1- Introduction

Have you ever thought how the message we send from our computer finds its path to some other computer? The message travels via a complicated mesh of routers and tries to go via a path which has least cost. There are two issues involved in the above problem.

The first issue that needs to be addressed is whether there is a path between source and the receiver and if there are more than one path, then which is the shortest? The above problem is a subclass of the problem which would be discussed in this chapter. The chapter introduces single-source shortest path algorithm, known as ***Dijkstra's algorithm***, to find the shortest path from a node designated as 'source' to all other nodes.

2- weighted graphs

Before we can really get into Dijkstra's algorithm, we need to pick up a few seeds of important information that we will need along the way, first.

In the previous chapter, not only have we learned about various graph traversal algorithms, but we have also taught the fundamentals of graph theory, as well as the various ways of representing graphs. We already know that graphs can be directed, or undirected, and may even contain cycles. We have also learned how we can use breadth-first search and depth-first search to traverse through them.

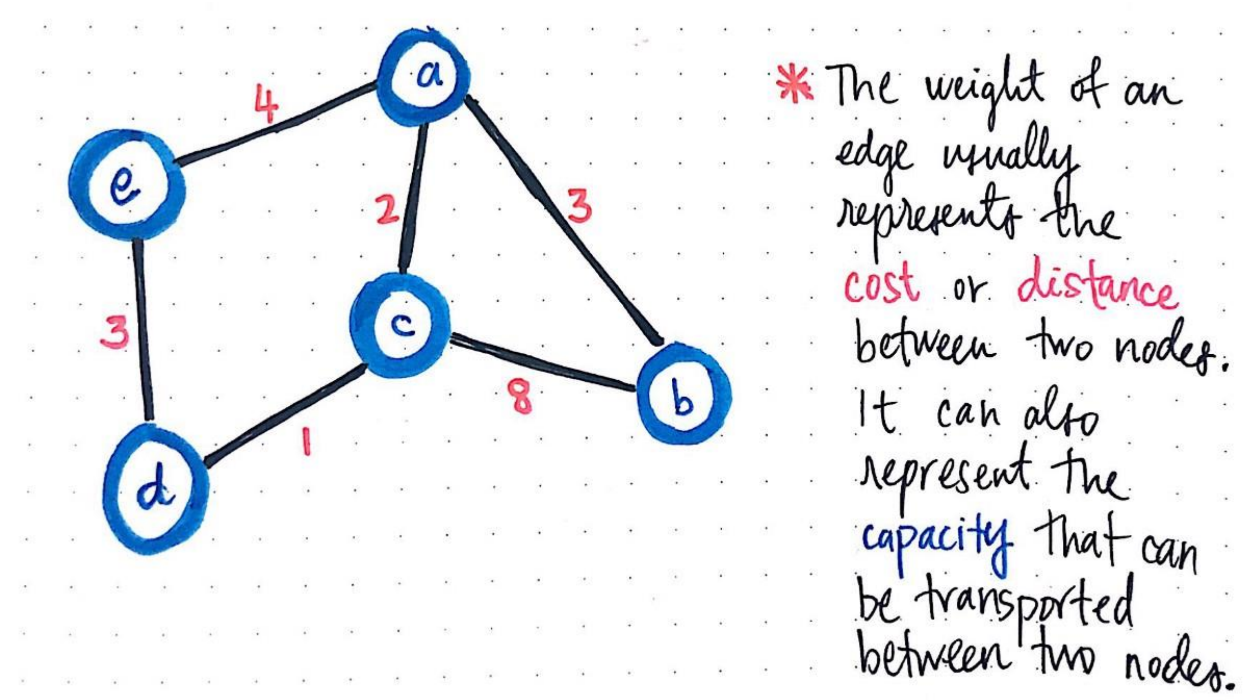
In our journey to understand graphs and the different types of graph structures that exist, there is one type of graph that we have managed to skip over—until now, that is weighted graph.

A weighted graph is interesting because it has little to do with whether the graph is directed, undirected, or contains cycles. At its core, a **weighted graph** is a graph whose edges have some sort of value that is associated with them. The value that is attached to an edge is what gives the edge its "weight".

A common way to refer to the "weight" of a single edge is by thinking of it as the *cost* or *distance* between two nodes. In other words, to go from node a to node b has some sort of cost to it.

Or, if we think of the nodes like locations on a map, then the weight could instead be the distance between nodes a and b. Continuing with the map metaphor, the "weight" of an edge can also represent the *capacity* of what can be transported, or what can be moved between two nodes, a and b.

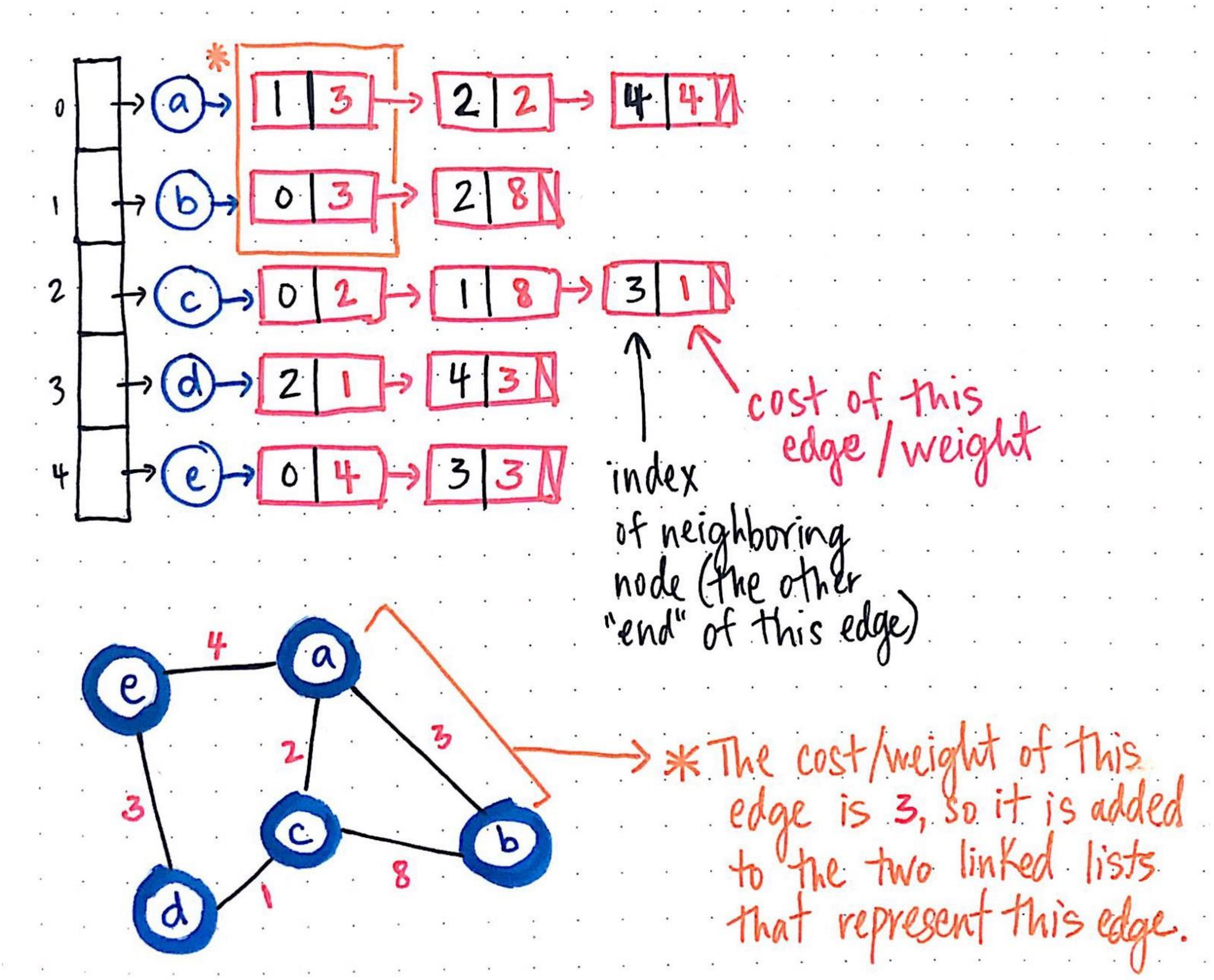
For example, in the following example, we could ascertain that the cost, distance, or capacity between the nodes c and b is weighted at 8.


