Investigating the effect on performance using a data structure in graph algorithms

How does the use of priority queue affect the performance of Dijkstra and Prim Algorithms?

A Computer Science Research Paper

3300 Words

Table of Contents

1. Introduction

Graph theory is the study of interrelationships between nodes and vertices. Such relationships help tackle many areas of sciences and others, such as statistical mechanics, electrical engineering, operations research, communication networks etc. [1]

Graph theory, holistically, deals with an array of subtopics that, as described above, have many applications. One of the most interesting problems from this topic is the travelling salesman problem, which has a wide variety of applications such as scan chain optimisation, DNA universal strings etc. [1]

Such theories help solve real-life problems computationally. However, achieving the solution in a specific set of constraints and/or limited resources is challenging. Therefore, algorithmic analysis is used, which is a method by which the computational complexity of algorithms is found [2]. Here, computational complexity refers to classifying algorithms based on their efficiency [3]. Conducting algorithmic analysis on algorithms related to graph theory makes it extremely efficient and thus allowing it to solve more problems with fewer resources.

Graph theory is considered an extremely vital aspect in many areas, specifically communications networks and the internet [1]. The algorithms primarily used for the investigation have a wide variety of uses in different fields. Dijkstra's algorithm is used in many areas, ranging from digital Mapping services in google maps to social networking applications and aviation[4]. Prim's algorithm is primarily applied for Network for roads, Cluster analysis, Game development, Cognitive science, Irrigation channels etc. [5]

This paper essentially seeks to investigate whether a data structure like the priority queue could help in making two of the most commonly used algorithms more efficient.

This research would be extremely helpful in many areas mentioned above, as the use of these algorithms would come with a cost of resources such as time, space, money etc. Many different data structures could be possibly used in both these algorithms. However, identifying the best one seems time taking and resource-intensive. Therefore, with help of this investigation could make it easy to identify whether this is the right one or not. Even though there are some areas where the use of such data structure can be done to the algorithms, they have never been experimentally proved. This paper would be one of the sources of proof for such cases.

To investigate the improvement of analysis, the algorithms were first implemented with multiple random inputs without and with the priority queue, where the input in both cases remains the same. The time taken to implement would be recorded and accordingly, the trends of these recordings were discussed

2. Theoretical Background

2.1 Basics of Graph Theory

The concepts explained below is quintessential in answering the research question as the algorithms handle graphs with the below properties.

Vertices and Edges

It is a set of points represented as V in the pair (V, E). Edges are considered as the lines that join the vertices, which are represented as E in the pair (V, E).

Weighted Graph

A graph wherein the edges are without any value. This means that the vertices essentially are the ones with a numerical value in the graph.

Undirected Graph

Graphs for which the edges are not directed to a particular node/edge. They are essentially bidirectional in nature. Dijkstra and Prim's algorithms are mainly used in these graphs for finding the shortest path and minimum spanning tree.

Big O Notation

The notation helps in analysing the algorithm(s) mathematically and give an abstract expression to show its behaviour with an increasing number of inputs/nodes and provides the worst-case notation of the algorithm, thereby allowing to understand the maximum time it can take for a set of inputs. For the benefit of validating the experiment and analysing the results given, the notation will be used for Dijkstra and Prim algorithm. To give an example, a function of an algorithm $f(n)=m(n)+c$ is represented as $O(n)$. The constant which are part of the function is not added as they don't put a considerable change in the overall time.

2.2 Priority Queue

A priority queue is used for storing a weighted directed graph based on priorities i.e. as per their importance. Depending upon this importance, the element is accordingly removed [13]. It can be represented using different data structures, ranging from linked lists to heaps. Because of its characteristic of prioritising elements and putting them in order in that way, the addition or removing of elements becomes efficient.

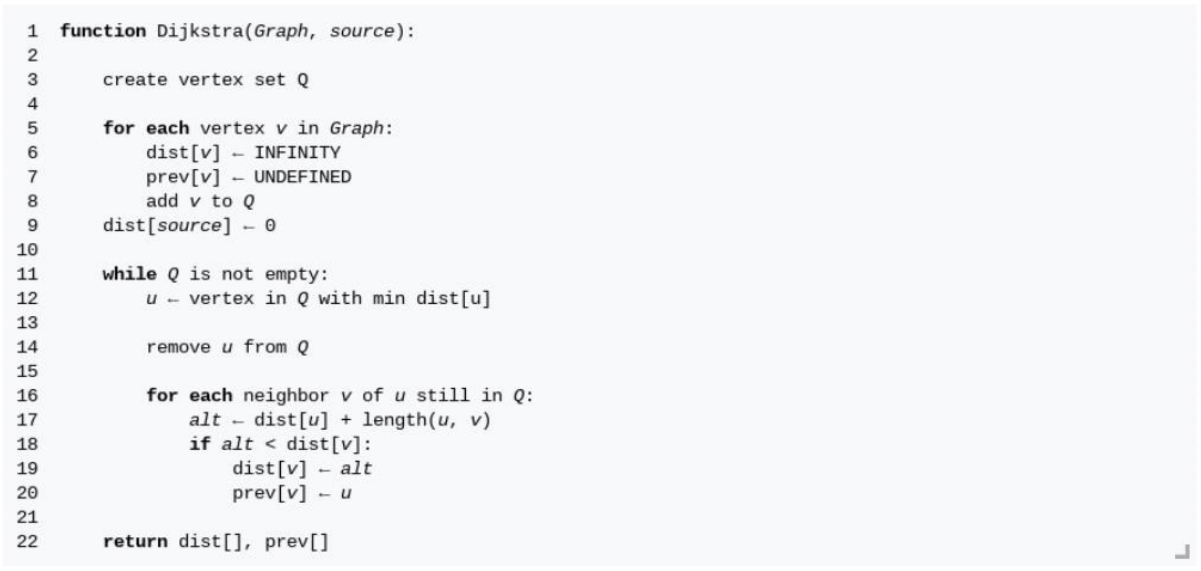
2.3 Dijkstra's Algorithm

In the field of graph theory, Dijkstra's algorithm is an algorithm used for finding the shortest path between nodes in a graph, from the source of the graph to any vertices of the graph. From any point of the graph, the path with the most minimum cost is determined from the source of the graph. The graph is essentially meant for weighted graph G = (V, E) graphs with edges having a particular count or cost. Here, all the edges are considered non-negative [8].

Every node that is part of the graph apart from the source would be considered us *unvisited node*. All the unvisited nodes, together are added to an unvisited set. All the nodes are assigned with a tentative distance value to either 0 or infinity, where 0 is meant for the initial node since the distance from the node to itself would be 0. The rest are considered as infinite which will later be changed since they haven't been part of the path. The initial or the source node would be set as the current node. From the current node, the neighbouring unvisited nodes are considered by greedily calculating the distance it takes to reach the node and comparing it with the current distance. If it turns to be smaller, then the current node would be the one with the smallest tentative distance. After considering all the possible unvisited nodes, the current node is considered as visited, which is then removed from the unvisited set. Such a process of greedily finding the shortest path is continued endlessly until there are two possible situations, which forces the algorithm to stop k: [11].

1. If the destination node has been marked visited
2. The tentative node between the nodes is infinity(only possible when there is no possible connection [9]

Pseudocode of Dijkstra’s Algorithm



```
 1  function Dijkstra(Graph, source):
 2
 3      create vertex set Q
 4
 5      for each vertex v in Graph:
 6          dist[v] ← INFINITY
 7          prev[v] ← UNDEFINED
 8          add v to Q
 9      dist[source] ← 0
10
11      while Q is not empty:
12          u ← vertex in Q with min dist[u]
13
14          remove u from Q
15
16          for each neighbor v of u still in Q:
17              alt ← dist[u] + length(u, v)
18              if alt < dist[v]:
19                  dist[v] ← alt
20                  prev[v] ← u
21
22      return dist[], prev[]
```

Figure 1Pseudocode of Dijkstra's Algorithm

The pseudocode is broken down into three loops, where each loop is working on one of the sub-parts of the algorithm. This section will be covering each loop, explaining how they individually achieve Dijkstra’s algorithm **for each in vertex v in Graph from line 5**

This loop essentially names each vertex as infinity and the source node is undefined. This is the implementation of steps 1 and 2 described in the previous section. The arrays are tools to record the distance possible from the source node. In addition, line 9 is the adding all the vertices to the set Q

while Q is not empty

The loop is hovering through the set and checking the ones that are **not** in the minimum distance. These vertices are being removed from the set Q. Line 12 and 14 are essentially doing it. Step 3 is being executed using this loop. The loops take place till the set Q is not empty.

for each neighbour v of u still in Q

This loop essentially does the rest of the steps described as the process. The loop explores all the potential neighbours of u which are part of the set Q. It then measures the distance it takes to move the neighbours by calculating the sum as shown in line 17, where distance and the length of the edges are being added. Now, the summation is compared using the **if** construct in line number 18. Line 19 and 20 ensure that the best possible node to move is added or marked as visited.

Miscellaneous code

dist[source] -> 0 is the idea of calling the distance from the source code to the source code is 0. ‘**return dist[], prev[]**’ returns the distance from the source node to the end node. prev[] returns the nodes from the shortest path can be taken.

