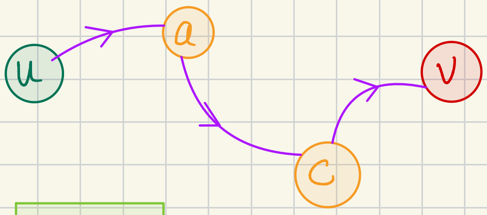
Hence, for every path (**Sp(u, v)**) containing a cycle we have an alternate shorter path (**Sp'(u,v)**) which does not have that cycle.

Proved!

We can visualise this as:



$\boxed{Sp(u, v)}$



$\boxed{Sp'(u, v)}$

Shortest Path Algorithms

The shortest path algorithms aim to **relax** the distance between the source node and the destination node to the smallest possible value.

Let's assume we have a weighted-directed graph **G(V, E)** and a source node **s**. We aim to find the shortest path from **s** to all other nodes.

A general shortest path algorithm does this using two key operations:

1. Initialisation
2. Relaxation

Initialisation

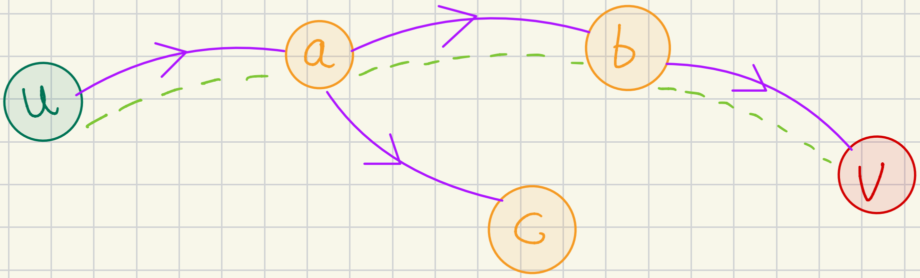
We already assumed a weighted-directed graph **G(V, E)** previously. Let's also assume we have an array **distance** where:

distance[vi] -> shortest distance from source **s** to node **vi**

We will also have another array **parent** which stores the **predecessor** node in the shortest path.

parent[vi] -> predecessor node of **vi** in the shortest path from **s** to **vi**

Let's understand this with an example:



If,

Sp(u, v) = Sp(u -> a -> b -> v)

Then,

distance[v] = distance[u -> a -> b -> v]

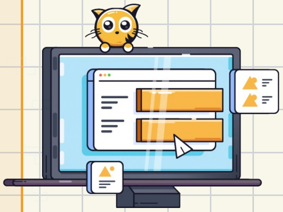
parent[v] = b

In the process of **Initialisation** we will initialise the distance of every node from the source node with a maximum possible value.

Later, in the algorithm we will try to eventually perform better by relaxing these distances.

Let's see at how the Initialisation algorithm looks like:

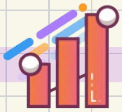
```
INITIALISATION (G(V, E), s):

  For node v IN G.V:
    distance[v] <- Infinity
    parent[v] <- null

  distance[s] <- 0
```



The above algorithm does what we discussed previously. But we can also see this line:

```
distance[s] <- 0
```

The above line makes sure that the distance of source node (**s**) from itself is always **0**. This is obvious!



Relaxation

In the previous section we initialised the distance of every other vertex in a graph from source to a maximum possible value.

In this section we will relax the distances. We will search if we can find a shorter distance from source **s** to a node.