a
e
c
b
d
4
2
3
3
8
1
* The weight of an edge usually represents the cost or distance between two nodes. It can also represent the capacity that can be transported between two nodes.

A weighted graph can be represented with an adjacency list, with one added property: a field to store the cost/weight/distance of every edge in the graph.

For every single edge in our graph, we will tweak the definition of the linked list that *holds* the edges so that every element in the linked list can contain two values, rather than just one. These two values will be the opposite node's index, which is how we know where this edge connects to, as well as the *weight* that is associated with the edge.

The above weighted graph, therefore, can be represented using an adjacency list as follows:



0
1
2
3
4
a
b
c
d
e
1 3
2 2
4 4
0 3
2 8
0 2
1 8
3 1
2 1
4 3
0 4
3 3
*
cost of this edge / weight
index of neighboring node (the other "end" of this edge)
4
2
3
3
1
8
*The cost/weight of this edge is 3, so it is added to the two linked lists that represent this edge.

3- Dijkstra's algorithm

Finding the shortest path between two nodes becomes much trickier when we have to take into account the weights of the edges that we are traversing through. Algorithm 1 shows the pseudocode of a single-source shortest path algorithm (also called Dijkstra's algorithm).

Algorithm 1 Single-source shortest path (Dijkstra's algorithm)

Input

• A graph G = (V, E)

• The cost corresponding to each edge

Output

• Shortest paths from the source node to all other nodes

Strategy

• Start from the source node and select the path which has minimum cost.

• The paths from the node selected and the source node are then explored. In order to go to a node, say X, the direct path from a source node and path via any of the selected nodes are considered, whichever is smaller is selected.

The array selected_vertex[ ] keeps track of the vertex selected at any instant. The initial value of each element is 0, it becomes 1 if that vertex is selected. Another array distance[ ] stores the minimum distance of a node from the source vertex. In the algorithm, *i* is a counter and *n* is the number of vertices in the graph.

```
Single Source shortest path returns distance[]
{
while ( i<n)
     {
        selected_vertex[i]=0;
     }
i=0;
selected_vertex[i]= 1;
while(i<n)
   {
     From amongst (n-1) edges (maximum) originating from I vertex,
     select the one which has minimum cost.
     Let the selected vertex be k.
     selected_vertex[k]=1;
     for each vertex m adjacent to i such that selected_vertex[m]=0
          {
               if(distance[k]>distance[m]+cost[m, k])
```

```
                    {
                         distance[k] = distance[m]+cost[m,k];
                    }
          }
   }
return distance[]
}
```

Complexity: The first loop runs *n* times, the second loop runs *n* times, and in each iteration, the inner loop runs, thus, making the complexity as $O(n^2)$. However, if all the shortest paths are to be determined using the above algorithm, then the complexity would be $O(n^3)$.

Ex(1): Apply single-source shortest path (Dijkstra's algorithm) to find the minimum distance from A of graph G1 (Fig. 1) to each vertex.

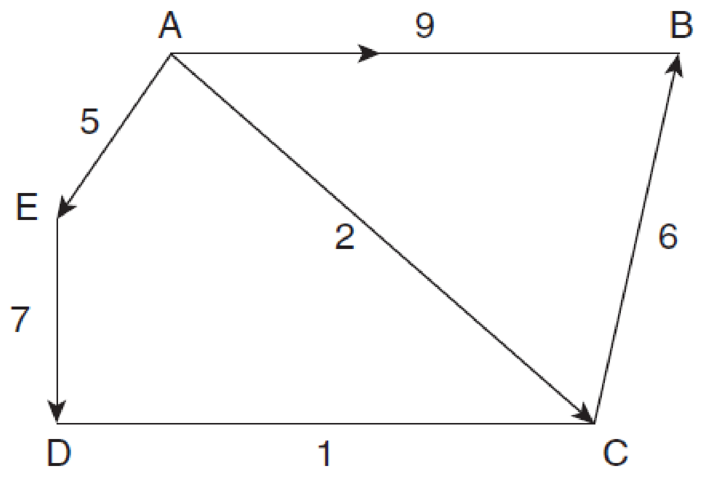


Figure 1: Graph G1

Solution From A, there are three outgoing edges having costs 5, 9, and 2. The edge having minimum cost is selected, as per the greedy approach (Fig. 2).

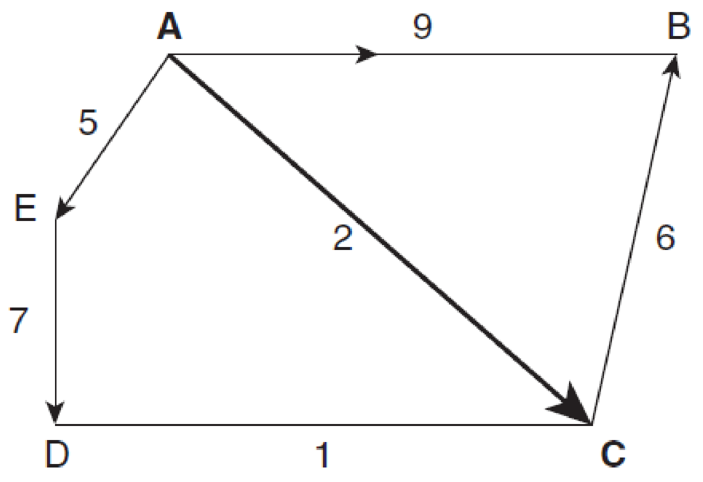


Figure 2: Graph G1, edge AC selected

The minimum cost edge from amongst the remaining edges is CD. So in order to go from A to D, the optimal path is A→C→D (Fig. 3).