<u>Example of using the algorithm</u>

Consider the following weighted undirected graph shown in Figure 2

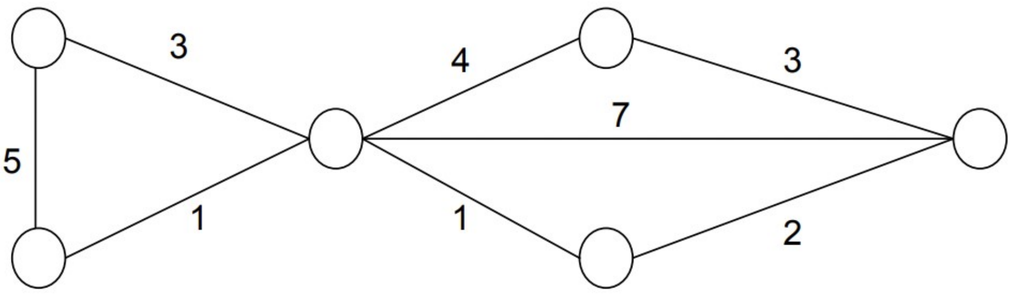


Figure 2 An example of a weighted undirected graph

In this figure, consider the first node connected to edges 3 and 5 as the current node. Except for that node, the rest are considered as infinite as there hasn't been a path decided. Figure 2 accurately shows this.

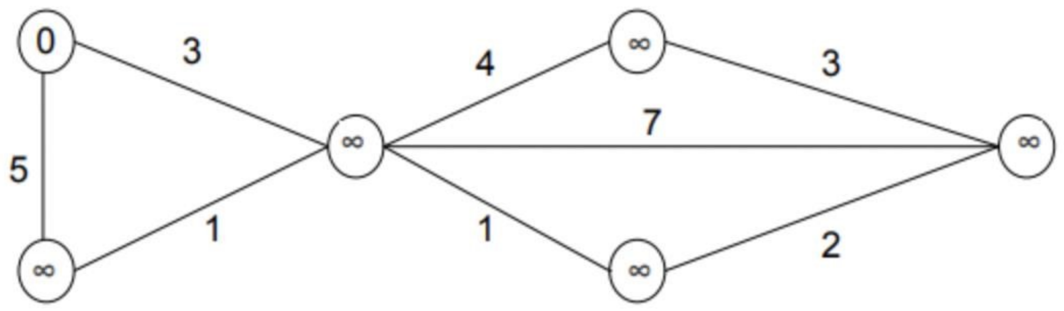


Figure 3 Graph with vertices named infinity and a source node of 0.

0 is considered as the current node and the rest as the unvisited node part of an unvisited set. From 0, each neighbouring node with a tentative distance is calculated and compared against the current one, to greedily find the shortest path. Figure 3 elaborates this further.

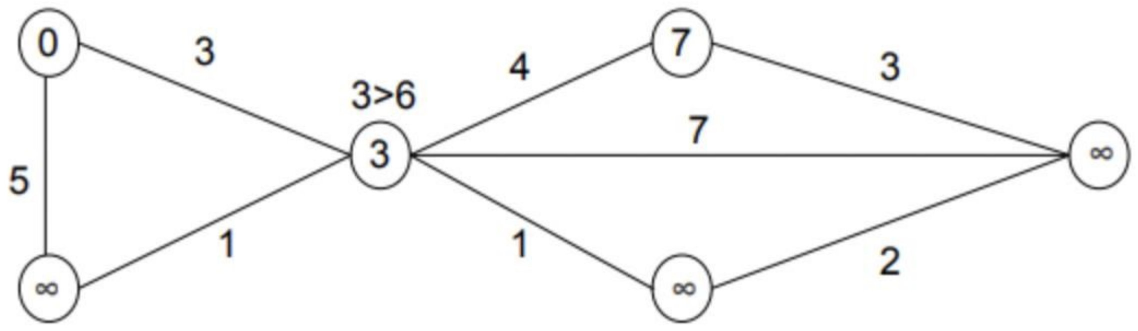


Figure 4 Updated graph with number vertices indicating the shortest path possible to each of them

Here, the comparison between the distance 3>6 shows how the algorithm traverses efficiently keeping the shortest edge in mind. From node 3, the neighbouring nodes are considered and are numbered based on the tentative distance. However, if the tentative distance of the adjacent vertex is lesser than the new tentative vertex, it mustn't be updated. It is seen in Figure 3.

Finally, Figure 4 shows the shortest distance from source 0 to the destination node with a tentative distance of 6 after considering all the possible vertices to traverse.

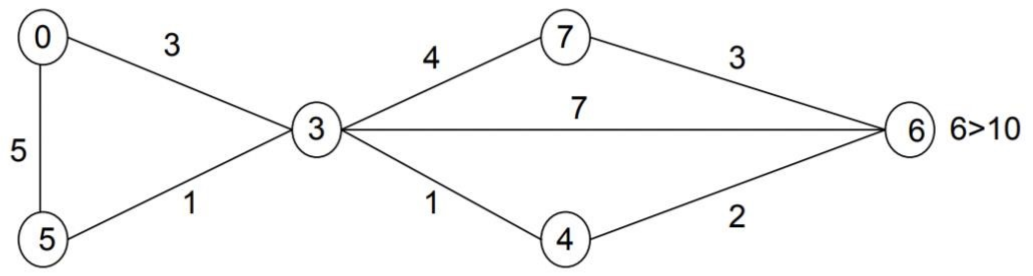


Figure 5 Final graph showing the shortest path possible to the destination from the source 0

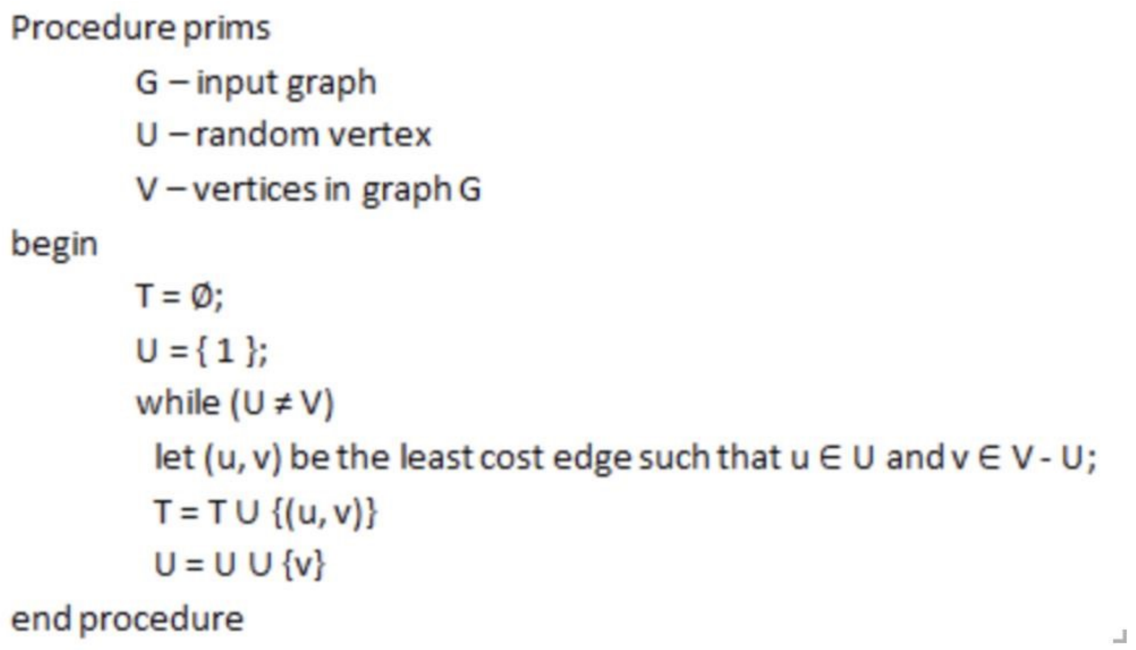
By using the priority queue in Dijkstra's algorithm, the very idea of prioritising the vertex and edge with the least tentative distance could allow for its growth in becoming much more efficient than the classical way, using adjacency matrix. In other words, the priority queue helps in finding the best first search, where rather than focusing on the depth or the breadth of traversing but finding the best or the most optimal solution.

2.4 Prim's Algorithm

Prim's algorithm is a minimum spanning tree algorithm which focuses on finding the subsets of the edges of that graph which from a tree that includes every vertex and that has the minimum sum of weights among all the trees that can be formed from the graph [9]. Just like Dijkstra's algorithm, Prim's algorithm focuses on weighted undirected graphs. [10]. In essence, the algorithm finds the minimum spanning forest in a disconnected graph [10]. Here, the minimum spanning tree is the subset of edges of a connected, edge-weighted undirected graph that connects all the vertices with the minimum weight possible.

Compared to Dijkstra's algorithm, Prim's algorithm functionality is rather a simple one and uses the greedy approach. A randomly chosen vertex is initialised, which is made like a tree. This tree is grown with one edge which is connected to vertices that are not part of the tree, and by whom the minimum-weight edge is found and accordingly transferred to the tree. The above process is repeated extensively until all the vertices are part of the tree. [10]

Pseudocode



```
Procedure prims
        G – input graph
        U – random vertex
        V – vertices in graph G
begin
        T = Ø;
        U = { 1 };
        while (U ≠ V)
         let (u, v) be the least cost edge such that u ∈ U and v ∈ V - U;
         T = T U {(u, v)}
         U = U U {v}
end procedure
```

Figure 6 Pseudocode of Prim's Algorithm. Adapted from [15]

Here, three variables have been introduced: G, U and V for graph and vertices where G is the graph on which MST(minimum spanning tree) is found. U is used for choosing the initial vertex that helps start the tree, as explained above. U is first given with a particular value to initialise and introduced to a loop with V to allow the tree to grow with edges. The line "let (*u,v*) be the least cost ..... V-U; " is used for identifying the minimum value based vertex to add in the tree. Finally, T and U have been collected with the tree with the least values possible.