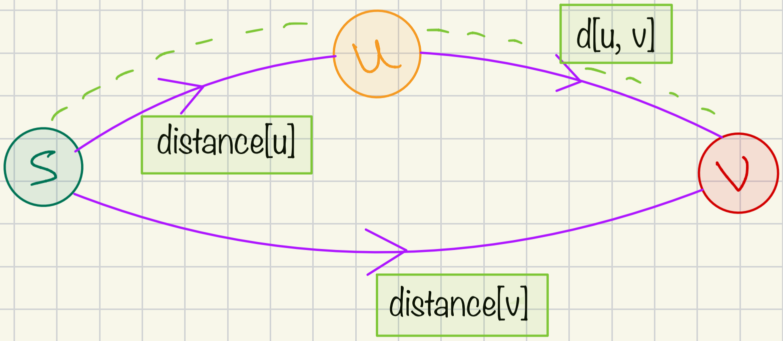
We will perform this step of **Relaxation** multiple times, till we get the best solution.

Let's look at the **Relaxation** algorithm.

```
RELAXATION(u, v, d[u, v]):

    IF distance[v] > distance[u] + d[u, v]
        distance[v] = distance[u] + d[u, v]
        parent[v] = u
```

The above algorithm exactly does what we discussed earlier. Here **d[u, v]** is the distance from node **u** to node **v**.

We can visualise this process as:



If the path going from **s -> u -> v** is smaller than path from **s -> v** then we found a better (shorter) path from node **s** to node **v** via node **u**.

This is the whole concept of **Relaxation**, where we aim to do better at every step.
These two concepts of **Relaxation** and **Initialisation** will be used ahead in our shortest path algorithm.

Dijkstra's Algorithm

The **Dijkstra's** algorithm is used to compute single source shortest path on a weighted-directed graph having non-negative weights.

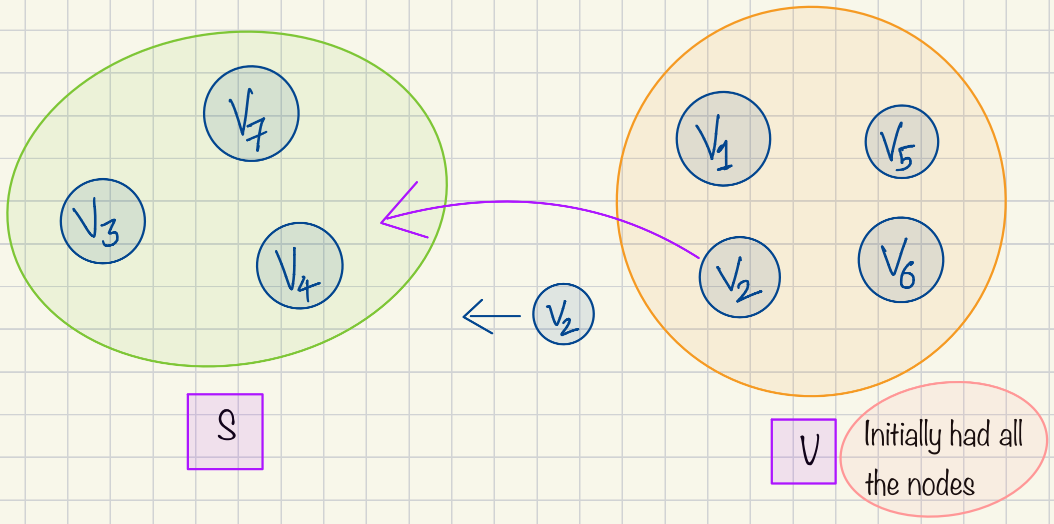
By **single-source shortest path**, we mean the shortest distance of all the other vertices of the graph from a single source vertex (say **s**).

We already discussed **INITIALISATION** and **RELAXATION**, the two key algorithms used in shortest path algorithms. We will use them in this algorithm as well.

Let us consider a weighted-directed graph **G(V, E)** where,

V => Set of Vertices/Nodes
E => Set of Edges
distance[u] => distance of node **u** from source node **s**

Dijkstra's algorithm maintains a set **S** which involves all the vertices/nodes whose shortest distance from source **s** has been calculated. The algorithm keeps on moving the nodes from set **V** to set **S**.



We start the algorithm with set **S** being **empty** and by the end all the reachable nodes from source **s** gets moved into set **S**.

