FINDING THE SHORTEST PATH

A MINOR PROJECT REPORT

Submitted by

Saurabh Pandey (RA2011003010207)
Vinay Sonkusale (RA2011003010167)
Gauri Tripathi (RA2011003010144)

Faculty Name: Dr. C. Muralidharan

BACHELOR OF TECHNOLOGY
in

COMPUTER SCIENCE AND ENGINEERING
of

FACULTY OF ENGINEERING AND TECHNOLOGY

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

S.R.M. Nagar, Kattankulathur, Kancheepuram District

DECLARATION

I, hereby declare that the work presented in this dissertation entitled "SHORTEST SAFE ROUTE IN A
PATH WITH LANDMINES" has been done by me and my team, and this dissertation embodies my own work.

Approved By:
Dr. C. Muralidharan

ACKNOWLEDGEMENTS

I would like to thank **Dr. C. Muralidharan (Assistant Prof.)** who has been a great inspiration and who have provided sufficient background knowledge and understanding of this subject.
Our humble prostration goes to her, for providing all the necessary resources and environment, which have aided me to complete this project successfully.

CONTRIBUTION TABLE

NAME	CONTRIBUTION
Gauri Tripathi	Developing algorithm, prepared and presented the ppt
Saurabh Pandey	Developing the algorithm, prepared the report
Vinay Sonkusale	Developing the algorithm, analysis of the algorithm, ideated the application of algorithm

TABLE OF CONTEXT

Shortest Path Finding Between Nodes

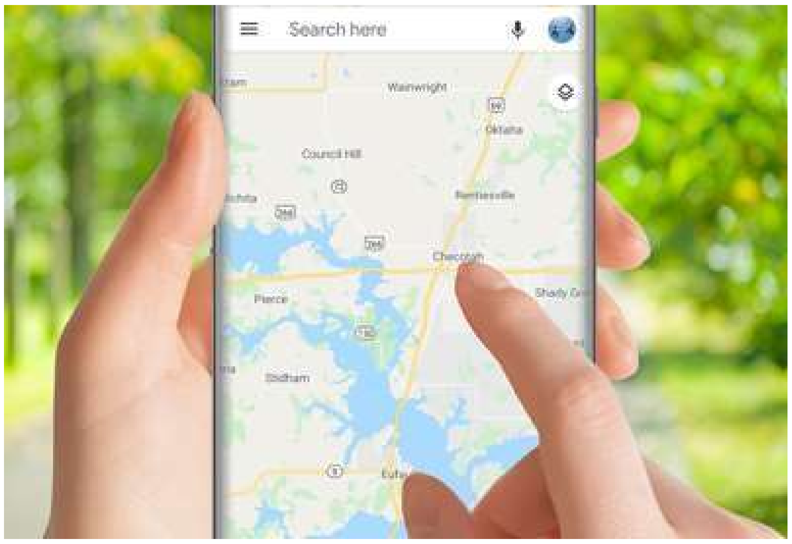
1. *Problem Definition:*

In the present day scenario when the roadways and all sort of means of commute and drastically evolving. There is now a need larger than ever for computer-aided navigation and intelligence. With our intention and ideation, we aim to counter this problem with help of a heuristic graph-dependent backtracking algorithm known as DIJKSTRA. Dijkstra is the brainchild of sir EDEFUGEUIFUU this algorithm aided with new technical advancements help. Make day-to-day commute an easier and more convenient task. While taking care of accuracy and other intricate details.

2. *Problem Explanation with diagram and example:*

Have you ever used Google Maps to find the shortest route or minimum cost of gas to get from one point to another? If so, then you've encountered an example of the shortest path problem. In math terms, this is a way to find the shortest possible distance between two vertices on a graph.

Suppose we're trying to find the shortest path from your house to Divya's house. We know the distances between various locations throughout town. If we let various locations be vertices and the routes between them be edges, we can create a weighted graph representing the situation. Quick definition: a weighted graph is a collection of vertices and edges with edges having a numerical value (or weight) associated with them.