<u>**Example of using the algorithm**</u>

To illustrate the above concept, consider the following example of an unweighted undirected graph shown in Figure 7.

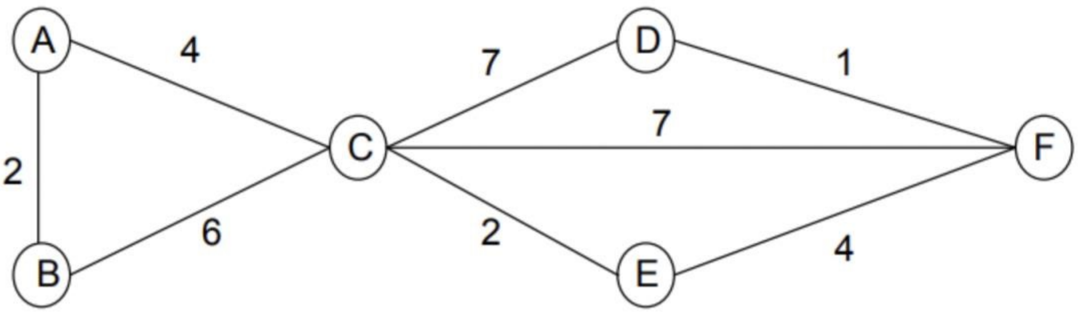


Figure 7 Example of an undirected weighted graph

Initially, a random vertex A has been chosen to begin a tree. Figure 8 shows the connection established between the vertices as a tree

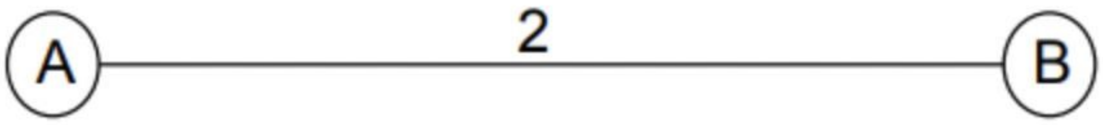


Figure 8 Initial node A growing a tree with the edge B

Consequently, A and C is further linked as shown in Figure 9 since A and C have a cost of 4 compared to A to B to C.

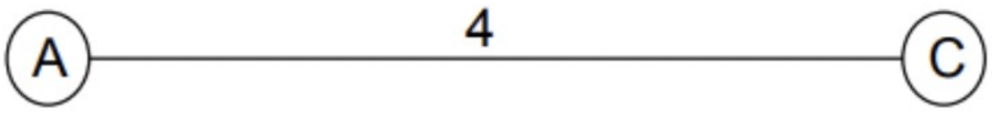


Figure 9 Connection the nodes A and C as part of the tree

Such process is further repeated to expand the tree as described in Figure 10

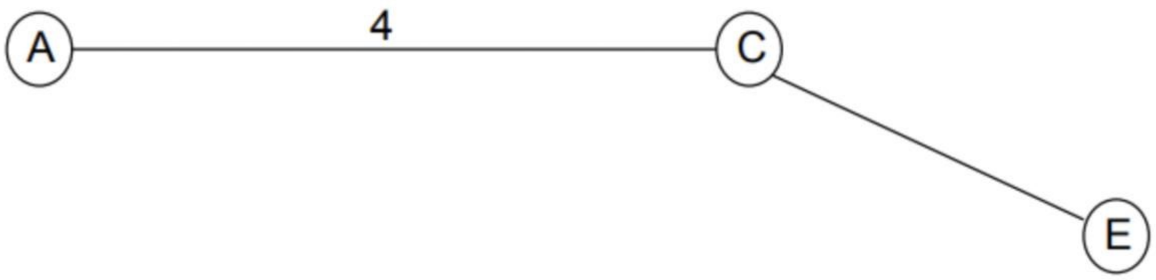


Figure 10 Expansion of the tree with the least cost

Priority queue makes the algorithm efficient in two ways [10]: Extracting the minimum from all the possible vertices that aren't part of the minimum spanning tree and extracting the vertex using the new keys of its adjacent vertex.

3. Experiment Methodology

Primary experimental data is the main source of data in this paper. Two algorithms, Dijkstra, and Prim were programmed (given in the appendix, heavily adapted from [6],[7], [12] and [14]) to run with 7 random inputs, increasing from 4 to 11 vertices, with and without priority queue. This experimental method was chosen because there was limited secondary data to answer this paper's question, except for the mathematical approach. An experimental method hasn't been done to see the data structure's effect on the performance that can be used. In addition, providing comparison through implementation helps in getting an accurate picture of the impact the data structure has on the performance of the algorithms since many constants are excluded in the mathematical interpretation of the algorithms. However, some limitations do exist in this methodology; the number of vertices has only been 7 as going further affect the control variable, the computer. The data given to the algorithm as input is written beforehand in the code itself. It is then executed, and the time is recorded as input. For every input given, the output is taken thrice and then averaged to increase the reliability of the values, thereby strengthening the primary data.

Variables' part of the Experiment

The independent variable would be the number of vertices provided to each of the algorithms since they will be under control. The dependent variable would be the amount of time taken with and without the priority queue since an increase in vertices would increase the algorithm's running time. The control variable would be the input used before and after using the priority queue and the computer specs.

4. The Experimental Results

4.1 Tabular Data Presentation

Prim with priority queue					Prim without priority queue				
Vertices	Output 1	Output 2	Output 3	Average	Vertices	Output 1	Output 2	Output 3	Average
4	77670	59249	67460	68126.3333	4	45040	63850	69000	59296.6667
5	61509	49680	56430	55873	5	76069	47120	73850	65679.6667
6	69570	52711	50810	57697	6	139171	89210	55831	94737.3333
8	47820	57010	67940	57590	8	53610	61770	98270	71216.6667
9	54130	56731	74030	61630.3333	9	76691	70030	62580	69767
10	83829	84140	74780	80916.3333	10	60620	128590	78930	89380
11	107580	87090	66009	86893	11	113240	90709	75590	93179.6667

Table 1 Time in Nanoseconds for Prim's algorithm with and without priority queue

Dijkstra without priority queue					Dijkstra with priority queue				
Vertices	Output 1	Output 2	Output 3	Average	Vertices	Output 1	Output 2	Output 3	Average
4	45040	63850	69000	59296.6667	4	58220	57080	46291	53863.6667
5	76069	47120	73850	65679.6667	5	50040	65360	69500	61633.3333
6	139171	89210	55831	94737.3333	6	73210	86660	96110	85326.6667
8	171420	52680	54840	92980	8	81820	67030	93710	80853.3333
9	82871	70370	84781	79340.6667	9	109109	114800	77550	100486.333
10	68510	90290	99880	86226.6667	10	94700	119710	94430	102946.667
11	99310	84960	130570	104946.667	11	105760	95711	87520	96330.3333

Table 2 Time in Nanoseconds for Dijkstra's algorithm with and without the priority queue

