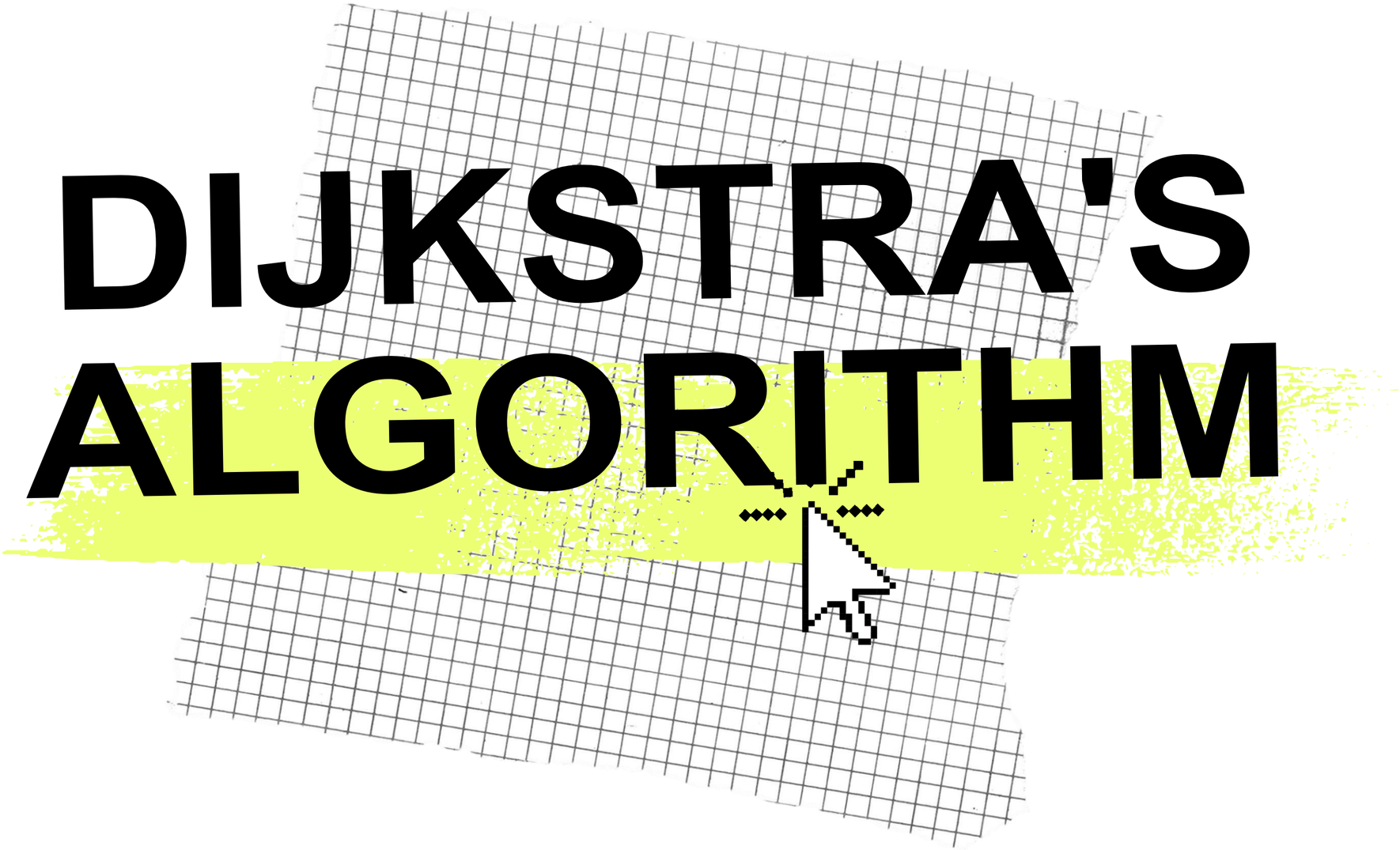
D1 SECTION
DDESIGN & ANALYSIS OF ALGORITHM

# DIJKSTRA'S ALGORITHM

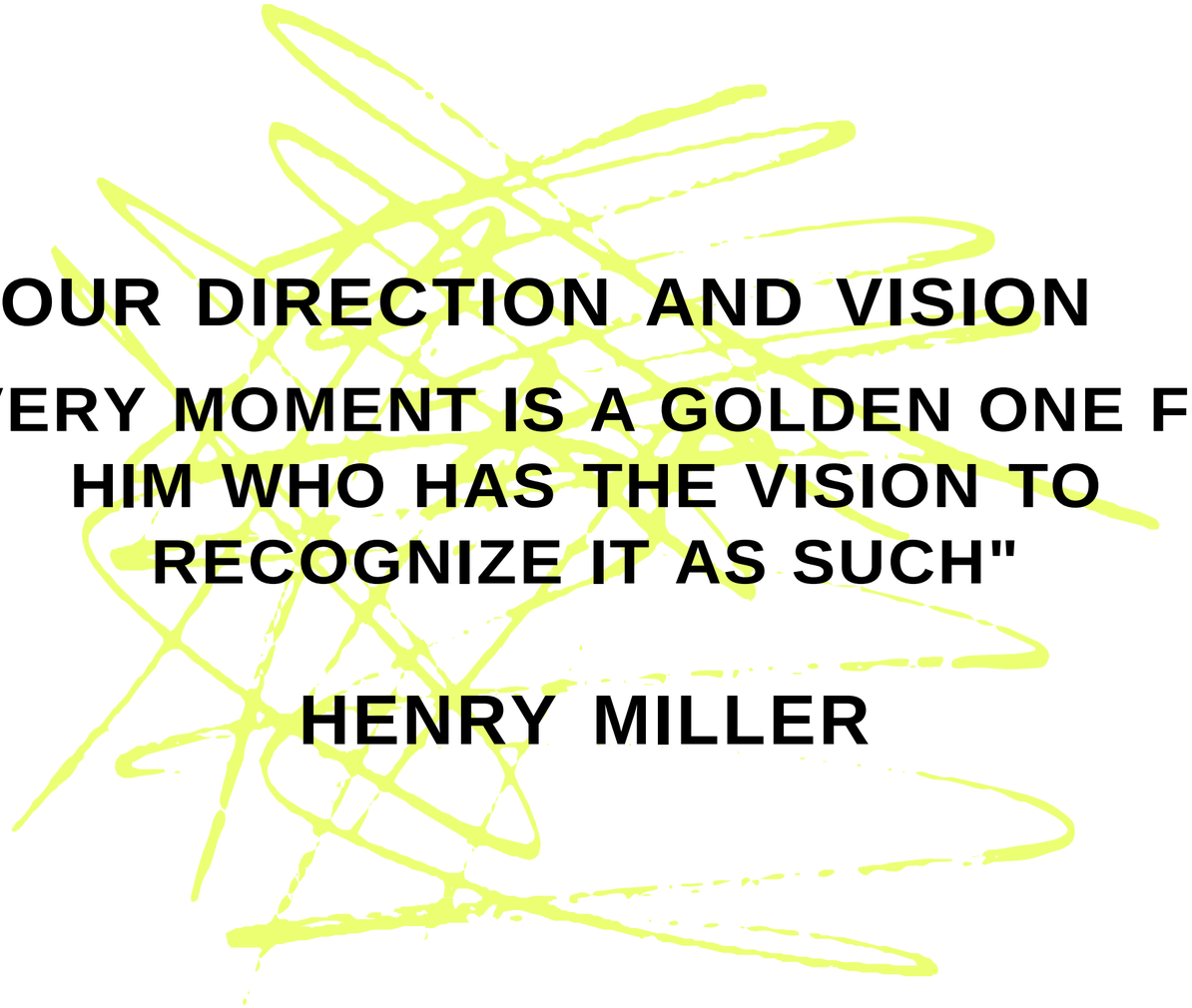


**Team leader :-Sneha Maurya (RA2011003010217)**

**Team Members :-**

**Akshat Gupta (RA2011003010221)**

**Adarsh Shukla (RA2011003010230)**



# OUR DIRECTION AND VISION

"EVERY MOMENT IS A GOLDEN ONE FOR HIM WHO HAS THE VISION TO RECOGNIZE IT AS SUCH"

HENRY MILLER

# BACKGROUND RESEARCH

## WHAT IS DIJKSTRA ALGORITHM

Dijkstra's algorithm (/ˈdaɪkstrəz/ DYKE-strəz) is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. 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 algorithm found the shortest path between two given nodes, but a more common variant fixes a single node as the "source" node and finds the shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

Dijkstra's algorithm works on undirected, connected, weighted graphs.