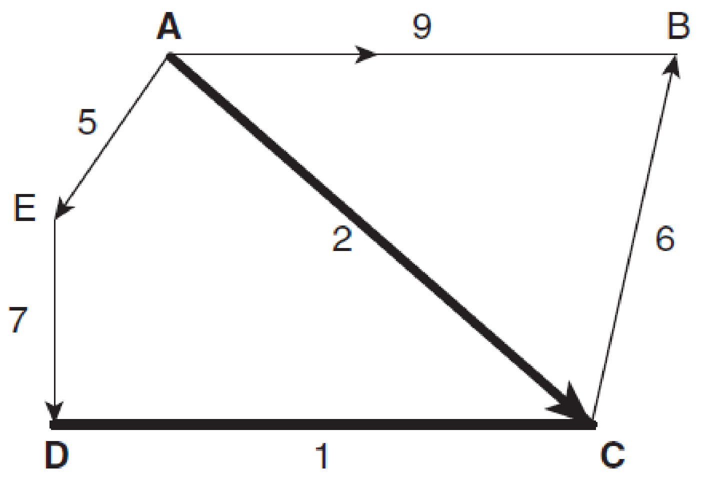


Figure 3: Graph G1, edge CD selected

A packet can traverse from A to B in two ways, either directly or through C. However, the path from C costs 2 + 6 = 8, whereas the direct cost is 9 (Fig. 4).

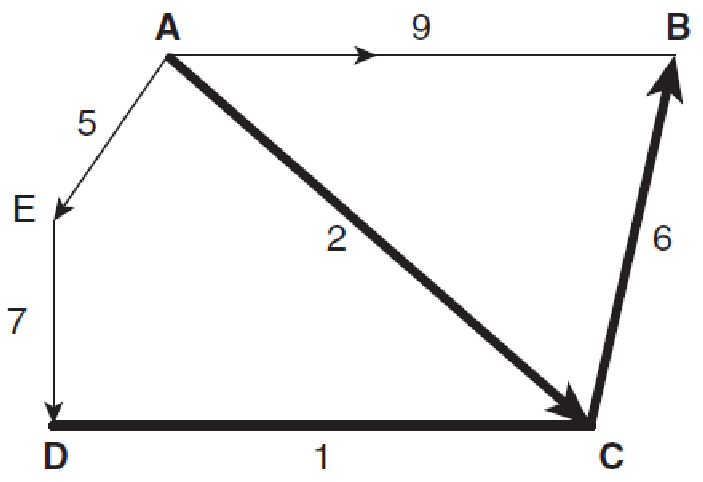


Figure 4: Graph G1, edge CB selected

A packet can traverse from A to E directly. The path costs 5 (Fig. 5).

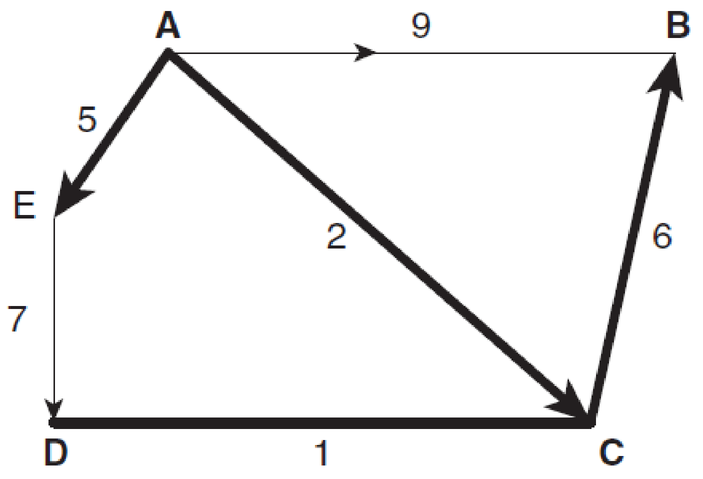
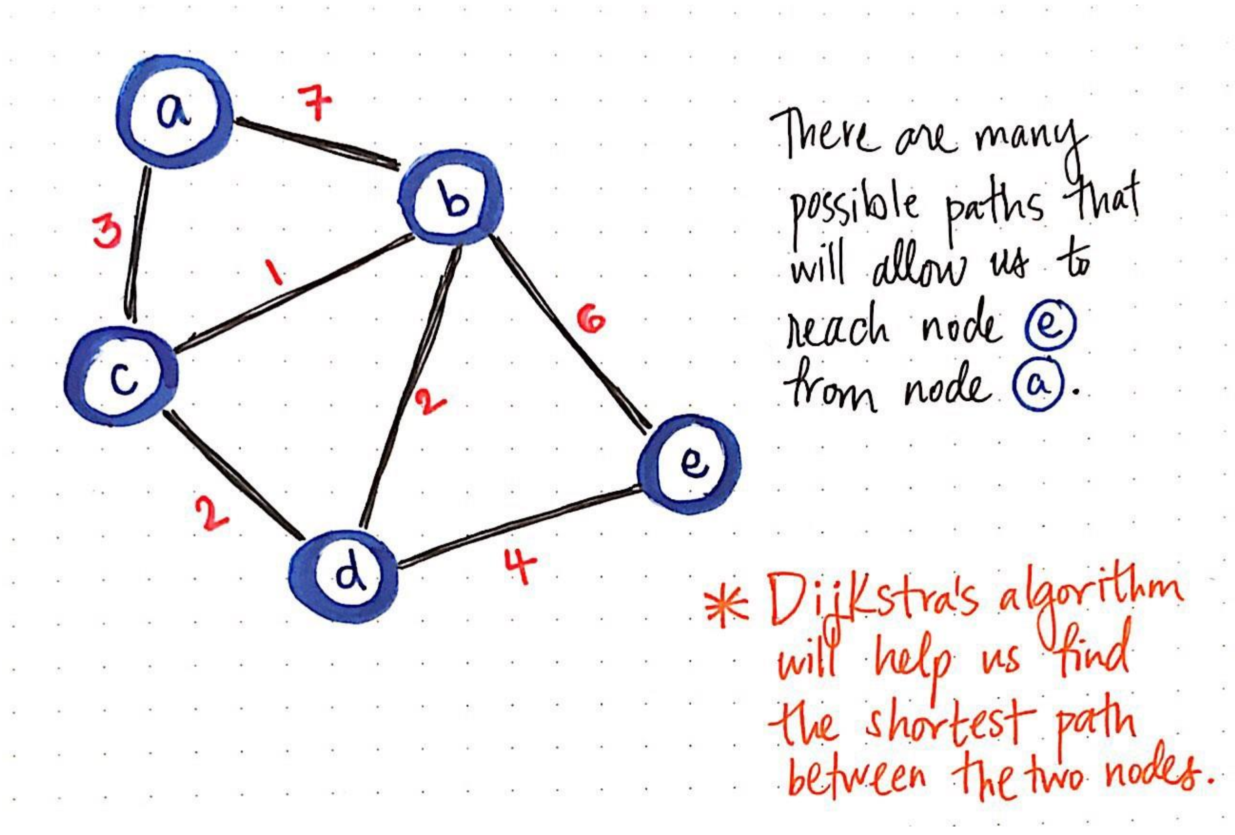


Figure 5: Graph G1, edge AE selected

Though we select the minimum cost edge at a particular point, however, the cost is also being compared with other costs (i.e., via *k*).

Ex(2): In the simple directed, weighted graph below, we have a graph with three nodes (a, b, and c), with three directed, weighted edges. Apply Dijkstra's algorithm to find the shortest path from node a to node e.


a
b
c
d
e
7
3
1
6
2
2
4
There are many possible paths that will allow us to reach node e from node a.
* Dijkstra's algorithm will help us find the shortest path between the two nodes.