Graphical Presentation of Dijkstra’s algorithm without a priority queue

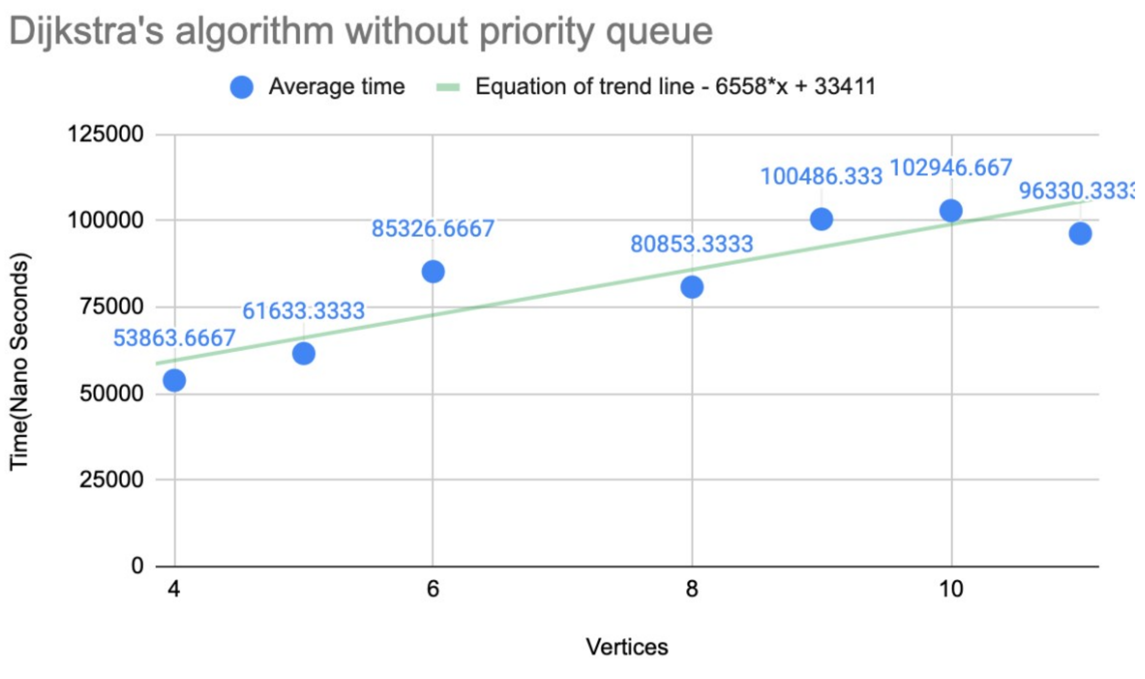


Figure 11 Graph of Dijkstra's algorithm without priority queue

Graphical Presentation of Dijkstra’s algorithm with a priority queue

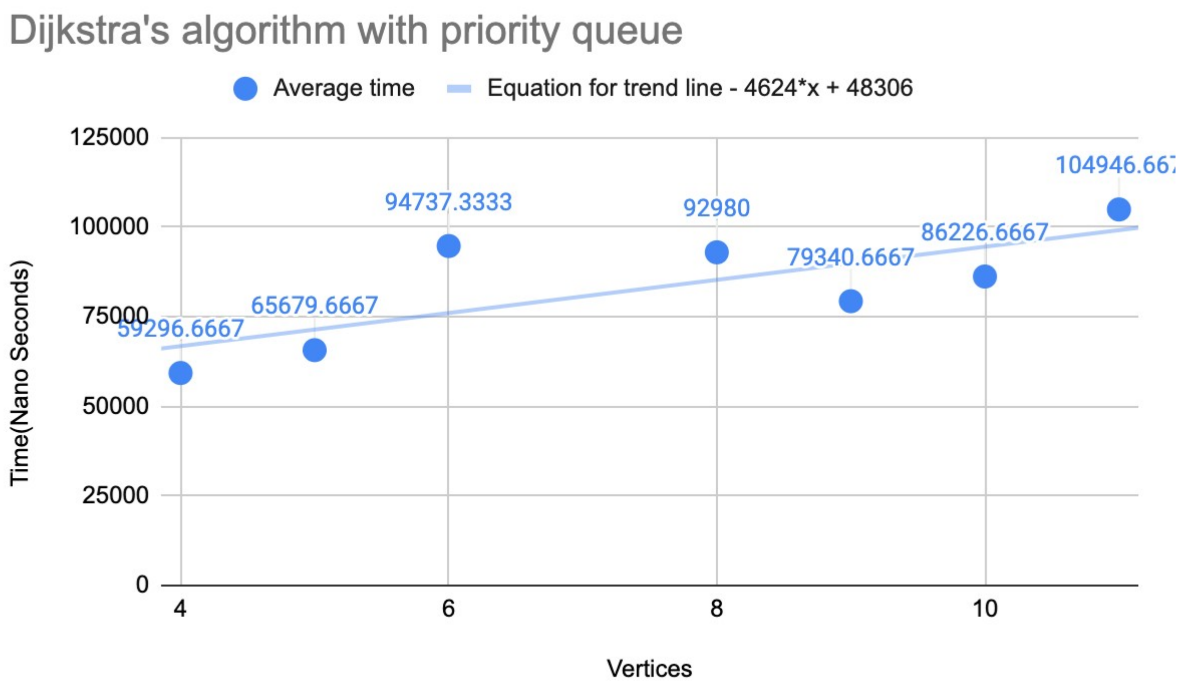


Figure 12 Graph of Dijkstra's algorithm with priority queue

Graphical Presentation of Prim’s Algorithm without a priority queue

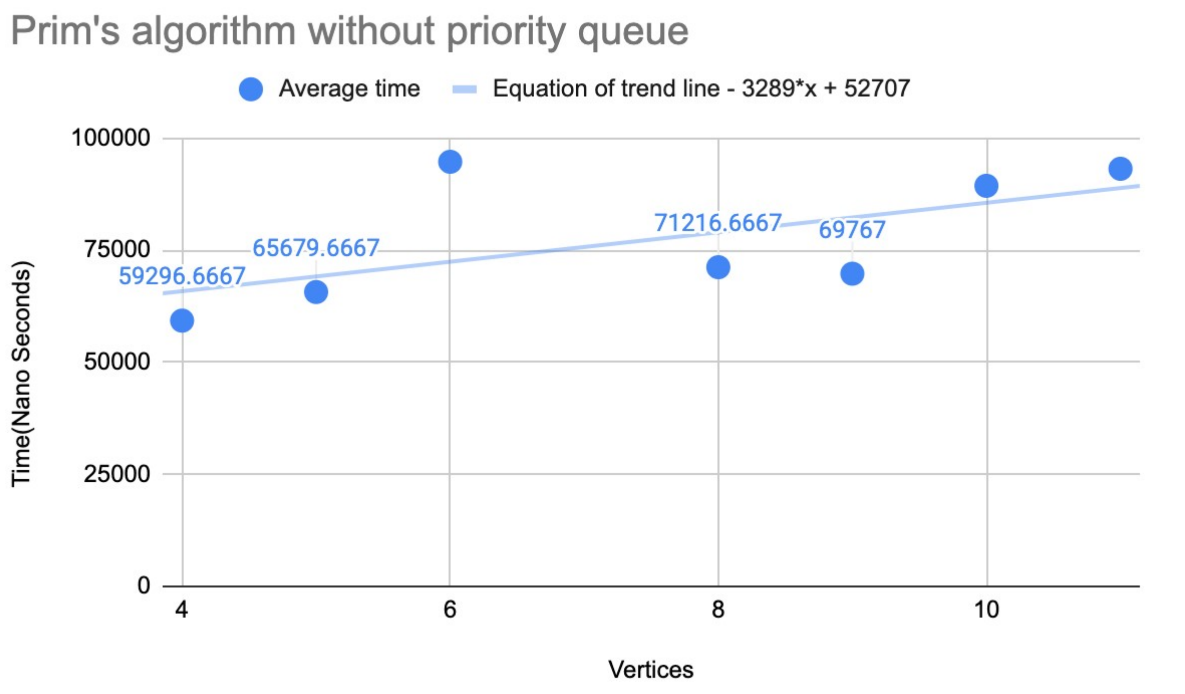


Figure 13 Graph of Prim's algorithm without priority queue

Graphical Presentation of Prim’s Algorithm with a priority queue

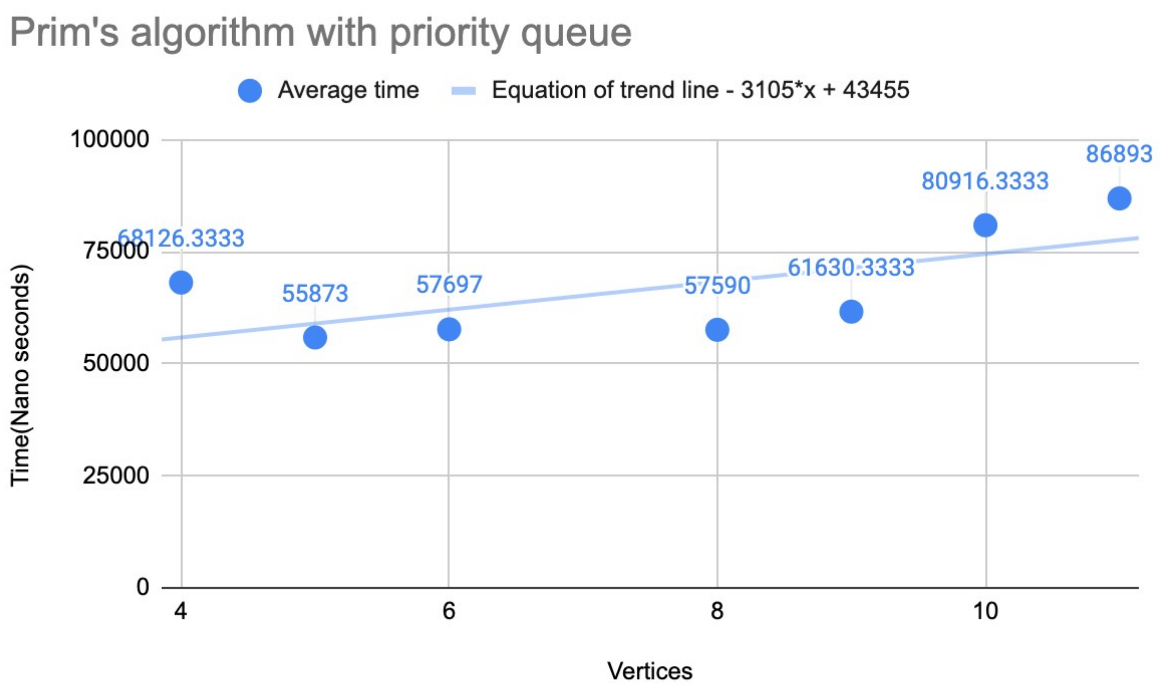


Figure 14 Graph of Prim's algorithm with priority queue

4.2 Data Analysis

The time is taken in nanoseconds as compared to other seconds, it is in integer format. Therefore, it becomes easier to draw graphs and analyse them accordingly. Scatter plot is the primary method for drawing graphs as it helps in understanding the relationships between variables, thereby allowing us to understand whether the use of data structure would affect the performance or not. As theorised, the data has been consistent with the kind of effect the data structure would have on the performance of the algorithms.