The algorithm will look like this:

```
DIJKSTRAS(G, s):

  INITIALISATION(G, s)
  S <- {}
  Q <- G.V

  WHILE (Q != {}):
    u <- EXTRACT_MIN(Q)
    S <- S U {u}

    FOR v IN adjacent_nodes(u):
      RELAX(u, v, d[u, v])
```

In the above algorithm, we maintain a **queue** (**Q**). At every step we run the function **EXTRACT_MIN(Q)** which extracts the node having minimum distance from the source node **s**.

EXTRACT_MIN(Q) => MIN(distance[vi]) where **vi** E **Q**

Once the algorithm terminates, the shortest distance of every node from source node **s** will be present in the **distance** array.

distance[vi] = Shortest distance of **vi** from source node **s**

We can also compute the shortest path for every node from the source node **s** by backtracking through the **parent** array.

```
SHORTEST_PATH(u):

  WHILE(u != s):
    print(u)
    u = parent[u]
```

Note: From the above explanation we can say that Dijkstra's algorithm is a **Greedy Algorithm** since it always look for the node with the minimum distance from the source node.

Analysis of Dijkstra's Algorithm

We discussed the Dijkstra's algorithm in depth in our previous section. From the above discussion we can state out two points:

1) Every node reachable from the source node is processed once and then put into the set **S**.

2) For every node being processed, all its adjacent edges are relaxed exactly once.

Now, lets assume

V	=>	Size of vertices
E	=>	Size of edges

extract_min(Q) = Complexity of extracting the node with minimum distance factor from the Queue.

We can say the overall run-time of the algorithm can be

$$\text{run\_time} = \sum_{i=1}^{V} (\text{extract\_min}(Q) + \text{adjacent\_edges}(v_i))$$

$$= \text{extract\_min}(Q) * \sum_{i=1}^{V} (1) + \sum_{i=1}^{V} \text{adjacent\_edges}(v_i)$$

Now, for every node in the graph if we compute the sum of their adjacent edges, we will get the count of edge in the graph.

So,

$$\sum_{i=1}^{V} \text{adjacent\_edges}(V_i) = E$$

And,

$$\text{extract\_min}(Q) * \sum_{i=1}^{V} (1) = V * \text{extract\_min}(Q)$$

Hence,

$$\boxed{\text{run\_time} = (V * \text{extract\_min}(Q) + E)}$$

We can see that the runtime of the algorithm depends upon the time taken to extract the minimum node from the Queue.

Implementation I:

If we implement the Queue as an array, then we have to iterate over the array to get the node with minimum distance factor, At max our Queue can have **V** nodes.

So,

$$\text{extract\_min}(Q) = V$$

Hence,

$$\begin{aligned} \text{run\_time} &= V * V + E \\ &= V^2 + E \\ &= O(V^2 + E) \end{aligned}$$

For an array implementation of the Queue, the Dijkstra's algorithm has a runtime in the order of:

$$\boxed{O(V^2 + E) \sim O(V^2)}$$

Implementation 2:

If we implement Queue as a Priority Queue or a Binary Min Heap, then we can extract the minimum element in **log(N)** time where **N** is the maximum number of elements in the queue.

So,

$$\text{extract\_min}(Q) = \log(V)$$

Hence,

$$\begin{aligned} \text{run\_time} &= V * \log(V) + E \\ &= V.\log(V) + E \\ &= O(V.\log(V) + E) \end{aligned}$$

For a priority-queue/binary min-heap implementation of Queue, the Dijkstra's algorithm has a runtime in the order of:

$$\boxed{O(V.\log(V) + E)}$$

Hope this handbook helped you in clearing out the concept of Shortest Path Algorithms.

Software Engineer | Content Creator

Subscribe to my engineering newsletter "System That Scale"

Sharing my tech journey here!

Saurav Prateek

WEB SOLUTION ENGINEER II @ Google