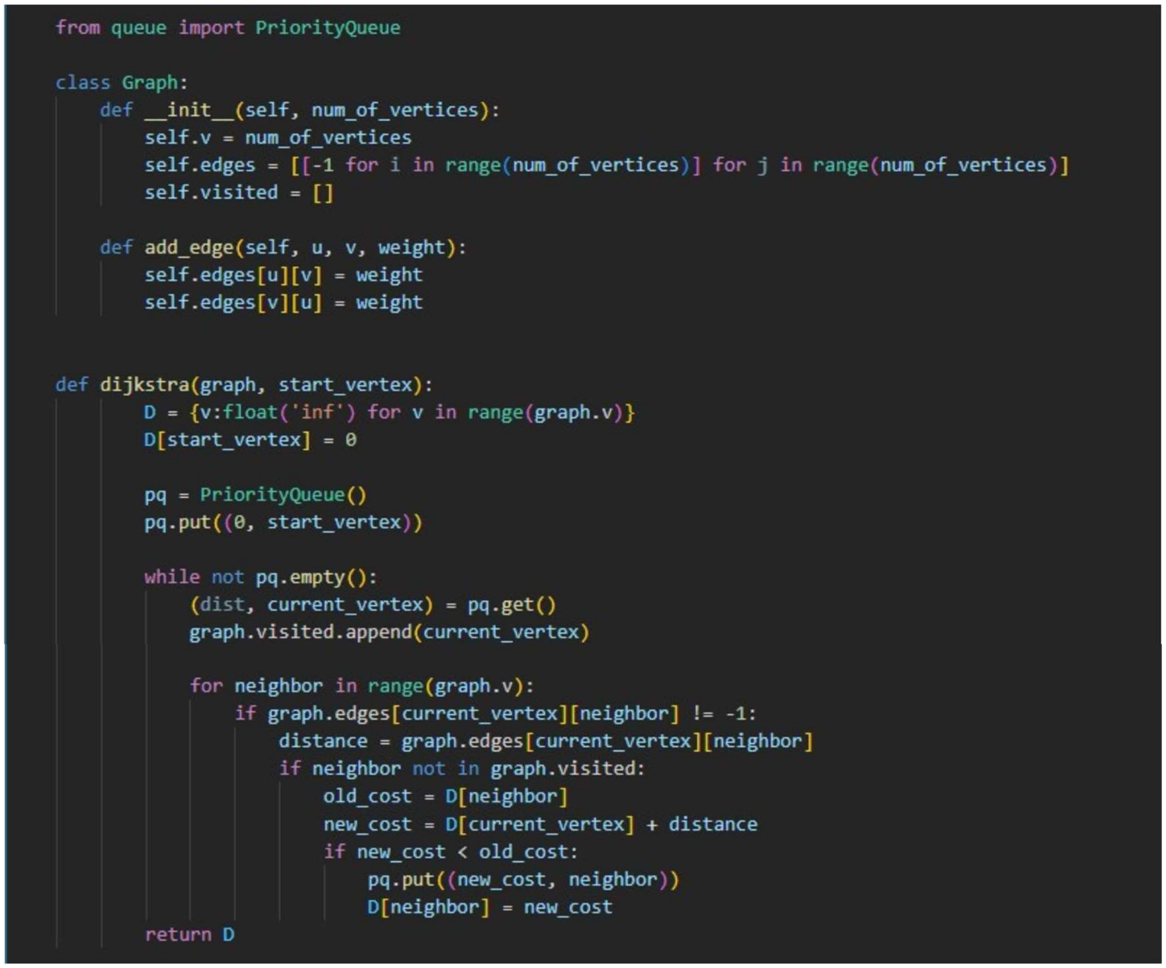
3. Design Techniques used:

Backtracking is a general algorithmic technique that considers searching every possible combination in order to solve an optimization problem. The basic principle of a backtracking algorithm, in regards to Sudoku, is to work forwards, one square at a time to produce a working Sudoku grid. When a problem occurs, the algorithm takes itself back one step and tries a different path. It's nearly impossible to produce a valid Sudoku by randomly plotting numbers and trying to make them fit. Likewise, backtracking with a random placement method is equally ineffective. Backtracking best works in a linear method. It is fast, effective if done correctly.