4.2.1 - Analysing Dijkstra's Algorithm

The equation of the trend line in Figure 11 is less than Figure 12's trend line. This suggests that there is relatively **less growth** by approximately **29%.** The growth rate is extremely important when it comes to the analysis of the performance. As the number of vertices increases, the growth rate shows the rate at which the time is growing. Therefore, it can be inferred that the growth of Dijkstra's algorithm without a priority queue is **less** than Dijkstra's algorithm with a priority queue. The primary that this is possible for the difference in growth is the loss of repetition that is brought by the priority queue. When an adjacency matrix is used in Dijkstra's algorithm, a single edge with a particular weight is repeated in two locations. For example, consider two vertices 3 and 4 with an edge comprising a weight of 7. Now in an adjacency matrix, 7 would be presented in the 3rd row of the 4th column and 4th row of the 3rd column. On contrary, the priority queue stores value with new edges and do not ensure repetition. This repetition could be one of the causes for the increase in time with the increasing number of vertices. Such trends from Figures 11 and 12 could help in understanding that the dynamic approach of storing information in a priority queue could be one of the reasons for the improvement of performance.

In Figure 12, an anomaly is detected (94737.3333) through intuition as compared to other data points, it is farther from the line. Such anomaly could be present due to the CPU processing time taken as it is subject to change for every execution. This can be noticed in Table 2 for vertice 6 where Output 1 is 139171 nanoseconds, which is way off the margin as compared to Output 2 and 3. Such anomaly could impose a problem in devising the correlation relationship. However, exempting them could provide an accurate understanding of how well the performance of the algorithm has improved with the use of a priority queue.

Figure 12 shows 4 points which compared to Figure 11, are at a **relatively lower** position **vertically**. For example, 53863 < 39296 [n =4]; 65679>61633 [n=5] etc. Although they have a different number of vertices, the difference between them in terms of vertices is extremely less. Therefore, they can be considered as examples. The most optimum performance for any algorithm is when the y=0. Therefore, it implies that the **closer** the complexity is, the more efficient it is. These 4 points help in showing that the approach with the priority queue makes the algorithm takes less time.

4.2.2 - Analysing Prim's Algorithm

Like Dijkstra's algorithm, Prim's algorithm does have similar characteristics as the essence is the same for both the algorithms. Figure 14 shows the trend line's equation is relatively less compared to the trend line's equation of Figure 13 by approximately **5%** which suggests that there is **somewhat improvement** in performance with the use of priority queue. Comparative to Dijkstra's algorithm, the use of priority queue didn't show any significant improvement by a higher margin. This could be primarily because of the CPU specs. Although considered as a control variable, the difference in running time could show how effective it is in impacting the time taken for an algorithm to execute. Because Dijkstra and Prim's algorithm uses the greedy approach in finding the vertices with the shortest weight, the difference in the use of priority queue must be the same. Since it isn't, it can be inferred that the computer specs can impact it, which is something that needs to be changed for a better experiment. This doesn't mean that the experimental results can be proved to be false but could be something that can be worked upon.

The overall line in Figure 14 is relatively closer to the x-axis than in Figure 13. As explained in the third argument for the analysis of Dijkstra's algorithm, the closer the points are, and by extension the line is, the more efficient it is. Such a trend can be observed in Figure 14 as compared to Figure 13, which is relatively farther away. This could be because of a big anomaly, such as the one with 6 vertices in Figure 13 with 94737 nanoseconds. Despite excluding them, the overall growth in time from Table 1 and Figure 13 helps in understanding the reason why the graph is quite farther away from the y-axis.

<u>5. Conclusion and Evaluation</u>

In this paper, the effect of using an abstract data structure in Dijkstra and Prim's algorithm is experimented with and analysed. Logical explanations for the trends and anomalies were provided.

The results show that the algorithm takes **relatively less time** with the use of data structure as compared to the classical adjacency matrix way. Although there were some anomalies outside the pattern due to the specs, the final trend of difference in performance was observed.

By using the priority queue, the repetition of storing the same element wouldn't be possible as it primarily has new edges and does not repeat two edges in terms of storage. Therefore, it supersedes in performance than the adjacency matrix. Although there isn't a significant improvement as per the experimental results, having more vertices with better computer specs as the control variable could make the difference more accurate and visible. However, it can be finally said that using a priority queue is a better option for these algorithms Hopefully, this paper will help software engineers, programmers and other students who encounter these algorithms to pursue this data structure to save their running time and thereby be able to present a much more efficient solution with the least resources, and be able to solve constraint-based problems easily

6. Works Cited

Most of these references were only used in the Theoretical Background information section.

[1] - Pirzada, S., 2008. __Applications of Graph Theory__. [online] https://onlinelibrary.wiley.com/. Available at: <https://onlinelibrary.wiley.com/doi/10.1002/pamm.200700981>

[2] - Wikipedia. (2020). *Analysis of algorithms*. [online] Available at: https://en.wikipedia.org/wiki/Analysis_of_algorithms [Accessed 15 Jun. 2020].

[3] - aofa.cs.princeton.edu. (n.d.). *Analysis of Algorithms*. [online] Available at: https://aofa.cs.princeton.edu/10analysis/ [Accessed 6 Jul. 2020].

[4] - dipesh99kumar (2020). *Applications of Dijkstra’s shortest path algorithm*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/applications-of-dijkstrasshortest-path-algorithm/ [Accessed Jul. 12AD].

[5] - Mitanshupbhoot (2020). *Comparative applications of Prim’s and Kruskal’s algorithm in real-life scenarios*. [online] Medium. Available at: https://medium.com/@mitanshupbhoot/comparative-applications-of-prims-and-kruskal-salgorithm-in-real-life-scenarios-4aa0f92c7abc [Accessed 20 Jul. 2020].

[6] - Geeks for Geeks (2016). *Dijkstra’s Shortest Path Algorithm using priority_queue of STL*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/dijkstras-shortestpath-algorithm-using-priority_queue-stl/ [Accessed 17 Jul. 2020].

[7] - Geeks for Geeks (n.d.). *Prim’s Minimum Spanning Tree (MST) | Greedy Algo-5 - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/primsminimum-spanning-tree-mst-greedy-algo-5/ [Accessed 16 Jun. 2020].

[8] - Cormen, T. H., & Leiserson, C. E. (2009). *Introduction to algorithms, 3rd Edition*. 24.3 - Dijkstra's algorithm on Page 658
[9] - www.programiz.com. (n.d.). *Prim’s Algorithm*. [online] Available at: https://www.programiz.com/dsa/prim-algorithm [Accessed 5 Sep. 2020].

[10] - Wikipedia. (2020). *Priority queue*. [online] Available at: https://en.wikipedia.org/wiki/Priority_queue [Accessed 18 Sep. 2020].

[11] - Wikipedia Contributors (2019). *Dijkstra’s algorithm*. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm [Accessed 13 Aug. 2020].

[12] - GeeksforGeeks. (2016). *Prim’s algorithm using priority_queue in STL*. [online] Available at: https://www.geeksforgeeks.org/prims-algorithm-using-priority_queue-stl/ [Accessed 18 Jun. 2020].

[13] - GeeksforGeeks. (2018). *Priority Queue | Set 1 (Introduction) - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/priority-queue-set-1-introduction/ [Accessed 19 Jun. 2020].

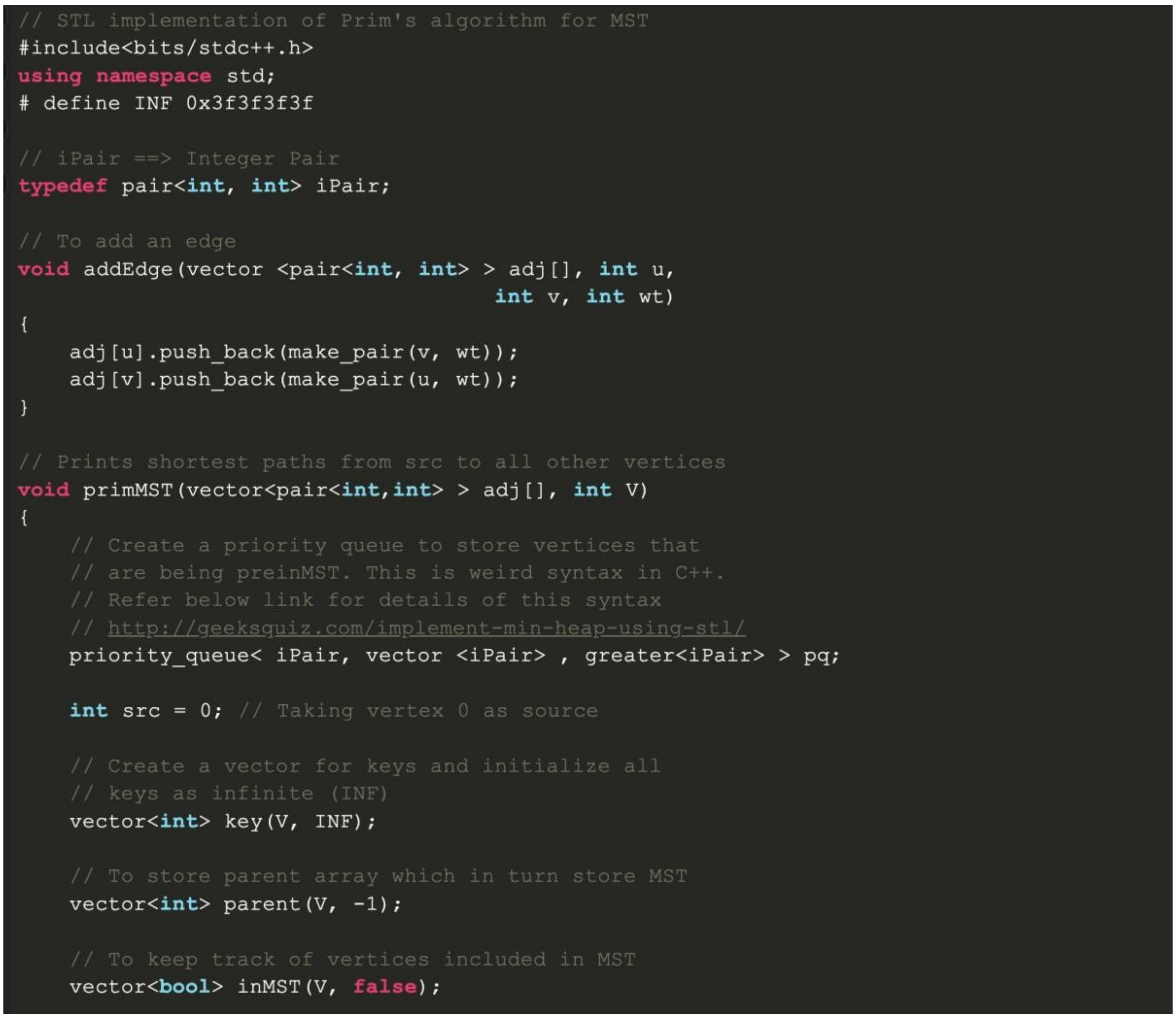
[14] - Geeks for Geeks (2018). *Dijsktra’s algorithm*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/ [Accessed 26 Sep. 2020].