## HOW DIJKSTRA ALGORITHM WORKS

Dijkstra's Algorithm works on the basis that any subpath B -> D of the shortest path A -> D between vertices A and D is also the shortest path between vertices B and D.

Djikstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest subpath to those neighbors.

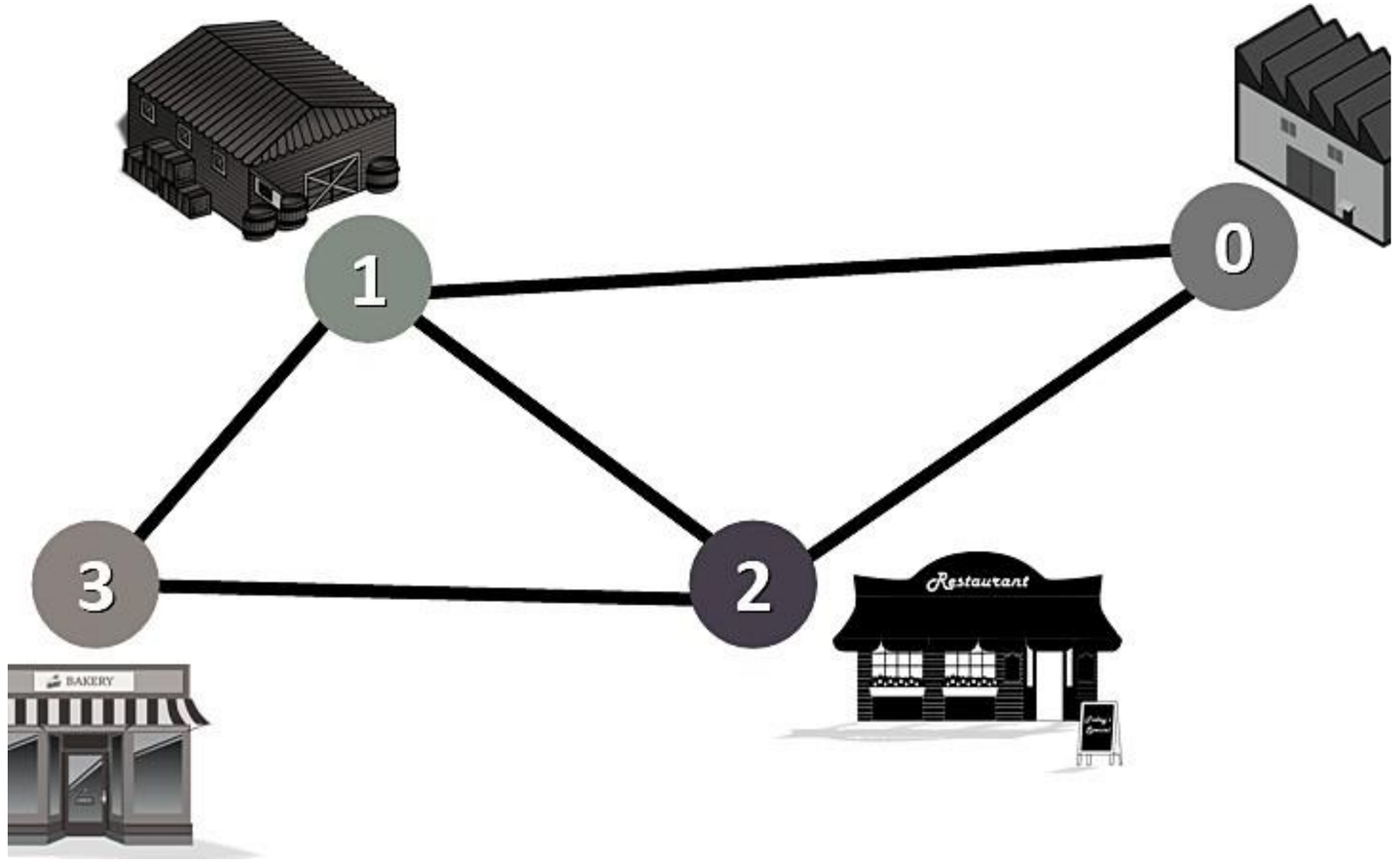
The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

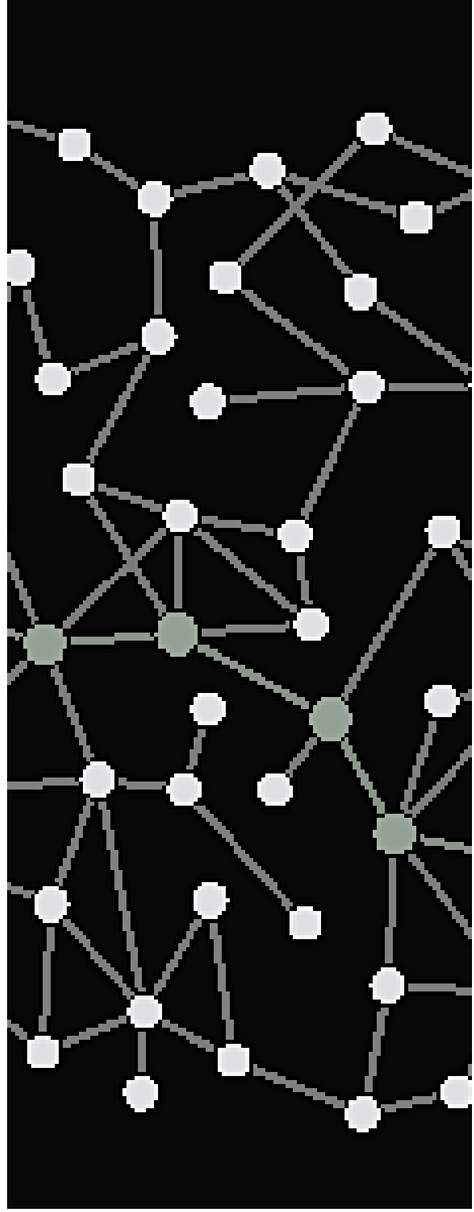
# APPLICATION OF DIJKSTRA

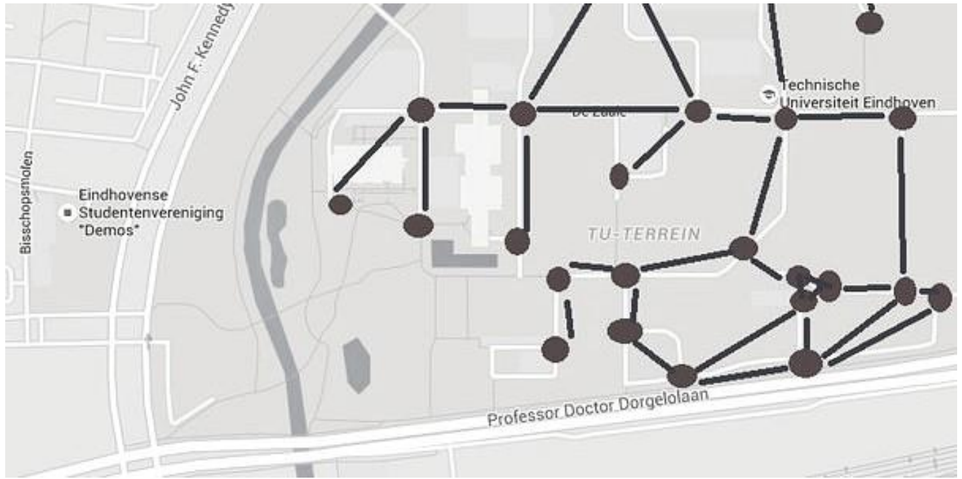
TO FIND THE SHORTEST PATH IN CONDITIONS AND SCENARIO LIKE.