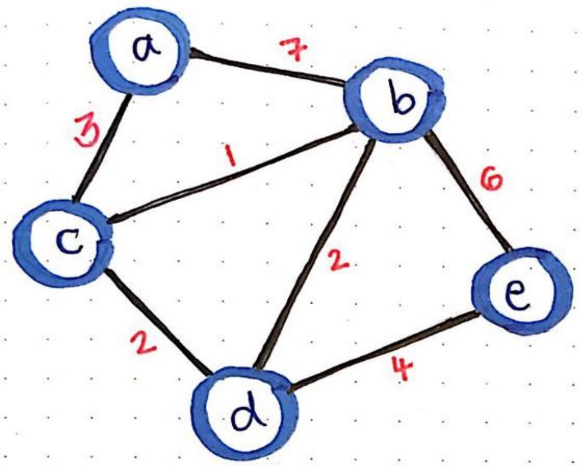
***Solution*:**

First, we need to initialize some things to keep track of some important information as this algorithm runs.

We will create a table to keep track of the shortest known distance to every vertex in our graph. We will also keep track of the previous vertex that we *came from*, before we "checked" the vertex that we are looking at currently.

Once we have our table all set up, we will need to give it some values. When we start Dijkstra's algorithm, we do not know anything at all! We do not even *know* if all of the other vertices that we have listed out (b, c, d, and e) are even *reachable* from our starting node a.

This means that, when we start out initially, the “shortest path from node a" is going to be *infinity* (∞). However, when we start out, we *do* know the shortest path for one node, and one node only: node a, our starting node, of course! Since we *start* at node a, we are already there to begin with. So, the shortest distance from node a to node a is actually just 0!

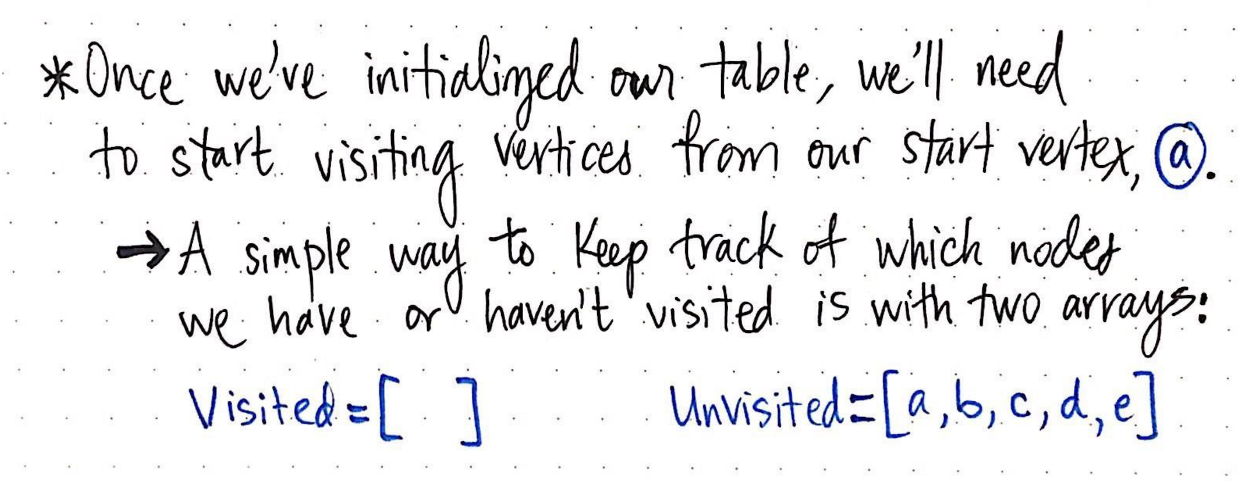


VERTEX	SHORTEST DISTANCE FROM ⓐ	PREVIOUS VERTEX
a	0	
b	∞	
c	∞	
d	∞	
e	∞	

* When we start Dijkstra's algorithm, we don't even know if all of the other vertices are reachable, so the shortest distance from ⓐ to every other node is INFINITY, ∞.

→ However, we <u>do</u> know the shortest distance of one node: ⓐ! Since we are already at ⓐ, our start vertex, the shortest distance is ZERO, 0.

Now that we have initialized our table, we will need one other thing before we can run this algorithm: a way to keep track of which nodes we have or have not visited! We can do this pretty simply with two array structures: a ***visited*** array and an ***unvisited*** array. When we start out, we haven't actually visited any nodes yet, so all of our nodes live inside of our unvisited array.



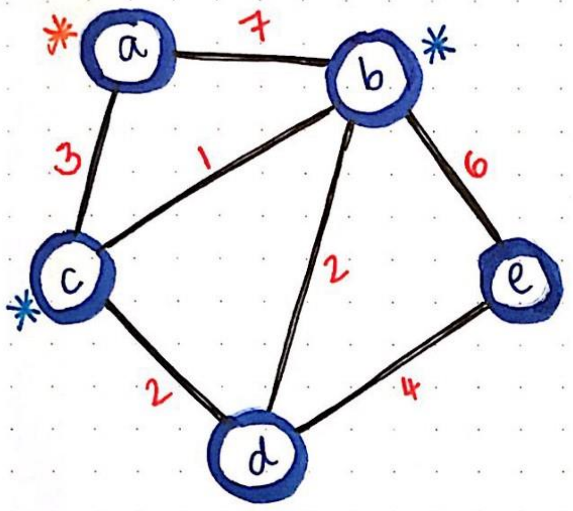
*Once we've initialized our table, we'll need to start visiting vertices from our start vertex, ⓐ.

→A simple way to keep track of which nodes we have or haven't visited is with two arrays:

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

First, we will visit the vertex with the smallest-known cost/distance. We can look at the column that tells us the shortest distance from a. Right now, every vertex has a distance of infinity (∞), except for a itself! So, we will visit node a.

Next, we will examine it's neighboring nodes, and calculate the distance to them from the vertex that we are currently looking at (which is a). The distance to node b is the cost of a *plus* the cost to get to node b: in this case, 7. Similarly, the distance to node c is the cost of a *plus* the cost to get to node c: in this case, 3.



VERTEX	SHORTEST DIST. FROM ⓐ	PREVIOUS VERTEX
a	0	
b	∞	
c	∞	
d	∞	
e	∞	

Visited = [ ]

Unvisited = [ⓐ, b, c, d, e]

↑ current vertex

* Visit the vertex with the smallest-Known cost.

* Examine its neighboring nodes, and calculate the distance to them from the vertex we are visiting.

- distance to ⓑ : 0 + 7 = 7
- distance to ⓒ : 0 + 3 = 3

* If the calculated distance is less than our currently-Known shortest distance, update the shortest distance for these vertices.

- for node ⓑ : 7 < ∞
- for node ⓒ : 3 < ∞

} We will update our table's values for these node's shortest distances. We'll also add ⓐ as their previous vertex.