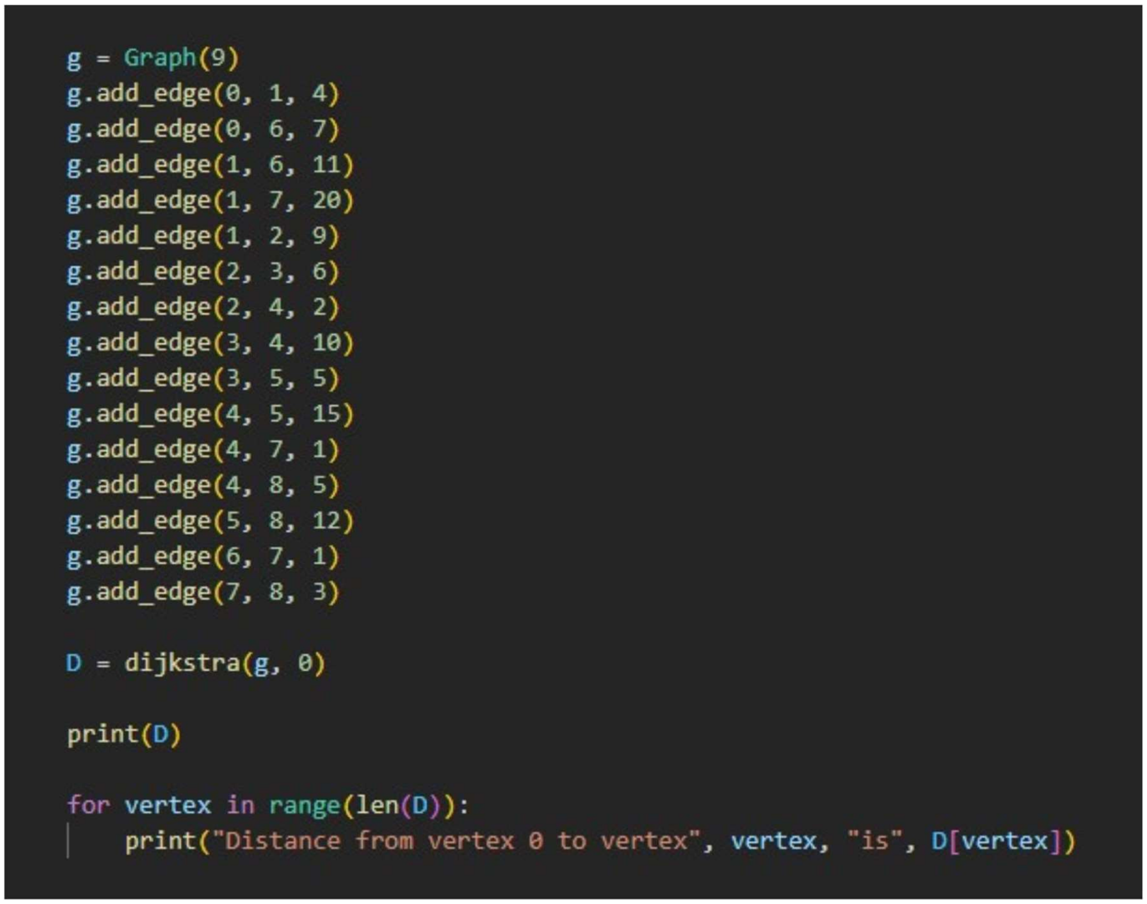
4. Algorithm and its explanation for the problem:

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 ofa 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. 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.