1- TELEPHONE LINE.
2- MAP INDEXING.
3-USED IN IP ROUTING TO OPEN SHORTEST PATH FIRST.

# INSPIRATION GALLERY






DIJKSTRA ALGORITHM

JUNE 18

# THE PROBLEM

A CONCISE AND NON RELAPSING WAY TO FIND THE SHORTEST PATH FROM INDEX A TO INDEX B PARAMETERS LIKE TRAFFIC AND BLOCKADE ARE NOT A PART OF THE SAMPLE SPACE. REAL TIME IMPLEMENTAION GPS NAVIGATION OR AI AIDED NAVIGATION.

# USING DIJKSTRA ALGORITHM WE CAN TACKLE THE PROBLEM STATEMENT IN THE COMING SLIDES WE WILL HAVE A COHERENT AND IN DEPTH VIEW OF THE SOLUTION

# ALGORITHM

## PART - 1

Let the node at which we are starting be called the initial node. Let the distance of node Y be the distance from the initial node to Y. Dijkstra's algorithm will initially start with infinite distances and will try to improve them step by step.

Mark all nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.
Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes. During the run of the algorithm, the tentative distance of a node v is the length of the shortest path discovered so far between the node v and the starting node. Since initially no path is known to any other vertex than the source itself (which is a path of length zero), all other tentative distances are initially set to infinity. Set the initial node as current.

For the current node, consider all of its unvisited neighbours and calculate their tentative distances through the current node. Compare the newly calculated tentative distance to the one currently assigned to the neighbour and assign it the smaller one. For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbor B has length 2, then the distance to B through A will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, the current value will be kept.

# ALGORITHM

## PART -2

When we are done considering all of the unvisited neighbors of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again (this is valid and optimal in connection with the behavior in step 6.: that the next nodes to visit will always be in the order of 'smallest distance from initial node first' so any visits after would have a greater distance).
If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.

Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new current node, and go back to step 3.
When planning a route, it is actually not necessary to wait until the destination node is "visited" as above: the algorithm can stop once the destination node has the smallest tentative distance among all "unvisited" nodes (and thus could be selected as the next "current").

# PROOF OF CORRECTNESS

## PART -1

Proof of Dijkstra's algorithm is constructed by induction on the number of visited nodes.

Invariant hypothesis: For each node v, dist[v] is the shortest distance from source to v when traveling via visited nodes only, or infinity if no such path exists. (Note: we do not assume dist[v] is the actual shortest distance for unvisited nodes.)

The base case is when there is just one visited node, namely the initial node source, in which case the hypothesis is trivial.

Otherwise, assume the hypothesis for n-1 visited nodes.In which case, we choose an edge vu where u has the least dist[u] of any unvisited nodes such that dist[u] = dist[v] + Graph.Edges[v,u]. dist[u] is considered to be theshortest distance from source to u because if there werea shorter path, and if w was the first unvisited node on that path then by the original hypothesis dist[w] > dist[u] which creates a contradiction. Similarly if there were a shorter path to u without using unvisited nodes, and if the last but one node on that path were w, then we would have had dist[u] = dist[w] + Graph.Edges[w,u], alsoa contradiction.

# PROOF OF CORRECTNESS

## PART -2

After processing u it will still be true that for each unvisited node w, dist[w] will be the shortest distance from source to w using visited nodes only, because if there were a shorter path that doesn't go by u we would have found it previously, and if there were a shorter path using u we would have updated it when processing u.

After all nodes are visited, the shortest path from source to any node v consists only of visited nodes, therefore dist[v] is the shortest distance.

# PRACTICAL OPTIMISATION

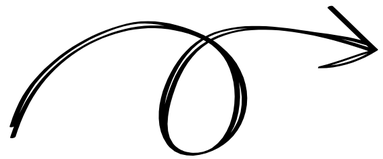
IIn common presentations of Dijkstra's algorithm, initially, all nodes are entered into the priority queue. This is, however, not necessary: the algorithm can start with a priority queue that contains only one item, and insert new items as they are discovered (instead of doing a decrease-key, check whether the key is in the queue; if it is, decrease its key, otherwise insert it). This variant has the same worst-case bounds as the common variant, but maintains a smaller priority queue in practice, speeding up the queue operations.

# PRACTICAL OPTIMISATION

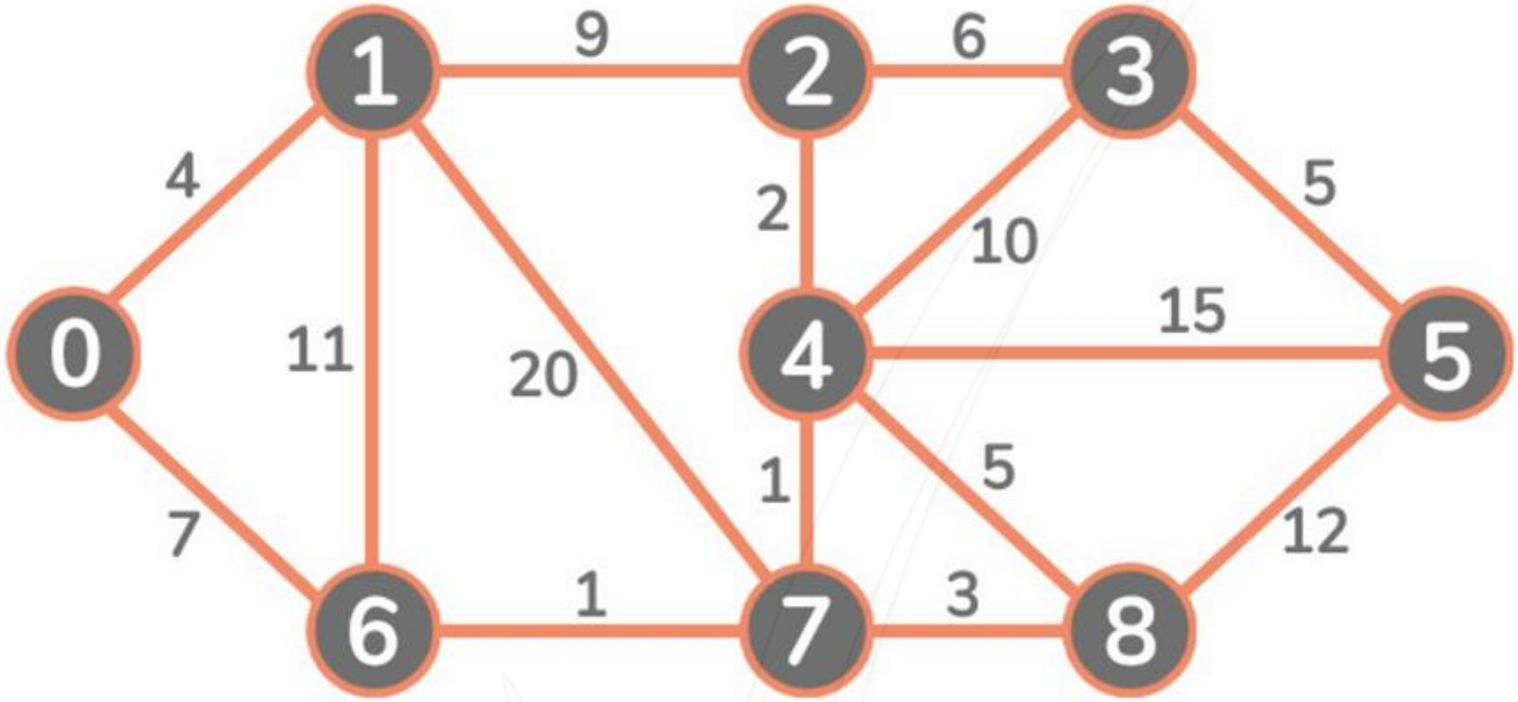
Moreover, not inserting all nodes in a graph makes it possible to extend the algorithm to find the shortest path from a single source to the closest of a set of target nodes on infinite graphs or those too large to represent in memory. The resulting algorithm is called uniform-cost search (UCS) in the artificial intelligence literature.