[15] - Software Testing Help. (n.d.). *Minimum Spanning Tree Tutorial: Prim's and Kruskal's Algorithms*. [online] Available at: https://www.softwaretestinghelp.com/minimum-spanningtree-tutorial/ [Accessed 19 Jun. 2020].

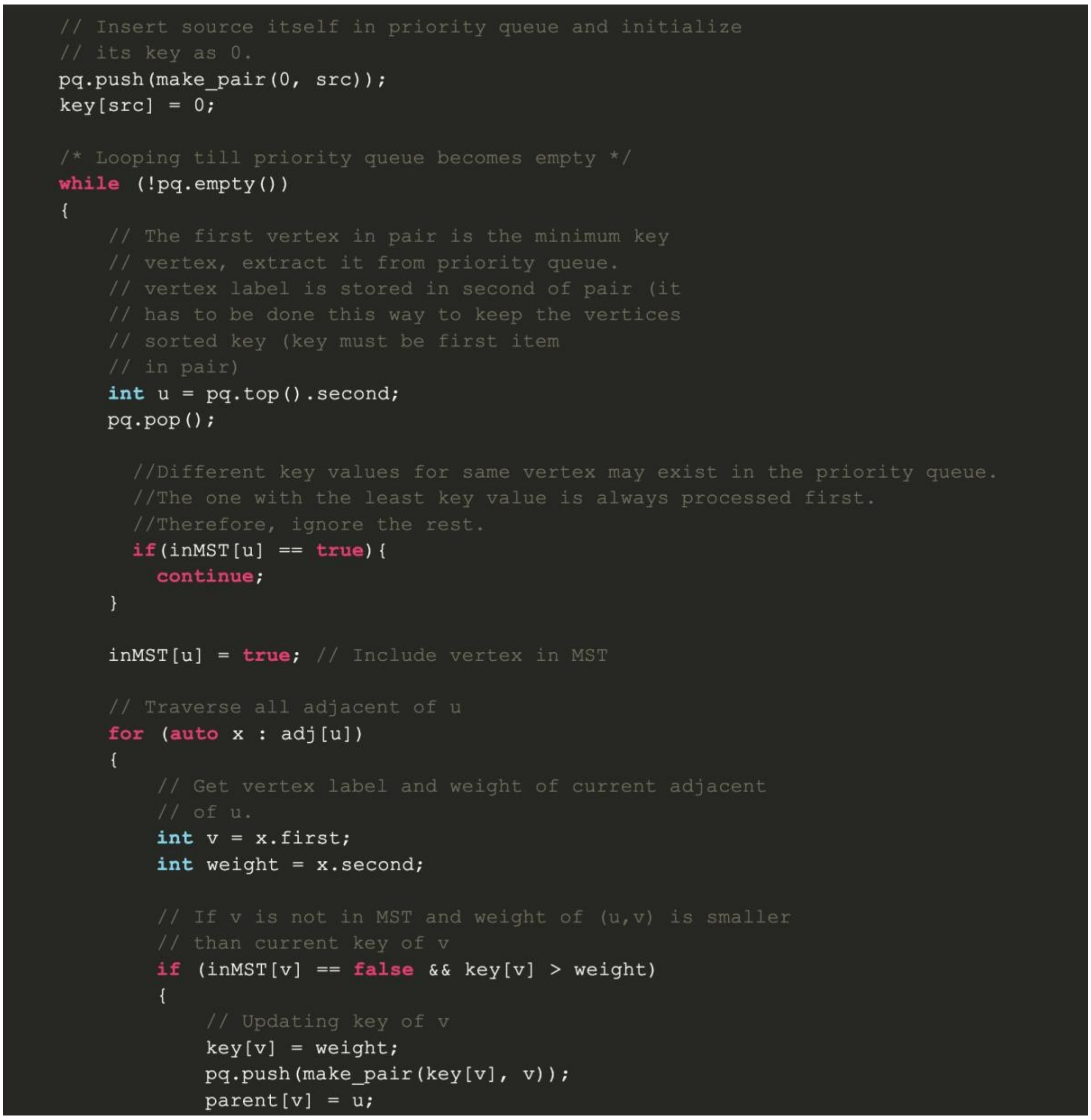
7. Appendix

The used codes given below are taken and modified from [6]and [7] and have been modified while collecting data.