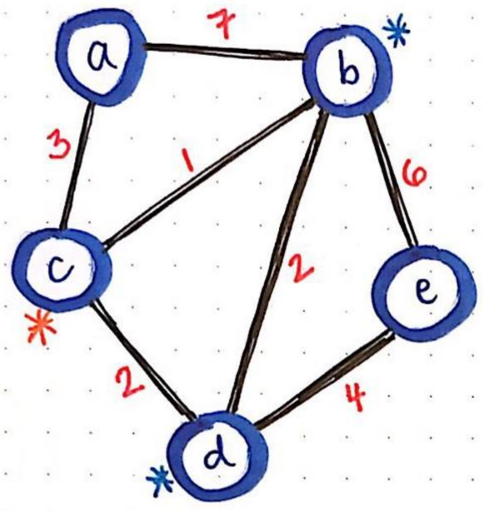
Finally, if the calculated distance is less than our currently-known shortest distance for these neighboring nodes, we will update our tables values with our new "shortest distance". Well, currently, our table says that the shortest distance from a to b is ∞, and the same goes for the shortest distance from a to c. Since 7 is less than infinity, and 3 is less than infinity, we will update node b's shortest distance to 7, and node c's shortest distance to 3. We will also need to update the previous vertex of both b and c, since we need to keep a record of where we came from to get these paths! We will update the previous vertex of b and c to a, since that is where we just came from.

Now, we are done checking the neighbors of node a, which means we can mark it as visited! Onto the next node.

Again, we will look at the node with the smallest cost that has not been visited yet. In this case, node c has a cost of 3, which is the smallest cost of all the unvisited nodes. So, node c becomes our current vertex.

We will repeat the same procedure as before: check the unvisited neighbors of node c, and calculate their shortest paths from our origin node, node a. The two neighbors of node c that have not been visited yet are node b and node d. The distance to node b is the cost of a *plus* the cost to get from node c to b: in this case, 4. The distance to node d is the cost of a *plus* the cost to get from node c to d: in this case, 5.

Now, let 's compare these two "shortest distances" to the values that we have in our table. Right now, the distance to d is infinity, so we have certainly found a shorter-cost path here, with a value of 5. But what about the distance to node b? Well, the distance to node b is currently marked as 7 in our table. But, we have found a shorter path to b, which goes through c, and has a cost of only 4. So, we will update our table with our shorter paths!



	VERTEX	SHORTEST DIST. FROM ⓐ	PREVIOUS VERTEX
✓	a	0	
	b	~~∞~~ 7	a
	c	~~∞~~ 3	a
	d	∞	
	e	∞	

Visited = [a]

Unvisited = [b, ⓒ, d, e]

↑ current vertex

* We'll next head over to vertex ⓒ—remember, we need to visit the node with the smallest-known cost. Since ⓒ's cost is the smallest of our unvisited nodes, that's what we'll check next.

* Two out of three of ⓒ's neighbors are unvisited, so we'll check them + their shortest paths, from the start vertex, via ⓒ. ⟹

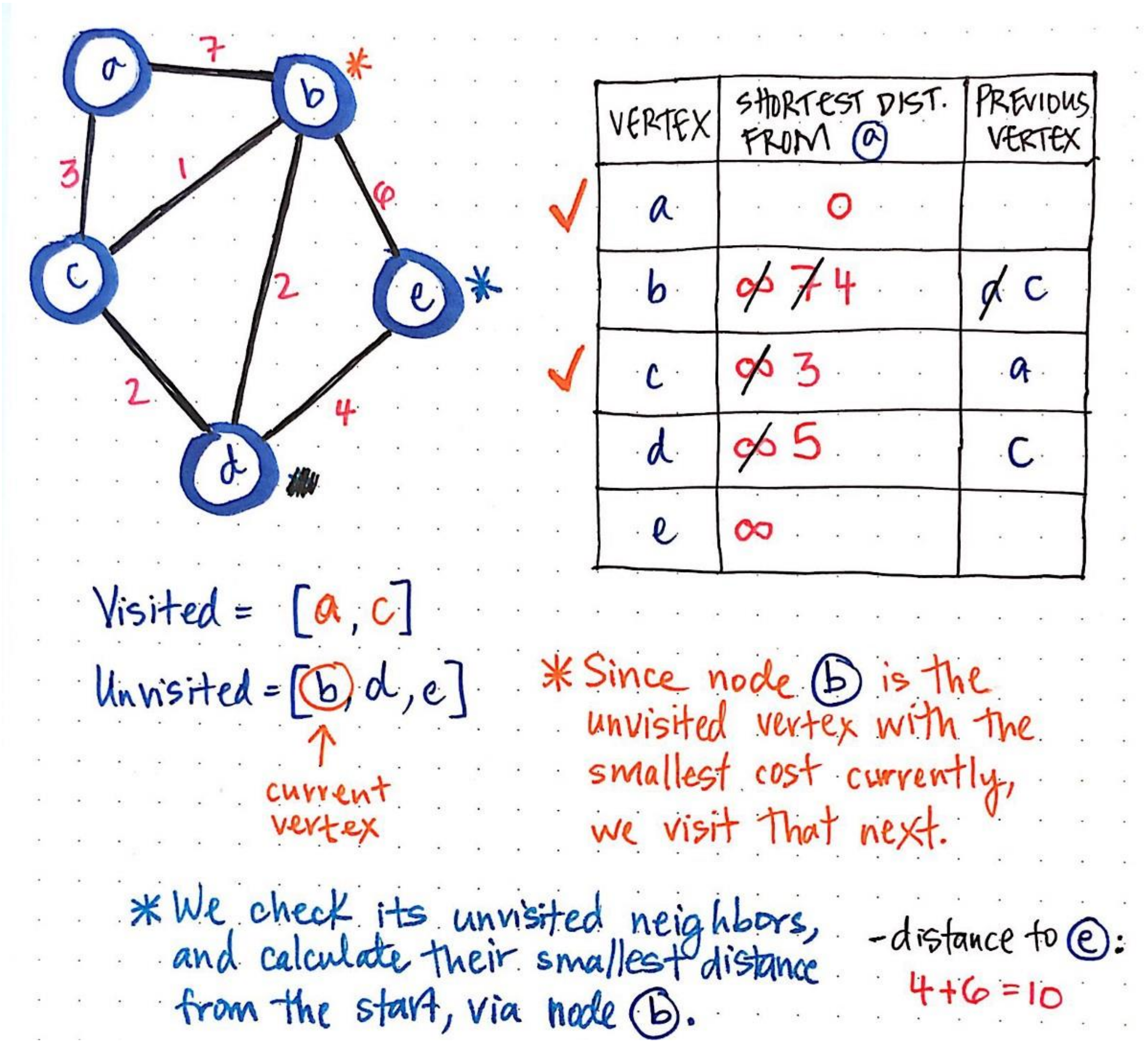
- distance to ⓑ: 3+1 = 4
- distance to ⓓ: 3+2 = 5

☆ Notice that the distance to ⓑ via node ⓒ is 4. This is shorter than the currently-known shortest distance in our table, which is 7.

→ We can update our shortest distance + previous vertex values for ⓑ, since we found a better path:

b	~~∞~~ ~~7~~ 4	~~a~~ c

We will also need to add vertex c as the previous vertex of node d. Notice that node b already has a previous vertex, since we found a path before, which we now know is not actually the shortest. No worries—we will just cross out the previous vertex for node b, and replace it with the vertex which, as we now know, has the shorter path: node c.



	VERTEX	SHORTEST DIST. FROM ⓐ	PREVIOUS VERTEX
✓	a	0	
	b	~~∞~~ ~~7~~ 4	~~a~~ c
✓	c	~~∞~~ 3	a
	d	~~∞~~ 5	c
	e	∞	

Visited = [a, c]

Unvisited = [b, d, e]

↑ current vertex

* Since node ⓑ is the unvisited vertex with the smallest cost currently, we visit that next.

