Dijkstra’s Algorithm

By Sergio Martinez

Objective

In this presentation I will describe what Dijkstra's Algorithm is as well as its purpose and the logic behind it. I will also give a walkthrough of the pseudocode with explanations for each step to to help understand it.

Origin

- Dijkstra's Algorithm (pronounced dike·struhz) was created by Dutch computer scientist, Edsger W. Dijkstra of the Netherlands, in 1959. He developed it so it could serve as demonstration of the capabilities of a new computer called ARMAC.
- It has similarities to an algorithm discovered two years prior called Primm's minimal spanning tree algorithm.

Purpose

- Dijkstra's Algorithm is used to find the shortest length between a source vertex and all other vertices in a given graph.
- This algorithm will work correctly so long as the weight of all the edges are non negative

The Process

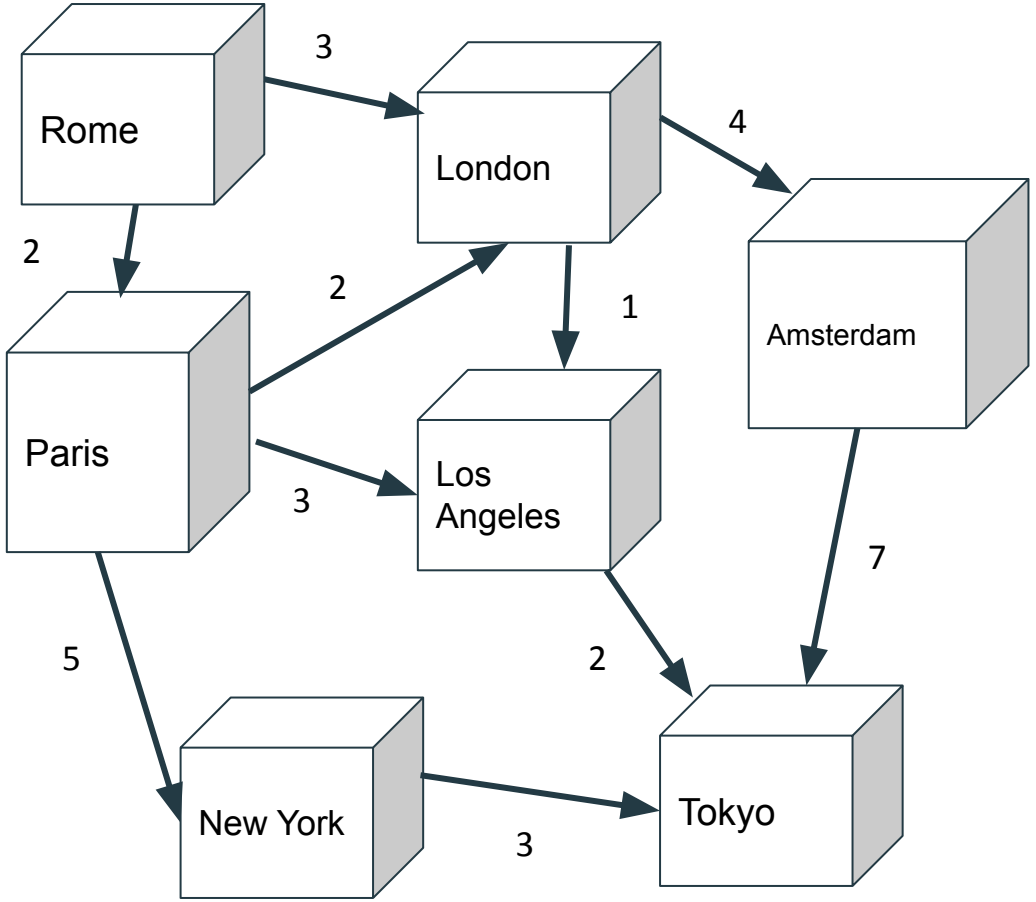
Essentially, Dijkstra's Algorithm is a 2 step process. After we begin at the source, this 2 step process is:

1. Move to next “unvisited” vertice with the shortest distance from the source.
2. If the path to another vertice from this vertice is shorter than the value it already contains, then we update the shortest distance of the vertice to this value.

After all vertices connected to this vertice are updated we repeat this process until every “unvisited” vertice is visited.