# DIJKSTAR ALGORITHM COMPLEXITY

**TIME COMPLEXITY: O(E LOG V)**
**WHERE, E IS THE NUMBER OF EDGES AND V IS THE NUMBER OF VERTICES.**
**SPACE COMPLEXITY: O(V)**



# IMPLEMENTATION




Let's say that Vertex 0 is our starting point. We are going to set the initial costs of vertices in this graph to infinity, except for the starting vertex:

We pick the vertex with a minimum cost - that is Vertex 0. We will mark it as visited and add it to our set of visited vertices. The starting node will always have the lowest cost so it will always be the first one to be added:

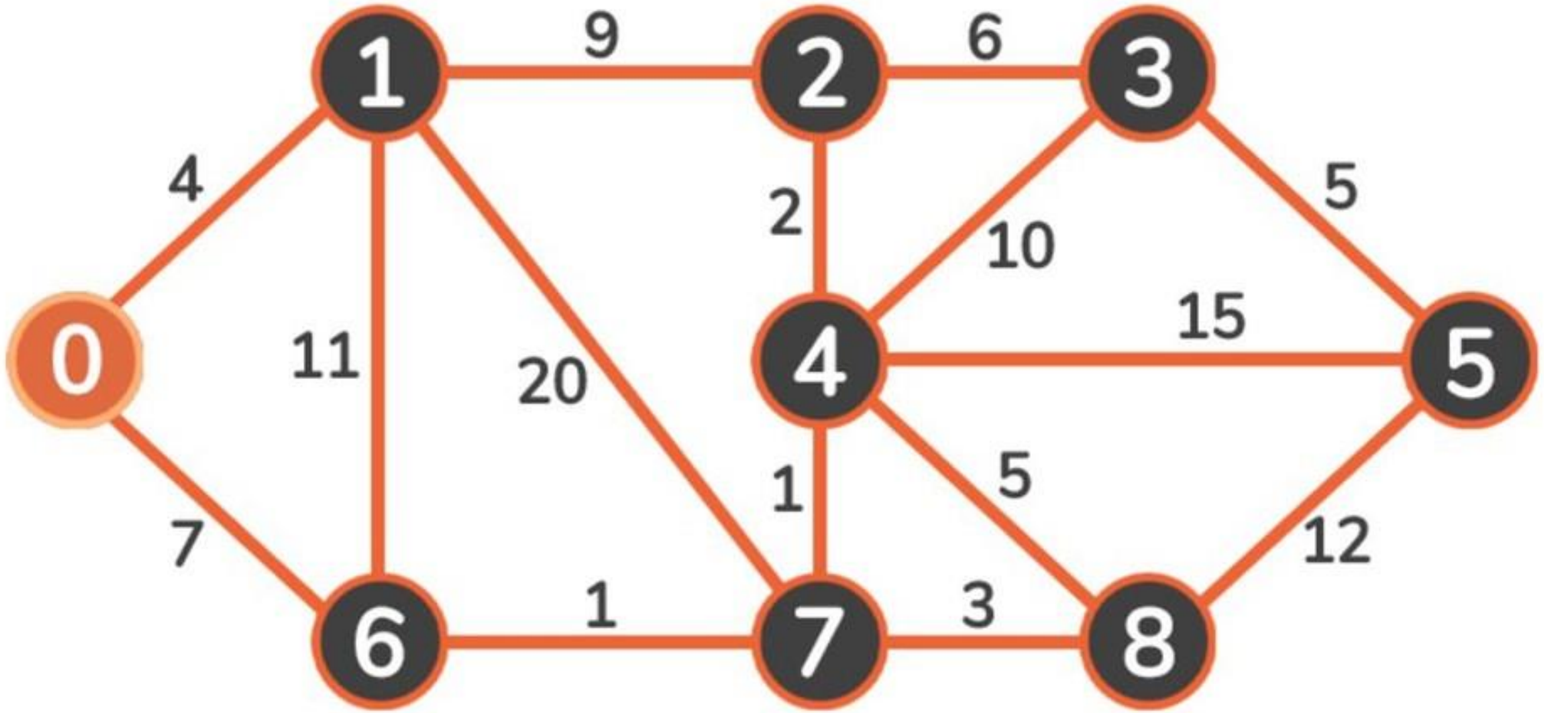
# VERTEX COST TO GET TO IT FROM VERTEX 0

| | |
|---|---|
| 0 | 0 |
| 1 | INF |
| 2 | INF |
| 3 | INF |
| 4 | INF |
| 5 | INF |
| 6 | INF |
| 7 | INF |
| 8 | INF |

STUDIO SHODWE

FACILITATOR: RICHARD SANCHEZ

MAY 02, 2025




Then, we will update the cost of adjacent vertices (1 and 6). Since 0 + 4 < infinity and 0 + 7 < infinity, we update the costs to these vertices:

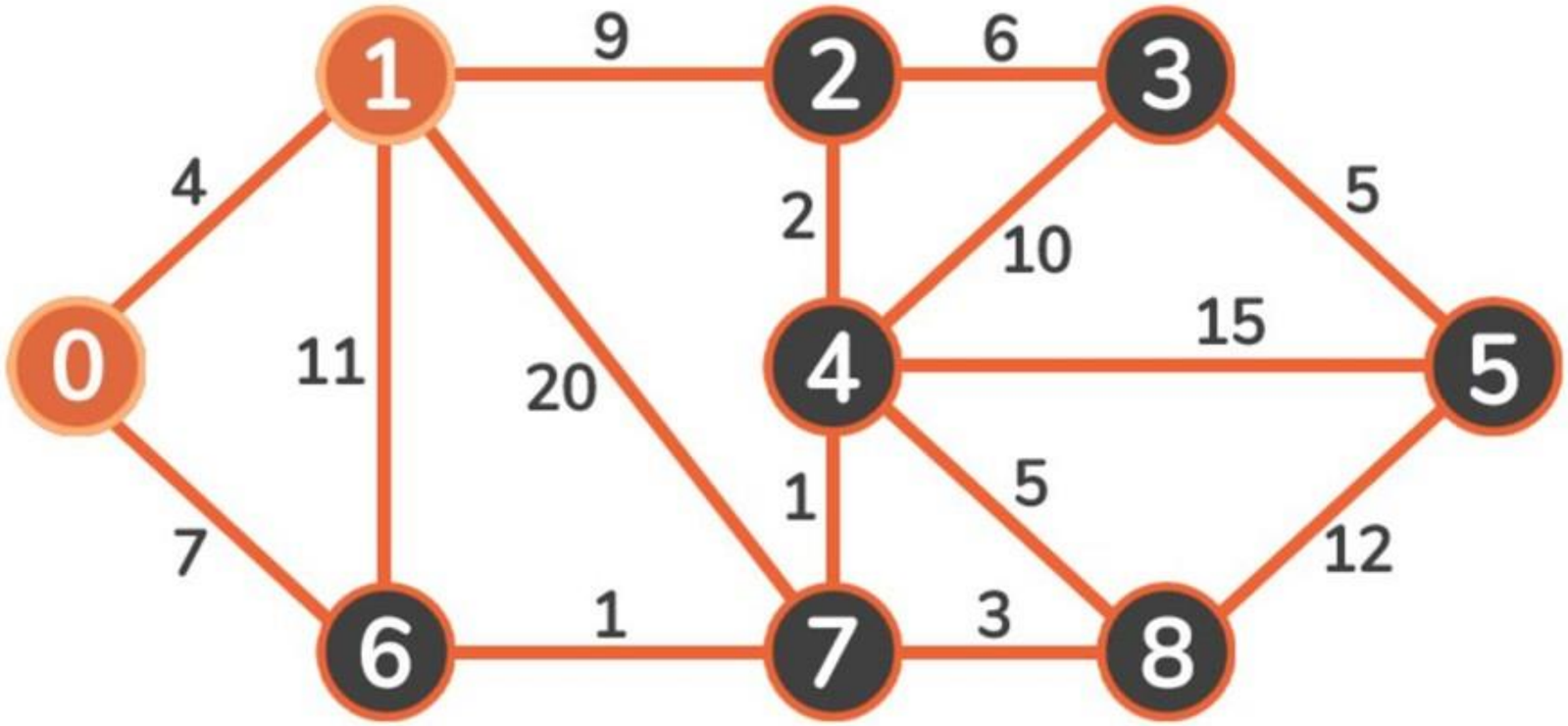
Now we visit the next smallest cost vertex. The weight of 4 is lower than 7 so we traverse to Vertex 1:

# VERTEX COST TO GET TO IT FROM VERTEX 0

| | |
|---|---|
| 0 | 0 |
| 1 | 4 |
| 2 | INF |
| 3 | INF |
| 4 | INF |
| 5 | INF |
| 6 | 7 |
| 7 | INF |
| 8 | INF |

DIJKSTRA ALGORITHM

JUNE 18



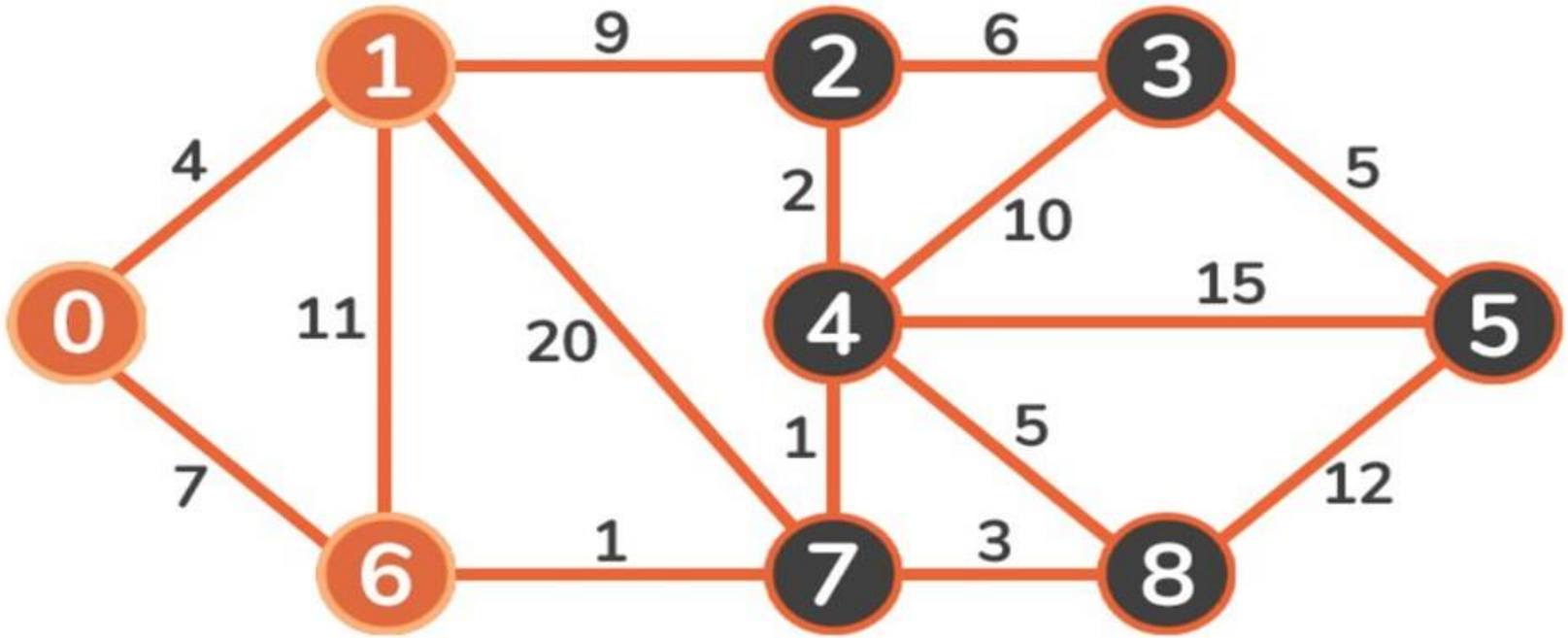

Upon traversing, we mark it as visited, and then observe and update the adjacent vertices: 2, 6, and 7:

Since 4 + 9 < infinity, new cost of vertex 2 will be 13
Since 4 + 11 > 7, the cost of vertex 6 will remain 7
Since 4 + 20 < infinity, new cost of vertex 7 will be 24
These are our new costs:

| VERTEX | COST TO GET TO IT FROM VERTEX 0 |
| --- | --- |
| 0 | 0 |
| 1 | 4 |
| 2 | 13 |
| 3 | INF |
| 4 | INF |
| 5 | INF |
| 6 | 7 |
| 7 | 24 |
| 8 | INF |

DIJKSTRA ALGORITHM

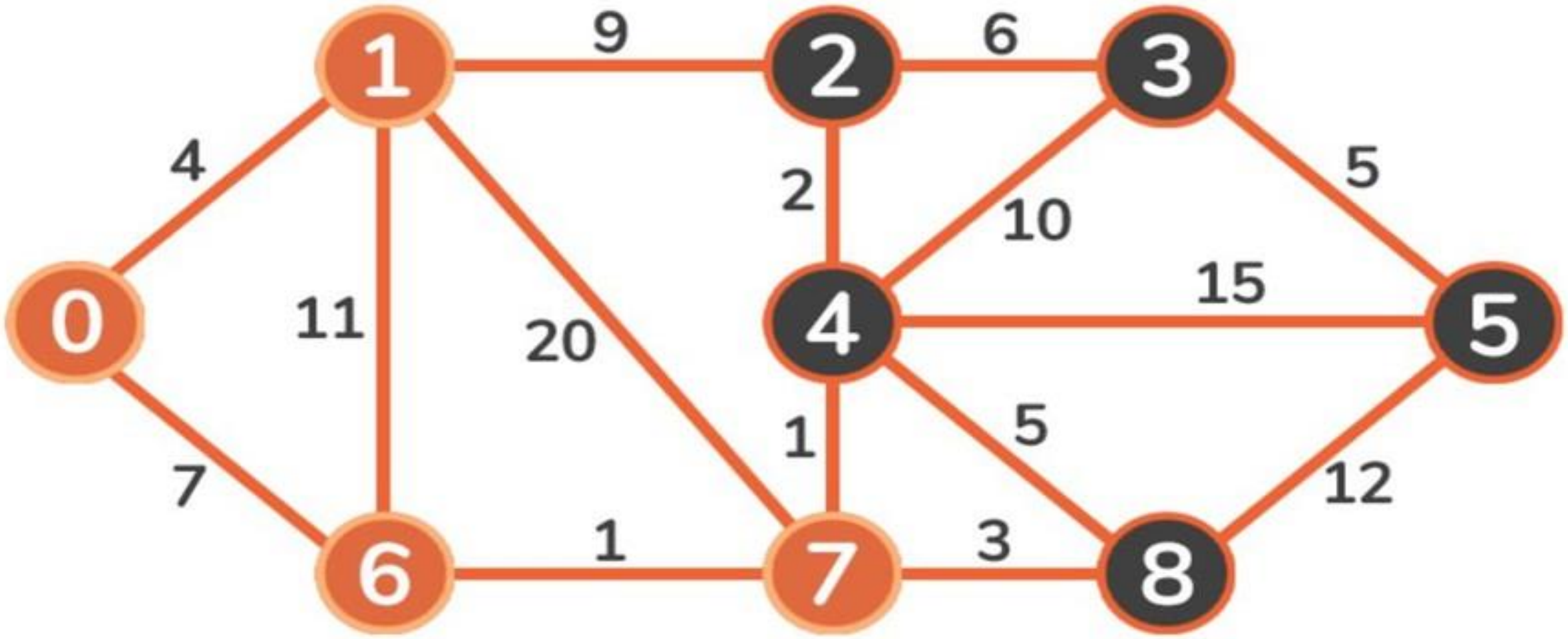
JUNE 18




The next vertex we're going to visit is Vertex 6. We mark it as visited and update its adjacent vertices' costs:

| VERTEX | COST TO GET TO IT FROM VERTEX 0 |
| --- | --- |
| 0 | 0 |
| 1 | 4 |
| 2 | 13 |
| 3 | INF |
| 4 | INF |
| 5 | INF |
| 6 | 7 |
| 7 | 8 |
| 8 | INF |