* We check its unvisited neighbors, and calculate their smallest distance from the start, via node ⓑ.

- distance to ⓔ: 4+6=10

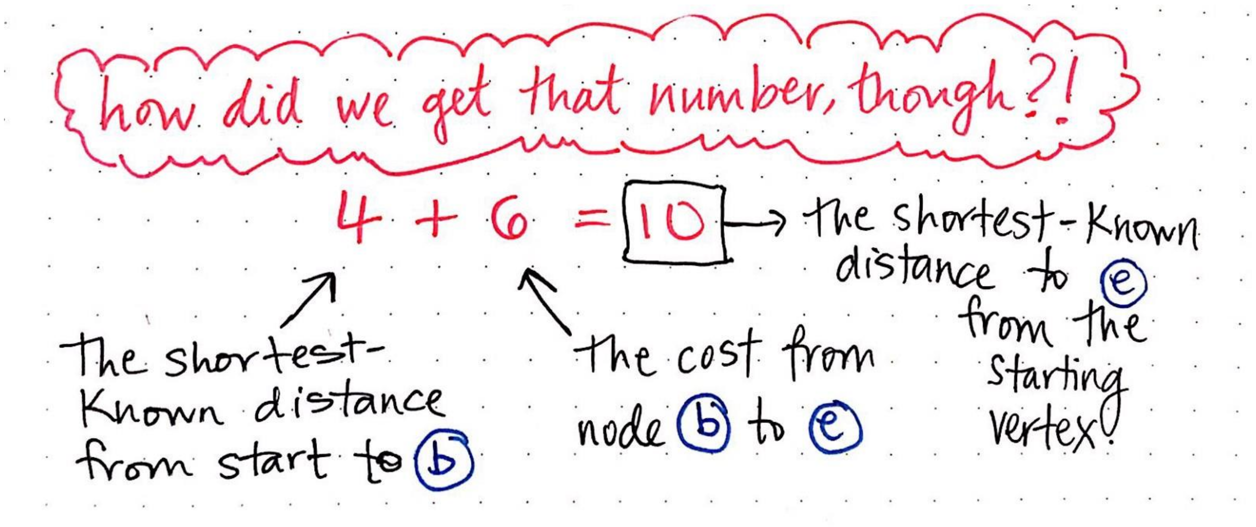
Alright, so now we have visited both node a and c. So, which node do we visit next? Again, we will visit the node that has the smallest cost; in this case, that looks to be node b, with a cost of 4.

We will check its unvisited neighbor (it only has one, node e), and calculate the distance to e, from the origin node, *via* our current vertex, b.

If we sum the cost of b, which is 4, with the cost that it takes to *get from b to e*, we'll see that this costs us 6. Thus, we end up with a total cost of 10 as the shortest-known distance to e, from the starting vertex, via our current node.

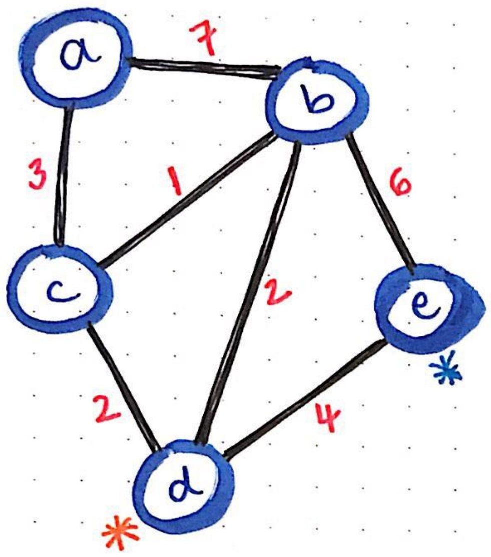
So, how did we get that number? It can seem confusing at first, but we can break it down into parts. Remember, no matter which vertex we are looking at, we always want to sum the shortest-known distance from our start to our current vertex. In simpler terms, we are going to look at the "shortest distance" value in our table, which will give us, in this example, the value 4. Then, we will look at the cost from our current vertex to the neighbor that we are examining. In this case, the cost from b to e is 6, so we will add that to 4.

Thus, 6 + 4 = 10 is our shortest-known distance to node e from our starting vertex.


how did we get that number, though?!
4 + 6 = 10
the shortest-known distance to e from the starting vertex!
The shortest-known distance from start to b
the cost from node b to e

We will continue doing the same steps for each vertex that remains unvisited. The next node we would check in this graph would be d, as it has the shortest distance of the unvisited nodes. Only *one* of node d's neighbors is unvisited, which is node e, so that is the only one that we will need to examine.

When we sum the distance of node d and the cost to get from node d to e, we will see that we end up with a value of 9, which is less than 10, the current shortest path to node e. We will update our shortest path value and the previous vertex value for node e in our table.



	VERTEX	SHORTEST DIST. FROM ⓐ	PREVIOUS VERTEX
✓	a	0	
✓	b	~~∞~~ ~~7~~ 4	~~a~~ c
✓	c	~~∞~~ 3	a
	d	~~∞~~ 5	c
	e	~~∞~~ 10	b

Visited = [a, c, b,]

Unvisited = [ⓓ, e]

↑ current vertex

* Of the two remaining unvisited nodes, ⓓ has the lowest cost.

* Since only one of ⓓ's neighbors is unvisited, that is the one we will examine.