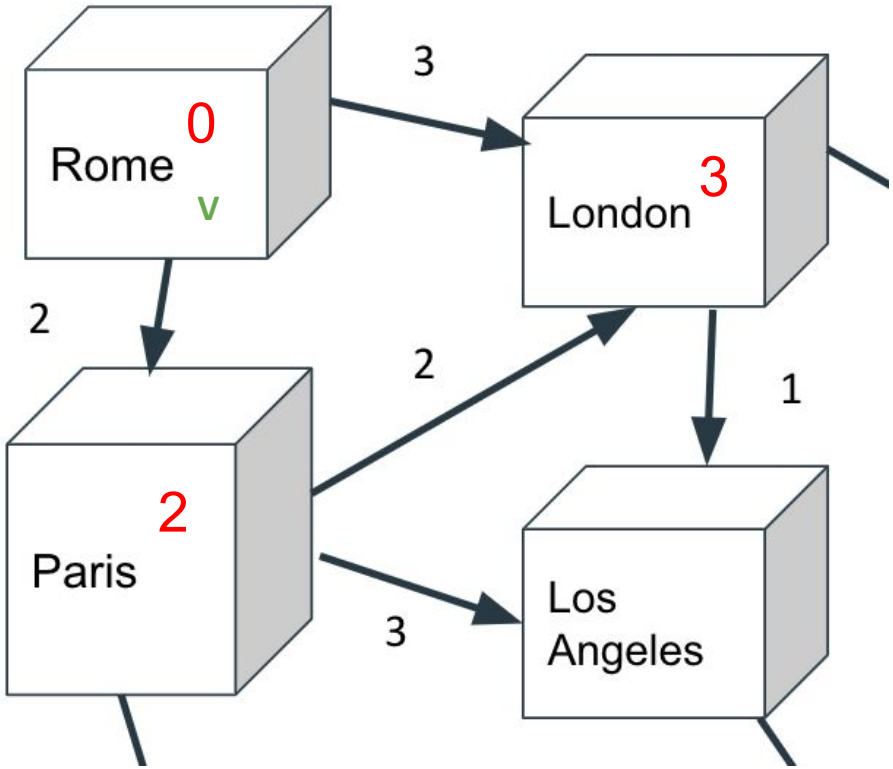
Example

We will use distance between cities as an example to help visualize this. In this example we are going to assume our source vertex is Rome. Thus, our objective is to find the shortest distance to each city from Rome. We assume that distance to begin is 0 and that every distance to each city is ∞ for the meantime.