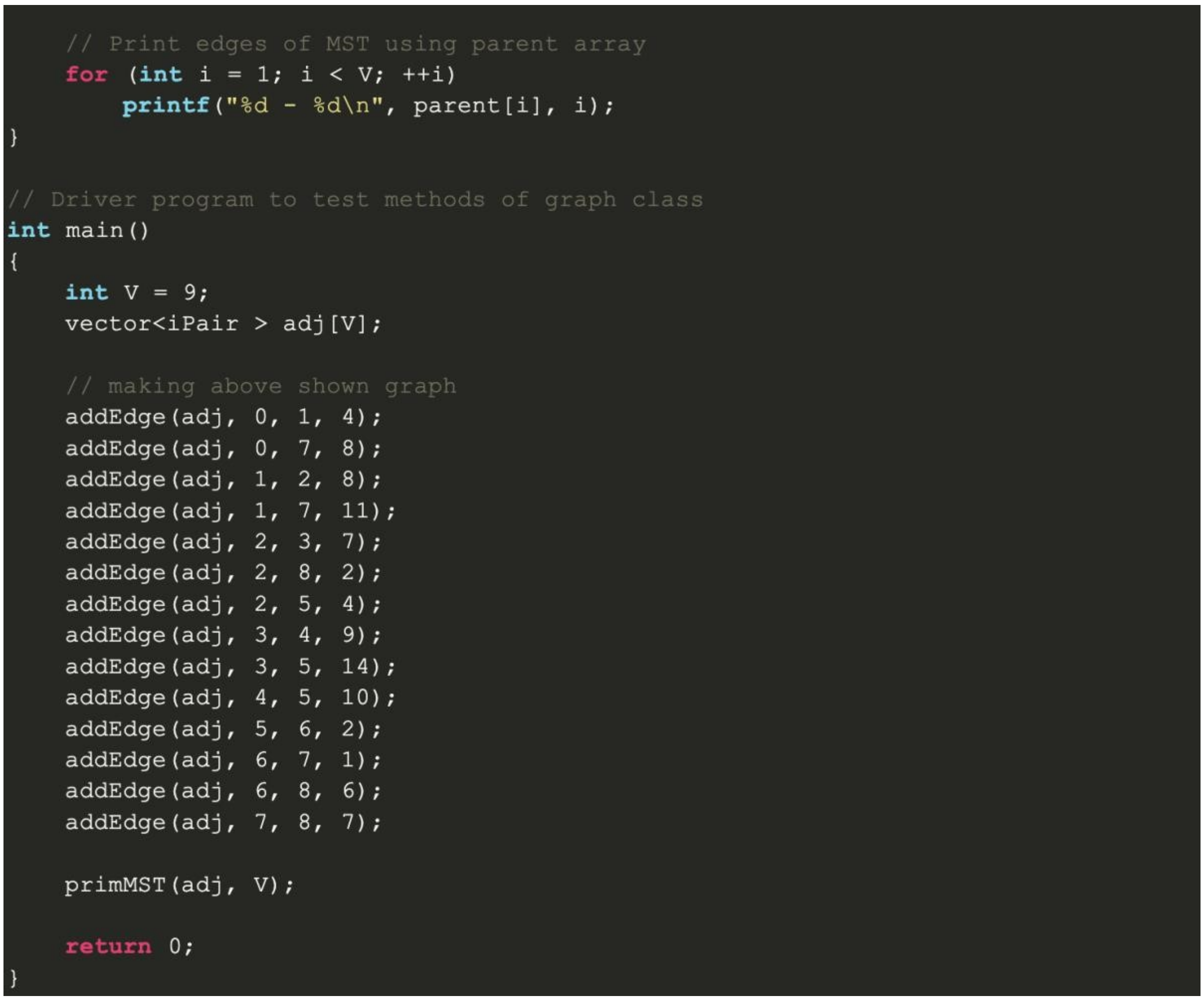
7.1 Programs for Prim's Algorithm



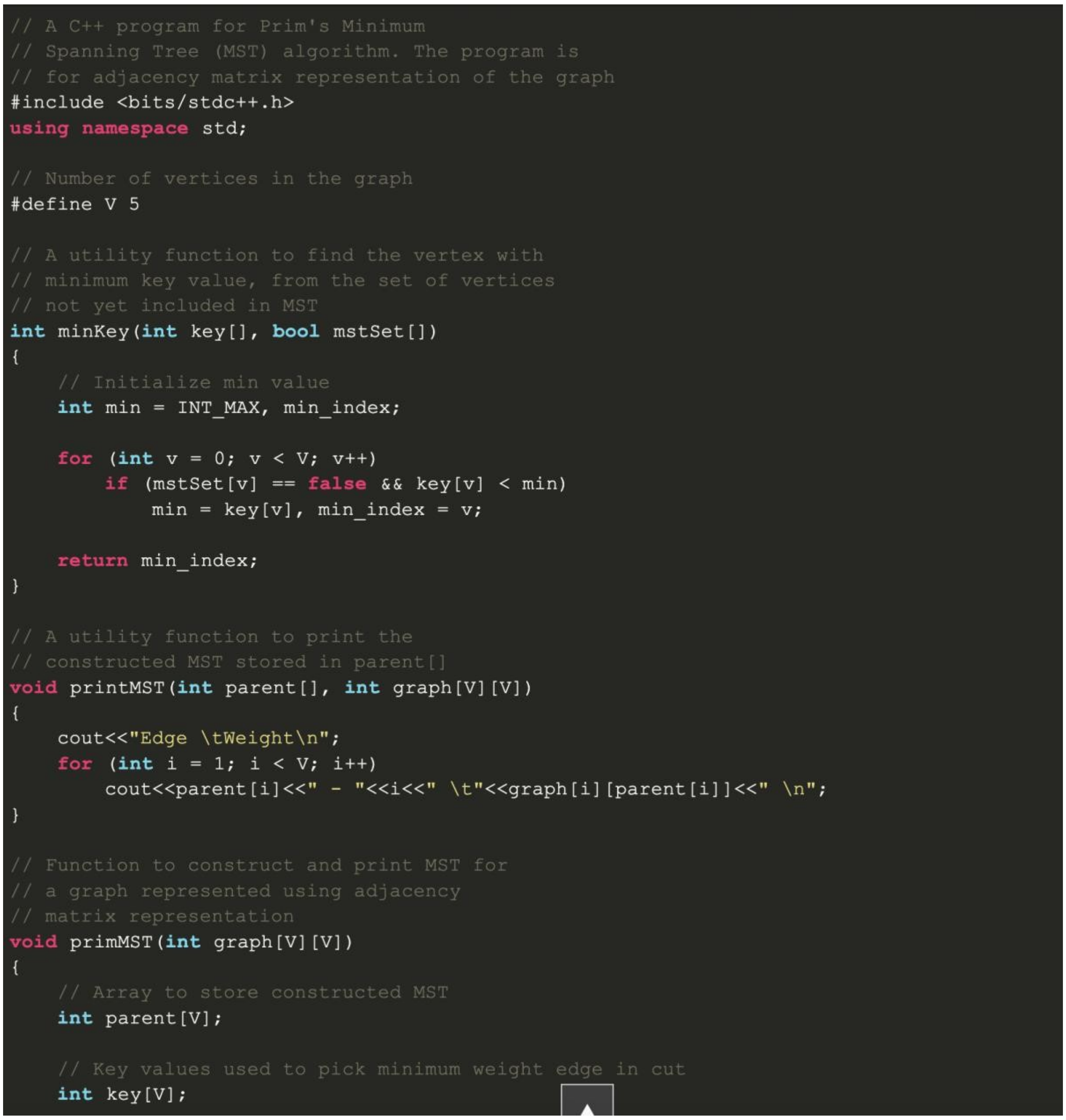
```
// STL implementation of Prim's algorithm for MST
#include<bits/stdc++.h>
using namespace std;
# define INF 0x3f3f3f3f

// iPair ==> Integer Pair
typedef pair<int, int> iPair;

// To add an edge
void addEdge(vector <pair<int, int> > adj[], int u,
                                    int v, int wt)
{
    adj[u].push_back(make_pair(v, wt));
    adj[v].push_back(make_pair(u, wt));
}

// Prints shortest paths from src to all other vertices
void primMST(vector<pair<int,int> > adj[], int V)
{
    // Create a priority queue to store vertices that
    // are being preinMST. This is weird syntax in C++.
    // Refer below link for details of this syntax
    // http://geeksquiz.com/implement-min-heap-using-stl/
    priority_queue< iPair, vector <iPair> , greater<iPair> > pq;

    int src = 0; // Taking vertex 0 as source

    // Create a vector for keys and initialize all
    // keys as infinite (INF)
    vector<int> key(V, INF);

    // To store parent array which in turn store MST
    vector<int> parent(V, -1);

    // To keep track of vertices included in MST
    vector<bool> inMST(V, false);
```



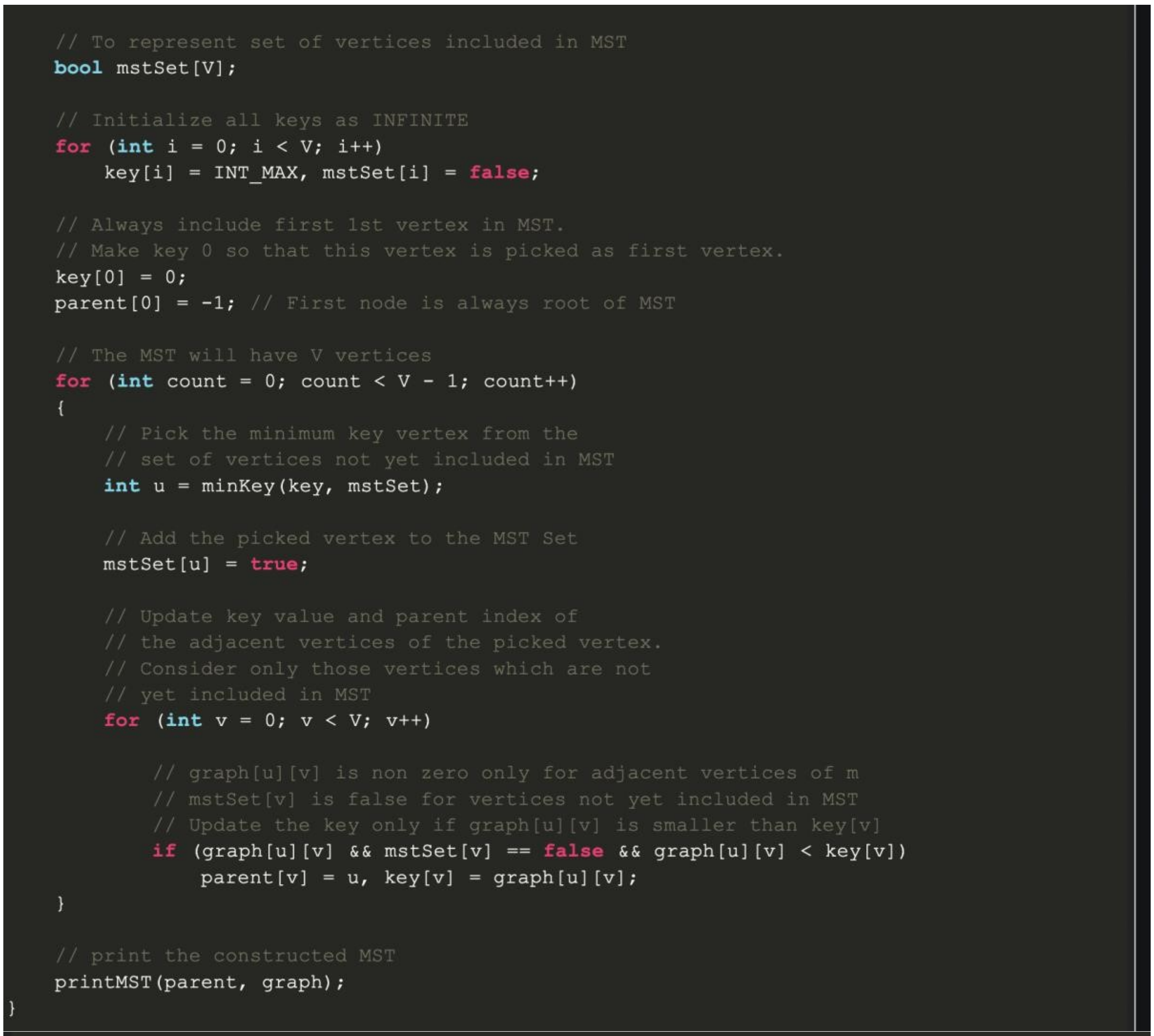
```
    // Insert source itself in priority queue and initialize
    // its key as 0.
    pq.push(make_pair(0, src));
    key[src] = 0;

    /* Looping till priority queue becomes empty */
    while (!pq.empty())
    {
        // The first vertex in pair is the minimum key
        // vertex, extract it from priority queue.
        // vertex label is stored in second of pair (it
        // has to be done this way to keep the vertices
        // sorted key (key must be first item
        // in pair)
        int u = pq.top().second;
        pq.pop();

          //Different key values for same vertex may exist in the priority queue.
          //The one with the least key value is always processed first.
          //Therefore, ignore the rest.
          if(inMST[u] == true){
            continue;
        }

        inMST[u] = true; // Include vertex in MST

        // Traverse all adjacent of u
        for (auto x : adj[u])
        {
            // Get vertex label and weight of current adjacent
            // of u.
            int v = x.first;
            int weight = x.second;

            // If v is not in MST and weight of (u,v) is smaller
            // than current key of v
            if (inMST[v] == false && key[v] > weight)
            {
                // Updating key of v
                key[v] = weight;
                pq.push(make_pair(key[v], v));
                parent[v] = u;
```