THE PROCESS IS CONTINUED TO VERTEX 7:

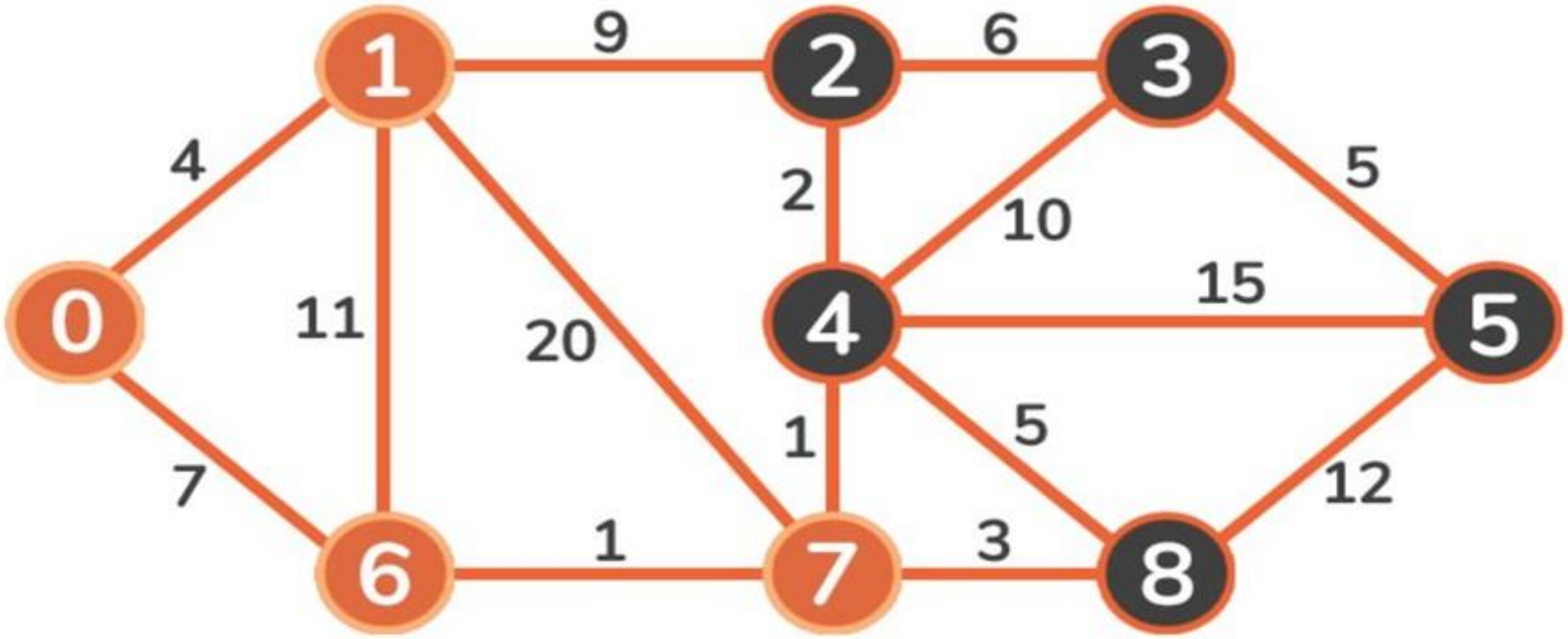



DIJKSTRA
ALGORITHM

JUNE 18

| VERTEX | COST TO GET TO IT FROM VERTEX 0 |
| --- | --- |
| 0 | 0 |
| 1 | 4 |
| 2 | 13 |
| 3 | INF |
| 4 | 9 |
| 5 | INF |
| 6 | 7 |
| 7 | 8 |
| 8 | 11 |

AND AGAIN, TO VERTEX 4:

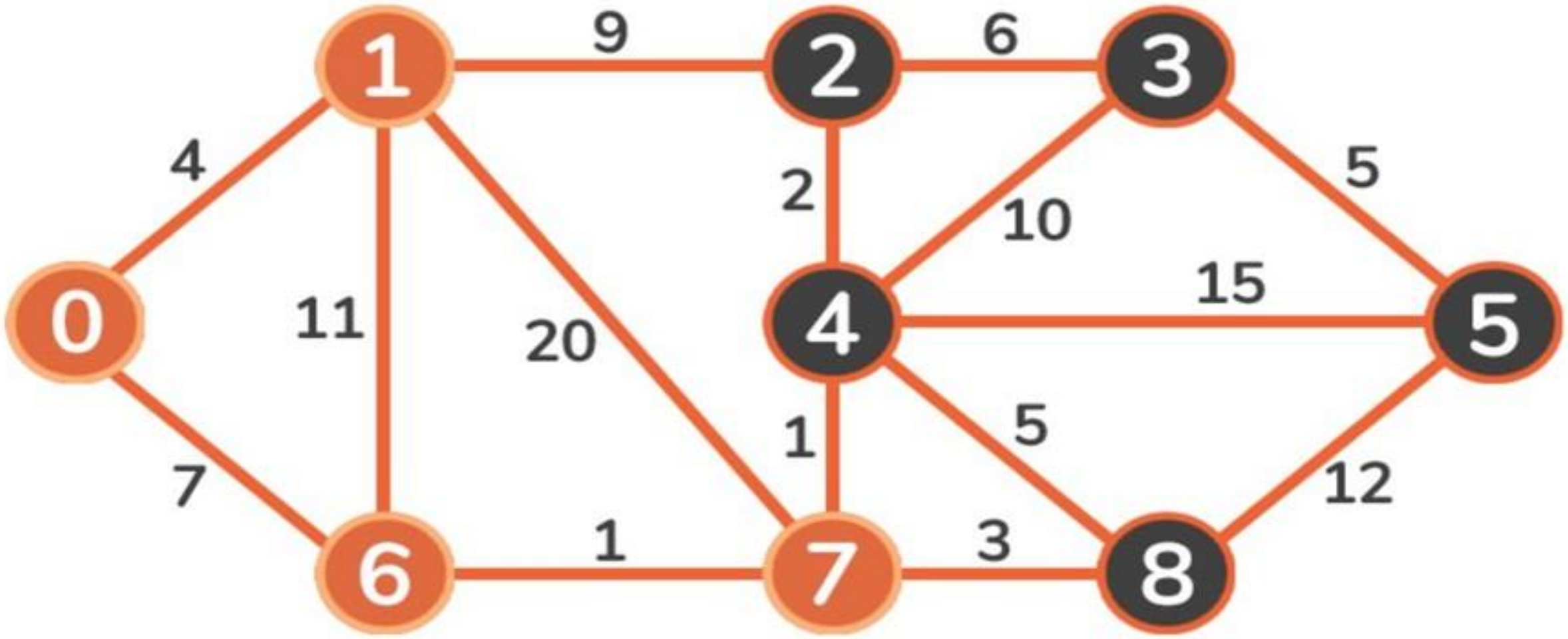



DIJKSTRA
ALGORITHM

JUNE 18

| VERTEX | COST TO GET TO IT FROM VERTEX 0 |
| --- | --- |
| 0 | 0 |
| 1 | 4 |
| 2 | 11 |
| 3 | 19 |
| 4 | 9 |
| 5 | 24 |
| 6 | 7 |
| 7 | 8 |
| 8 | 11 |