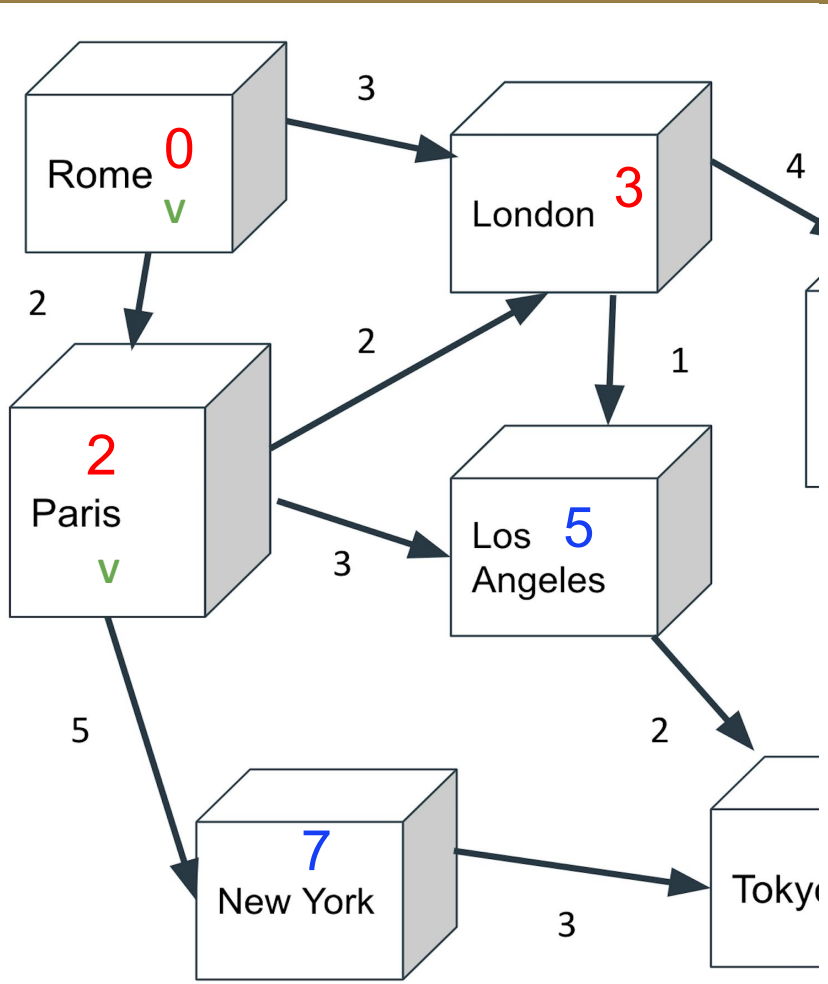
Start

To begin we start at Rome. Since the distance from Rome to Rome is 0 we place a zero beside it. We now have to look at the vertices that are connected to Rome. There is Paris and London and both are the only routes to each city. This means the shortest path to each city is the length between Rome and each city. We place this shortest value in red next to the city name. Now that we have addressed each connection for Rome, we can mark it as visited with a green ‘V’.



Next Vertice

Next we go to the Paris vertex since it is the shortest unvisited city. We update the values to each connecting city, if they are shorter than what the distance already is. New York is currently ∞ which is greater than the path: Rome to Paris to New York which is 7 (2+5). We now update the shortest distance to New York to 7. We now look at Los Angeles and do the same procdure which gives us 5. The final connecting vertex is London, but since 3 > 4, we leave it as is. Now that we addressed each vertex we can mark Paris as “visited”. Our next vertex is London since it is the closest “unvisited” city from Rome.

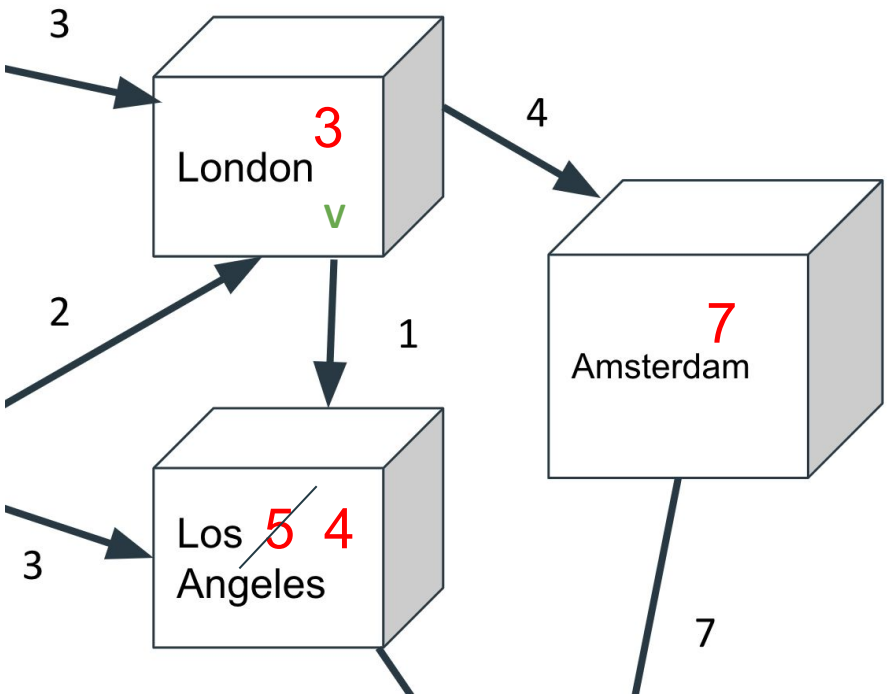


Next Vertice

Now that we are on London we have to visit Los Angeles and Amsterdam and update the values if necessary.