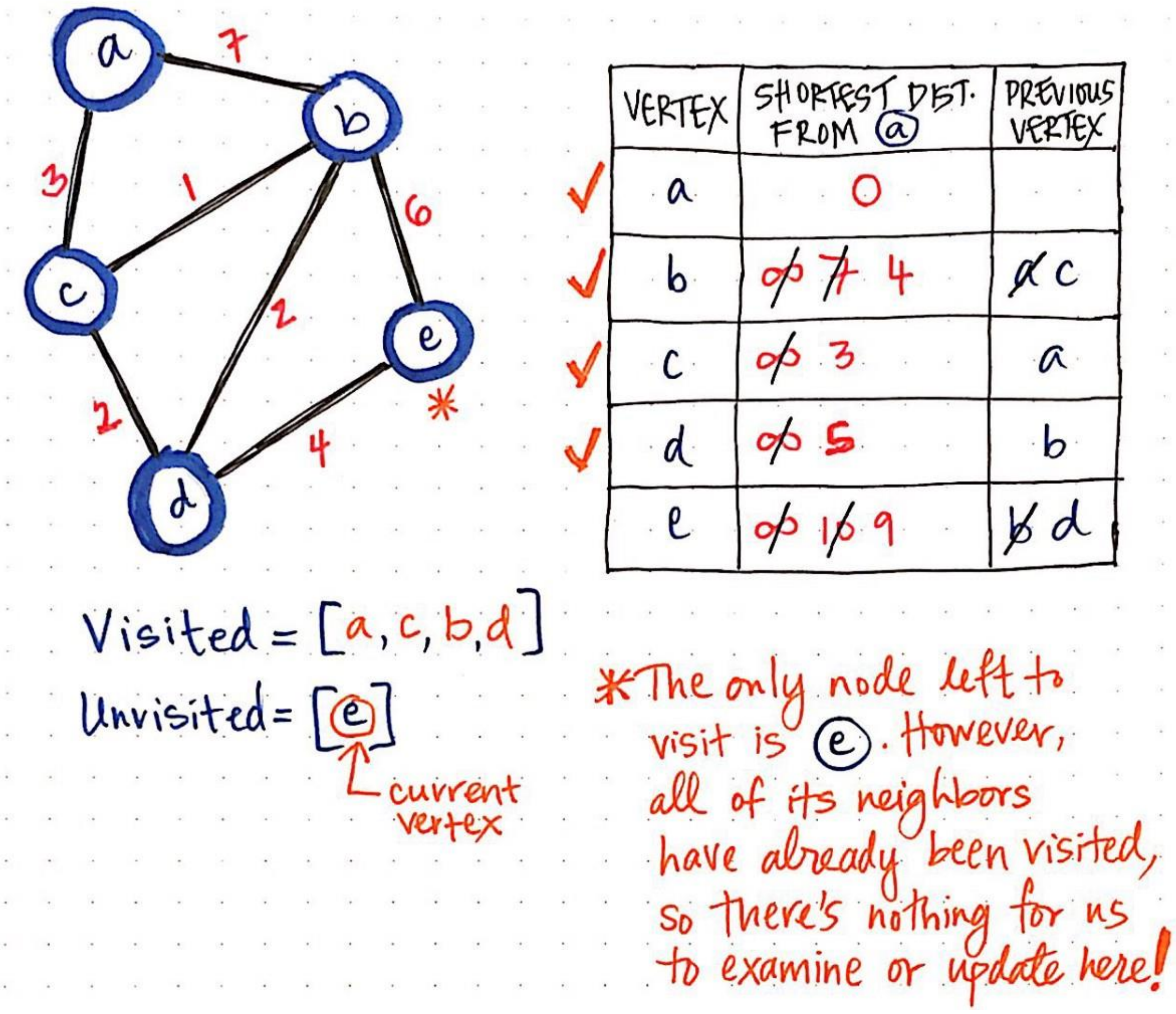
– distance to ⓔ : 5 + 4 = 9

⟹ Since we've found a shorter path to ⓔ, we'll update its row in the table.

e	~~∞~~ ~~10~~ 9	~~b~~ d

Finally, we end up with just one node left to visit: node e.

However, it becomes pretty obvious that there’s nothing for us to really *do* here! None of node e's neighbors need to be examined, since every other vertex has already been visited.



	VERTEX	SHORTEST DIST. FROM ⓐ	PREVIOUS VERTEX
✓	a	0	
✓	b	~~∞~~ ~~7~~ 4	~~a~~ c
✓	c	~~∞~~ 3	a
✓	d	~~∞~~ 5	b
	e	~~∞~~ ~~10~~ 9	~~b~~ d

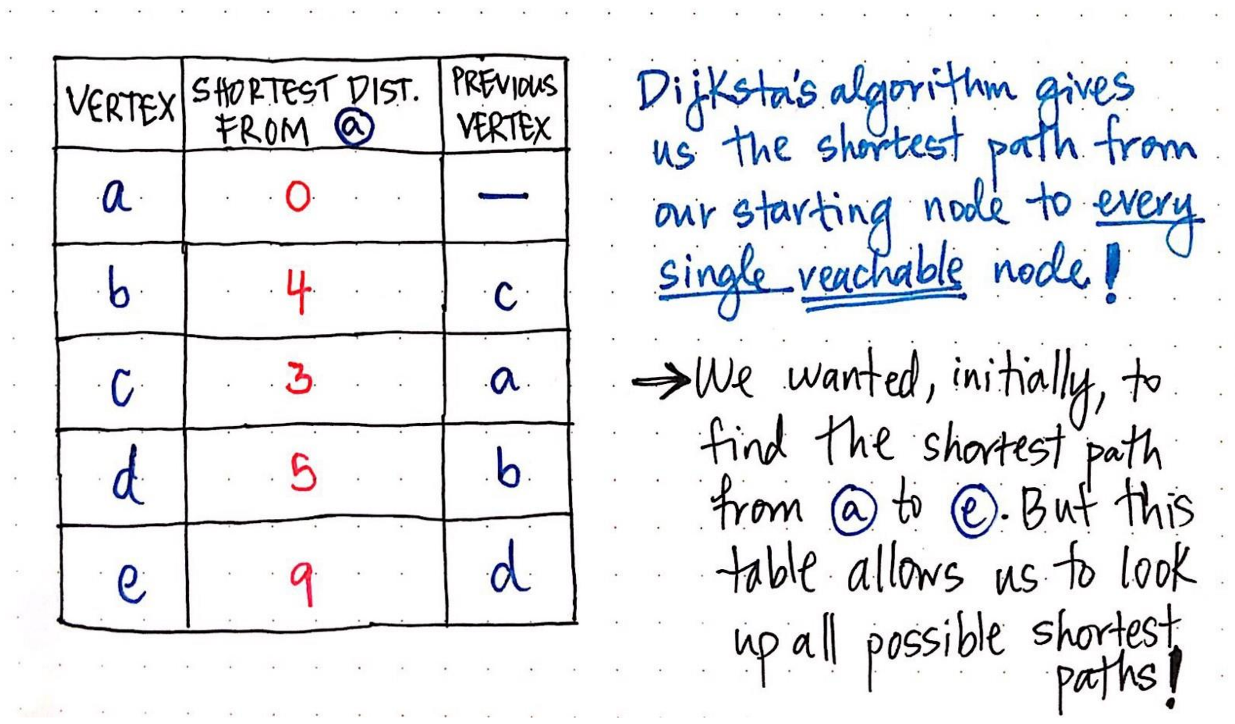
Visited = [a, c, b, d]

Unvisited = [ⓔ]

*The only node left to visit is ⓔ. However, all of its neighbors have already been visited, so there's nothing for us to examine or update here!

All we need to do is mark node e as visited. Now, we are actually completely *done* with running Dijkstra’s algorithm on this graph!

We have crossed out a lot of information along the way as we updated and changed the values in our table. Let’s take a look at a nicer, cleaner version of this table, with only the final results of this algorithm.