AND AGAIN, TO VERTEX 2:

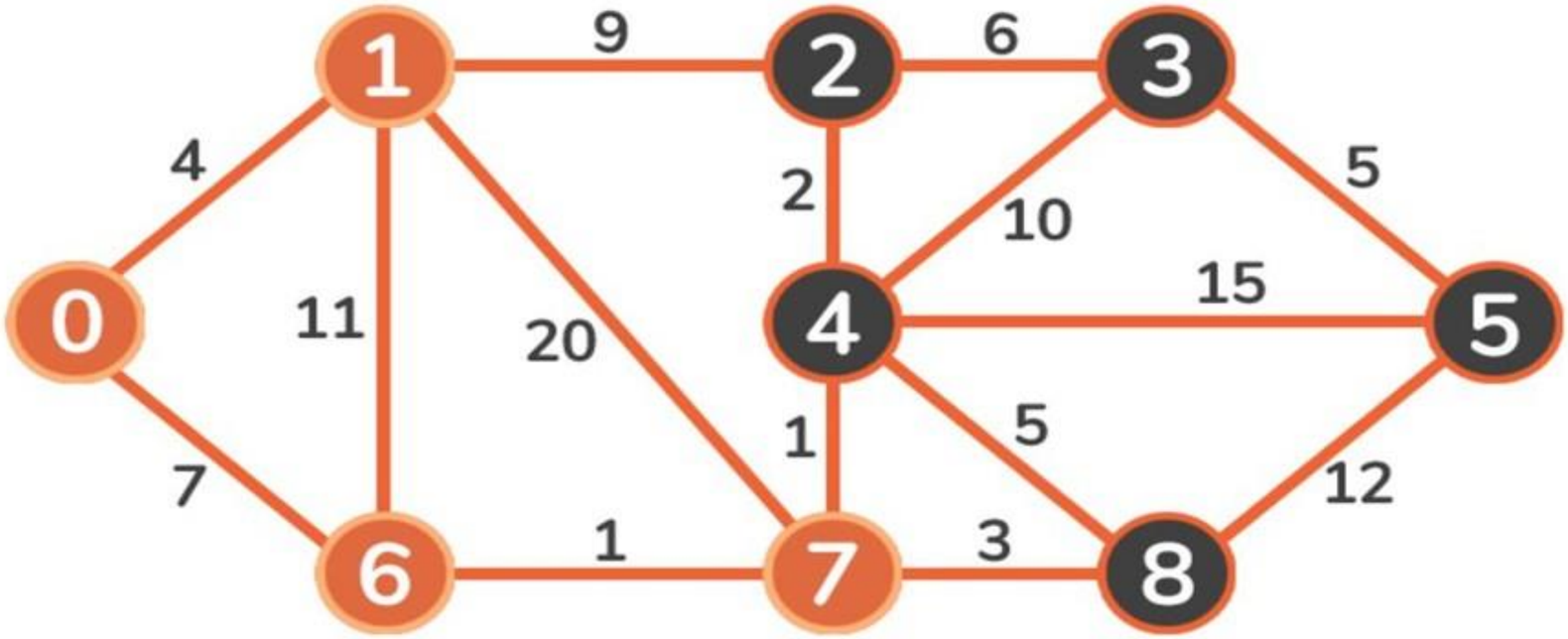



DIJKSTRA
ALGORITHM

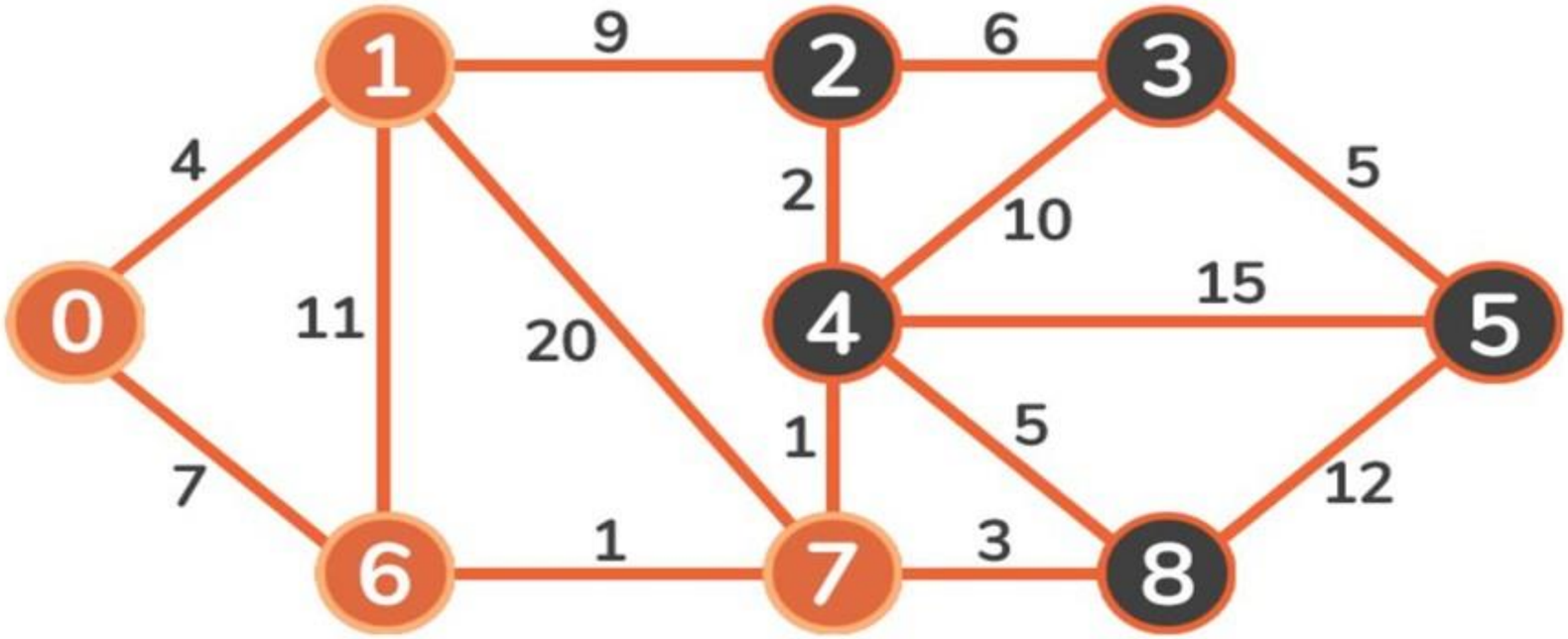
JUNE 18

| VERTEX | COST TO GET TO IT FROM VERTEX 0 |
| --- | --- |
| 0 | 0 |
| 1 | 4 |
| 2 | 11 |
| 3 | 19 |
| 4 | 9 |
| 5 | 24 |
| 6 | 7 |
| 7 | 8 |
| 8 | 11 |

AND AGAIN, TO VERTEX 2:



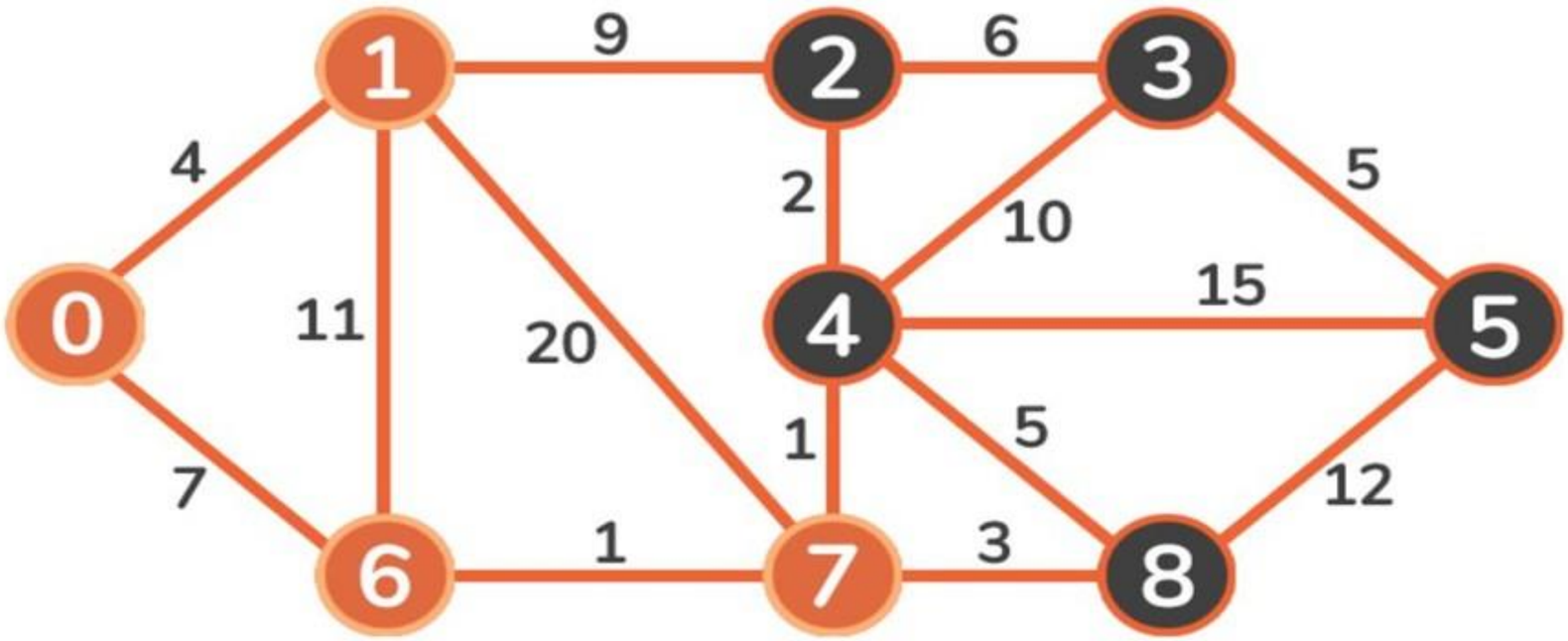

The only vertex we're going to consider is Vertex 3. Since 11 + 6 < 19, the cost of vertex 3 is updated.
Then, we proceed to Vertex 8:




Finally, we're updating the Vertex 5:

| VERTEX | COST TO GET TO IT FROM VERTEX 0 |
| --- | --- |
| 0 | 0 |
| 1 | 4 |
| 2 | 11 |
| 3 | 17 |
| 4 | 9 |
| 5 | 24 |
| 6 | 7 |
| 7 | 8 |
| 8 | 11 |

WE'VE UPDATED THE VERTICES IN THE LOOP-LIKE STRUCTURE IN THE END - SO NOW WE JUST HAVE TO TRAVERSE IT - FIRST TO VERTEX 3:




There are no more unvisited vertices that may need an update. Our final costs represents the shortest paths from node 0 to every other node in the graph:

| VERTEX | COST TO GET TO IT FROM VERTEX 0 |
| --- | --- |
| 0 | 0 |
| 1 | 4 |
| 2 | 11 |
| 3 | 17 |
| 4 | 9 |
| 5 | 24 |
| 6 | 7 |
| 7 | 8 |
| 8 | 11 |

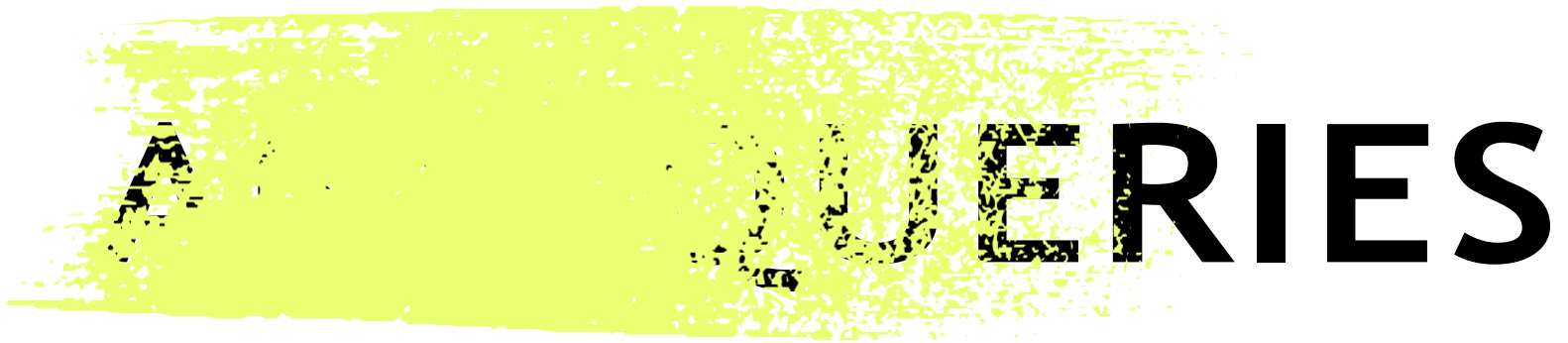
# CODE SNIPPET

```python
from queue import PriorityQueue

class Graph:
    def __init__(self, num_of_vertices):
        self.v = num_of_vertices
        self.edges = [[-1 for i in range(num_of_vertices)] for j in range(num_of_vertices)]
        self.visited = []

    def add_edge(self, u, v, weight):
        self.edges[u][v] = weight
        self.edges[v][u] = weight


def dijkstra(graph, start_vertex):
        D = {v:float('inf') for v in range(graph.v)}
        D[start_vertex] = 0

        pq = PriorityQueue()
        pq.put((0, start_vertex))

        while not pq.empty():
            (dist, current_vertex) = pq.get()
            graph.visited.append(current_vertex)

            for neighbor in range(graph.v):
                if graph.edges[current_vertex][neighbor] != -1:
                    distance = graph.edges[current_vertex][neighbor]
                    if neighbor not in graph.visited:
                        old_cost = D[neighbor]
                        new_cost = D[current_vertex] + distance
                        if new_cost < old_cost:
                            pq.put((new_cost, neighbor))
                            D[neighbor] = new_cost
        return D
```

```
g = Graph(9)
g.add_edge(0, 1, 4)
g.add_edge(0, 6, 7)
g.add_edge(1, 6, 11)
g.add_edge(1, 7, 20)
g.add_edge(1, 2, 9)
g.add_edge(2, 3, 6)
g.add_edge(2, 4, 2)
g.add_edge(3, 4, 10)
g.add_edge(3, 5, 5)
g.add_edge(4, 5, 15)
g.add_edge(4, 7, 1)
g.add_edge(4, 8, 5)
g.add_edge(5, 8, 12)
g.add_edge(6, 7, 1)
g.add_edge(7, 8, 3)

D = dijkstra(g, 0)

print(D)

for vertex in range(len(D)):
    print("Distance from vertex 0 to vertex", vertex, "is", D[vertex])
```

```
from queue import PriorityQueue ···
```

```
{0: 0, 1: 4, 2: 11, 3: 17, 4: 9, 5: 22, 6: 7, 7: 8, 8: 11}
Distance from vertex 0 to vertex 0 is 0
Distance from vertex 0 to vertex 1 is 4
Distance from vertex 0 to vertex 2 is 11
Distance from vertex 0 to vertex 3 is 17
Distance from vertex 0 to vertex 4 is 9
Distance from vertex 0 to vertex 5 is 22
Distance from vertex 0 to vertex 6 is 7
Distance from vertex 0 to vertex 7 is 8
Distance from vertex 0 to vertex 8 is 11
```




DIJKSTRA
ALGORITHM

JUNE 18

# THANK YOU