5. Code and Output:

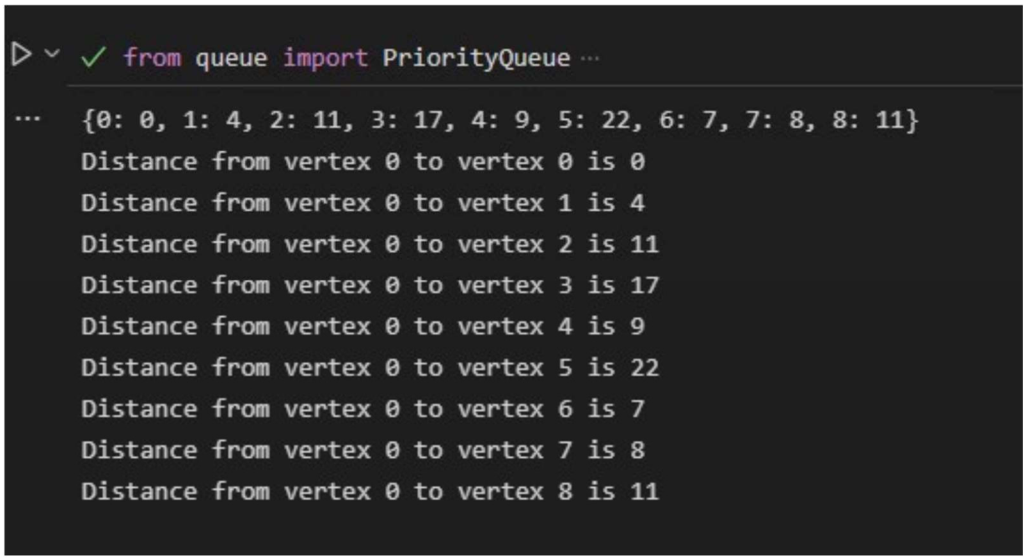


```python
from queue import PriorityQueue

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
```



```python
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])
```

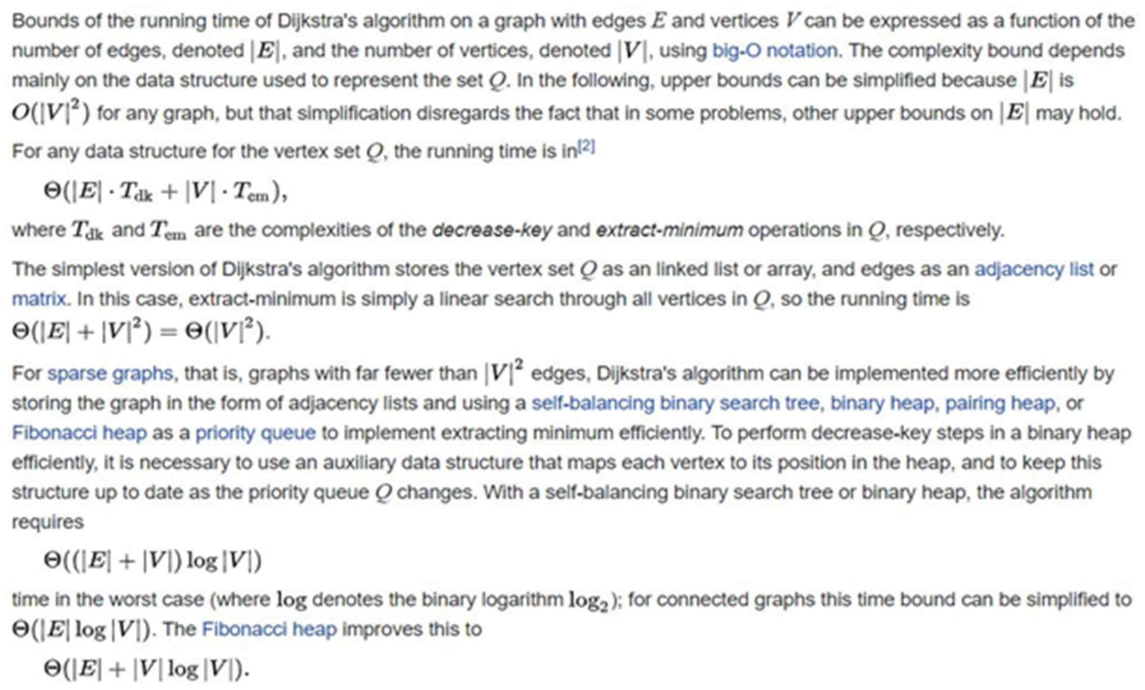
OUTPUT:



```
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
```