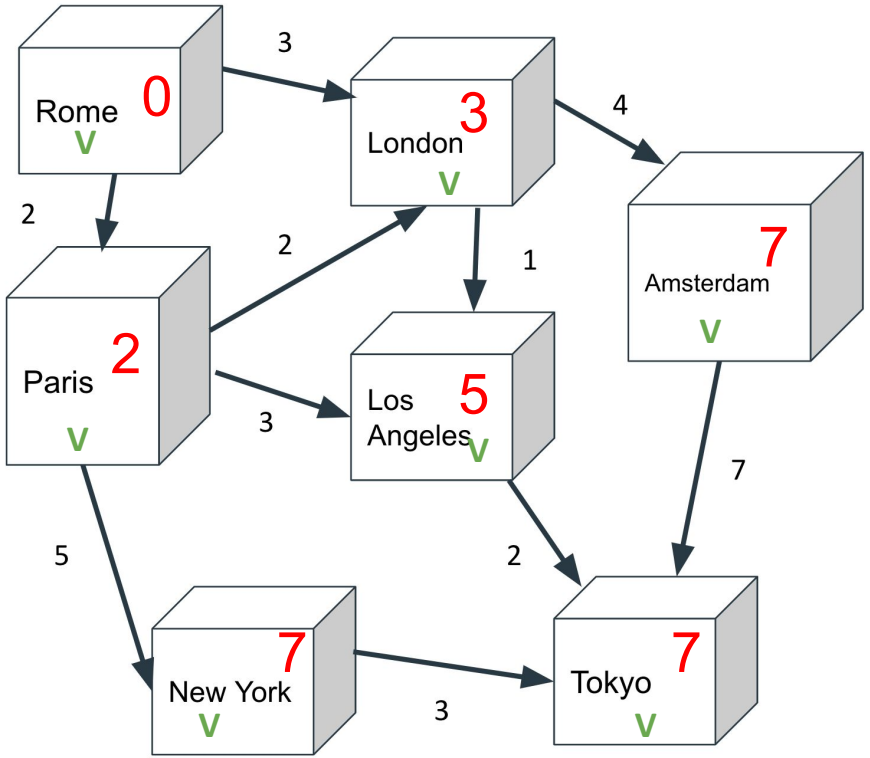
Since Amsterdam is ∞ we put a 7 (3+4) as the shortest distance. Los Angeles currently has a 5 as the shortest distance, but since we can go to London then to Los Angeles in 4 we then update its shortest distance to 4.

Now that we have visited each connection we can mark London as visited. Our next vertice to visit is Los Angeles because, again, it is our smallest unvisited vertice. We keep doing this until every vertex is visited.



Final

Once every vertex is visited we get the figure on the right that shows the shortest distance to each city, where the number in red is the distance. If we wanted to, we could have also stored the previous cities in a variable to see the route that must be taken to achieve that shortest distance. But in this case we just care about the minimum distance to each city.



Pseudocode

Prerequisite: weighted connected simple graph, with all weights positive → Need this because the algorithm will not function with negative values for the edges

Function Dijkstra(Graph, source)

for each vertex v: → Looks at each vertex (cities).

    dist[v] = ∞; → Initializes distance from source to infinity for each city

dist[source] = 0;

set all vertices to unvisited → All vertices are marked as unvisited

while city is unvisited → Keeps looping until every vertex is visited

    v = smallest dist[] of all unvisited vertices → Assigns the vertex with the smallest value of dis[] to v

    set v to visited → Marks the vertex selected as visited

    for each edge (v,w): → Loops through every edge of vertex v (all connecting cities)

      If dist[v] + length(v,w) < dist[w]:

        dist[w] = dist[v] + length(v,w) → If the distance from vertex v + the length of the edge from vertex v to vertex w is less than the distance of vertex w from the source, assign this sum to dist[w]

Applications

This is a very useful algorithm and has millions of applications in the real world. Some of these applications include:

- Digital Mapping Services
- IP routing
- Telephone networks
- Flight Agendas
- Social Networks

See Also

Another algorithm worth researching is Floyd Warshall which is a similar algorithm in the sense that it searches for the shortest possible path. The advantage of Floyd Warshall's is that it does not compute the shortest path from one source. Instead it searches all vertices for the shortest path starting from anywhere. It, unlike Dijkstra’s can function even with negative edges. Both are useful applications and each has their own distinct advantages.

Fin