VERTEX	SHORTEST DIST. FROM ⓐ	PREVIOUS VERTEX
a	0	—
b	4	c
c	3	a
d	5	b
e	9	d

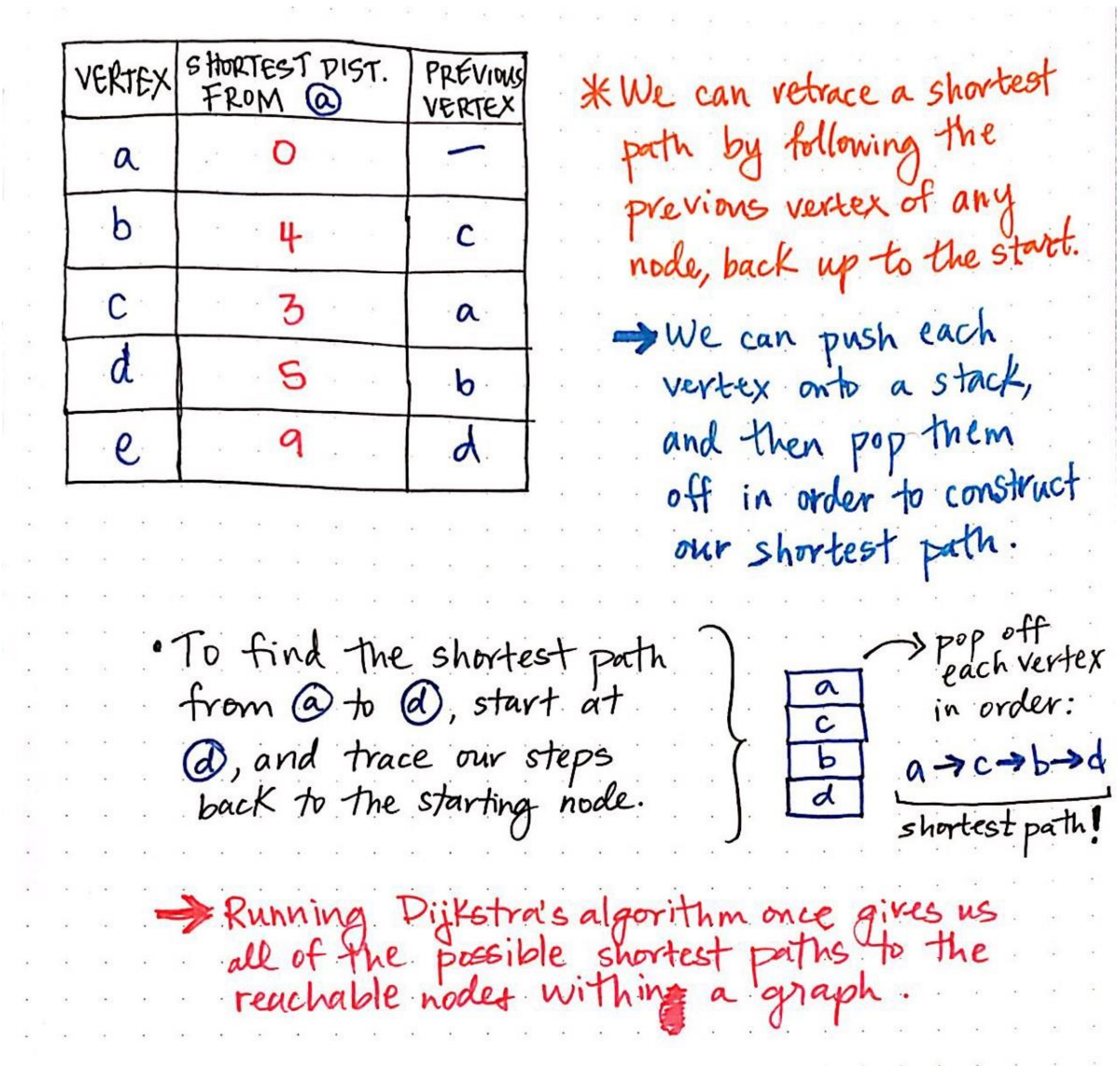
Dijksta's algorithm gives us the shortest path from our starting node to <u>every single reachable</u> node!

→We wanted, initially, to find the shortest path from ⓐ to ⓔ. But this table allows us to look up all possible shortest paths!

From looking at this table, it might not be completely obvious, but we have actually got *every single shortest path* that stems from our starting node a available here, right at our fingertips. We will remember that earlier, we learned that Dijkstra's algorithm can run once, and we can reuse all the values again and again—provided our graph does not change. This is exactly how that characteristic becomes very powerful. We *set out* wanting to find the shortest path from a to e. But, this table will allow us to look up *all* shortest paths!

The way to look up any shortest path in this table is by retracing our steps and following the "previous vertex" of any node, back up to the starting node.

For example, let's say that we suddenly decide that we want to find the shortest path from a to d. No need to run Dijkstra's algorithm again—we already have all the information we need, right here!



VERTEX	SHORTEST DIST. FROM ⓐ	PREVIOUS VERTEX
a	0	—
b	4	c
c	3	a
d	5	b
e	9	d

* We can retrace a shortest path by following the previous vertex of any node, back up to the start.

→ We can push each vertex onto a stack, and then pop them off in order to construct our shortest path.

• To find the shortest path from ⓐ to ⓓ, start at ⓓ, and trace our steps back to the starting node.

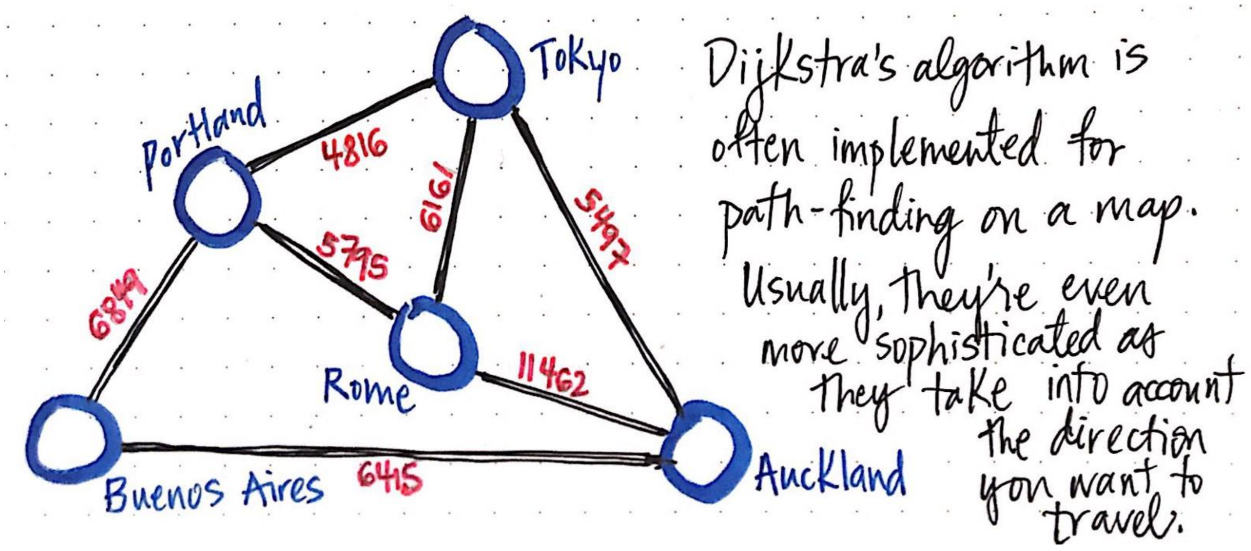
a
c
b
d

→ pop off each vertex in order:

$$\underbrace{a \to c \to b \to d}_{\text{shortest path!}}$$

→ Running Dijkstra's algorithm once gives us all of the possible shortest paths to the reachable nodes within a graph.

H.W: The most common example of Dijkstra's algorithm in the wild is in path-finding problems, like determining directions or finding a route on GoogleMaps. Apply Dijkstra's algorithm to find the shortest path from node Auckland to each vertex.


Tokyo
Portland
Rome
Buenos Aires
Auckland
4816
6161
5497
5795
6819
11462
6415
Dijkstra's algorithm is often implemented for path-finding on a map. Usually, they're even more sophisticated as they take into account the direction you want to travel.