6. Complexity analysis:

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



Bounds of the running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of the number of edges, denoted $|E|$, and the number of vertices, denoted $|V|$, using big-O notation. The complexity bound depends mainly on the data structure used to represent the set Q. In the following, upper bounds can be simplified because $|E|$ is $O(|V|^2)$ for any graph, but that simplification disregards the fact that in some problems, other upper bounds on $|E|$ may hold.

For any data structure for the vertex set Q, the running time is in[2]

$$\Theta(|E| \cdot T_{\mathrm{dk}} + |V| \cdot T_{\mathrm{em}}),$$

where T_{dk} and T_{em} are the complexities of the *decrease-key* and *extract-minimum* operations in Q, respectively.

The simplest version of Dijkstra's algorithm stores the vertex set Q as an linked list or array, and edges as an adjacency list or matrix. In this case, extract-minimum is simply a linear search through all vertices in Q, so the running time is $\Theta(|E| + |V|^2) = \Theta(|V|^2)$.

For sparse graphs, that is, graphs with far fewer than $|V|^2$ edges, Dijkstra's algorithm can be implemented more efficiently by storing the graph in the form of adjacency lists and using a self-balancing binary search tree, binary heap, pairing heap, or Fibonacci heap as a priority queue to implement extracting minimum efficiently. To perform decrease-key steps in a binary heap efficiently, it is necessary to use an auxiliary data structure that maps each vertex to its position in the heap, and to keep this structure up to date as the priority queue Q changes. With a self-balancing binary search tree or binary heap, the algorithm requires

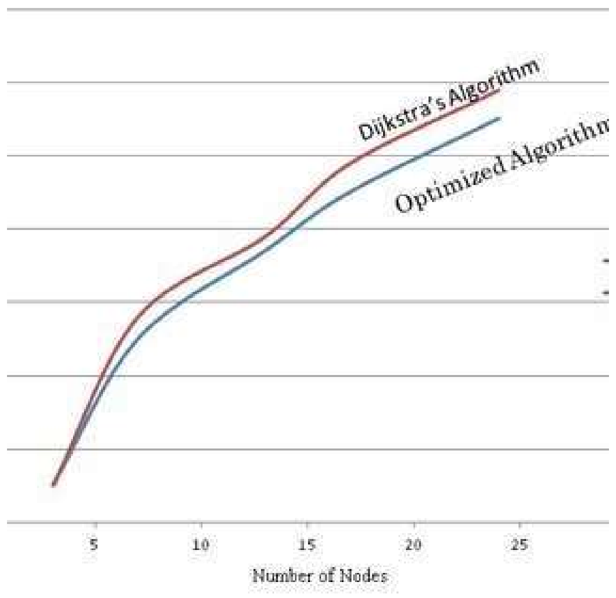
$$\Theta((|E| + |V|) \log |V|)$$

time in the worst case (where $\log$ denotes the binary logarithm $\log_2$); for connected graphs this time bound can be simplified to $\Theta(|E| \log |V|)$. The Fibonacci heap improves this to

$$\Theta(|E| + |V| \log |V|).$$

7. Conclusion:

We have successfully devised a solution which can identify modern scenario parameters and jot down a consice accurate and shortest path to commute from index a to b using the technical aid of Dijkstra algorithm. Akshay, Aryan and Sreevatsan were on their way back home and they traced back their way using Dijkstra algorithm.



8. References

Wikipedia - https://www.wikipedia.org/
Britannica - https://www.britannica.com/
Computerphile- https://blog.bachi.net/?p=10501