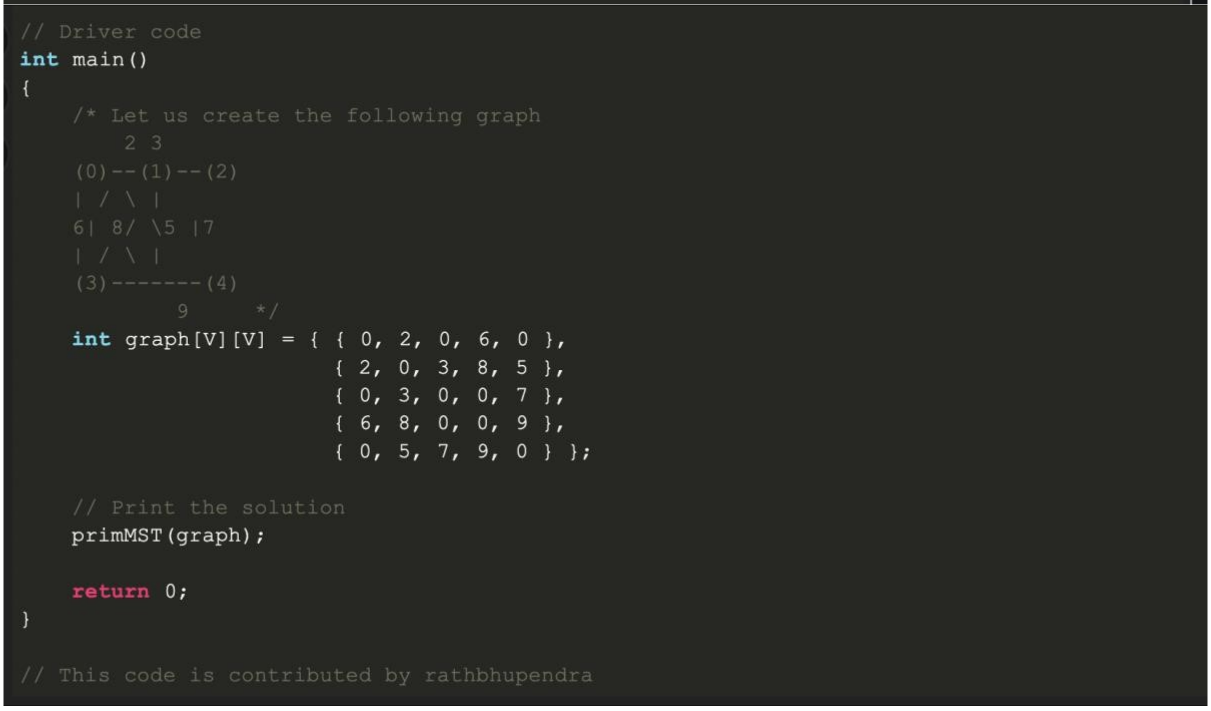
```
    // Print edges of MST using parent array
    for (int i = 1; i < V; ++i)
        printf("%d - %d\n", parent[i], i);
}

// Driver program to test methods of graph class
int main()
{
    int V = 9;
    vector<iPair > adj[V];

    // making above shown graph
    addEdge(adj, 0, 1, 4);
    addEdge(adj, 0, 7, 8);
    addEdge(adj, 1, 2, 8);
    addEdge(adj, 1, 7, 11);
    addEdge(adj, 2, 3, 7);
    addEdge(adj, 2, 8, 2);
    addEdge(adj, 2, 5, 4);
    addEdge(adj, 3, 4, 9);
    addEdge(adj, 3, 5, 14);
    addEdge(adj, 4, 5, 10);
    addEdge(adj, 5, 6, 2);
    addEdge(adj, 6, 7, 1);
    addEdge(adj, 6, 8, 6);
    addEdge(adj, 7, 8, 7);

    primMST(adj, V);

    return 0;
}
```



```
// A C++ program for Prim's Minimum
// Spanning Tree (MST) algorithm. The program is
// for adjacency matrix representation of the graph
#include <bits/stdc++.h>
using namespace std;

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V])
{
    cout<<"Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<" \n";
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    // Array to store constructed MST
    int parent[V];

    // Key values used to pick minimum weight edge in cut
    int key[V];
```

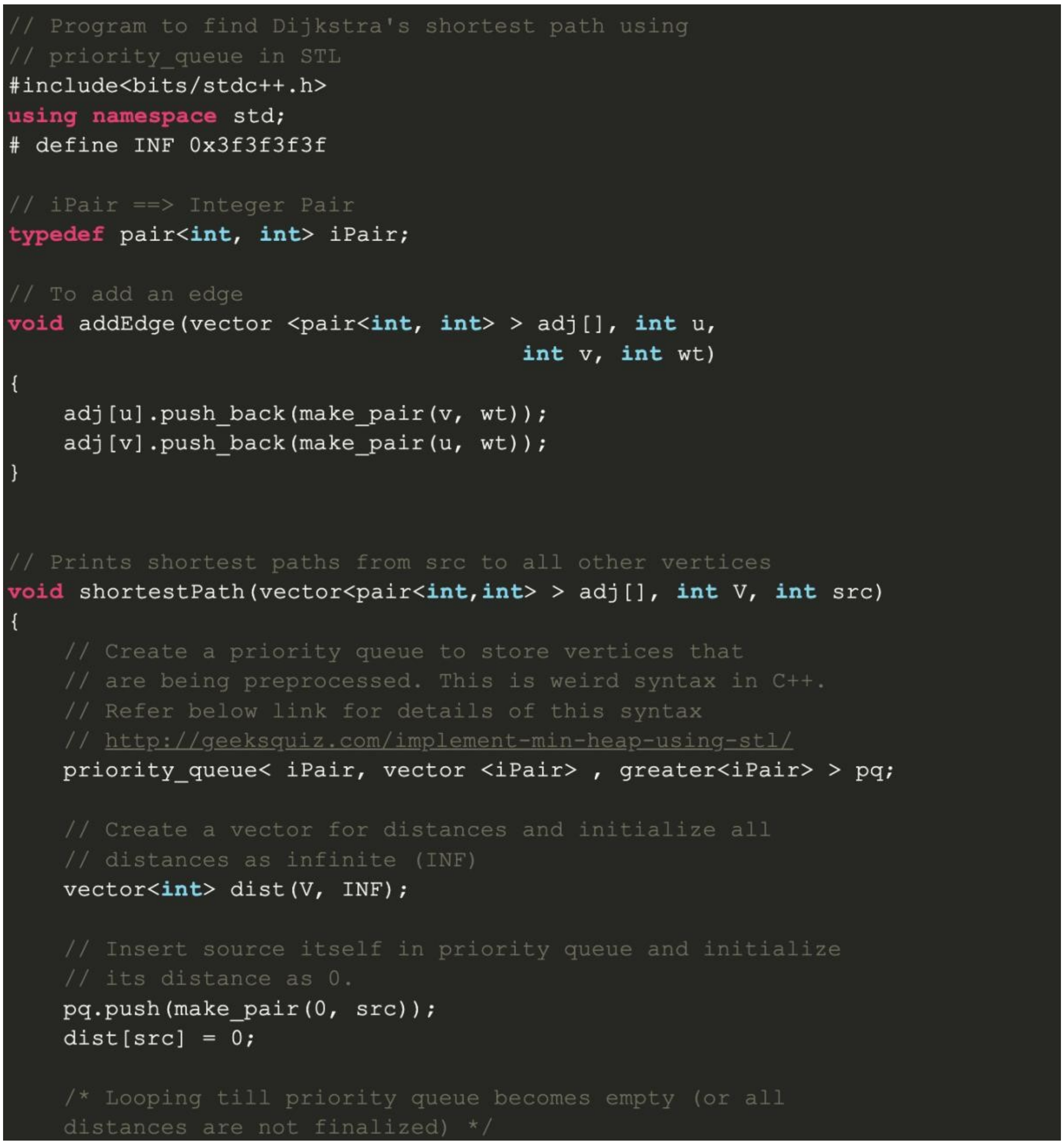


```
    // To represent set of vertices included in MST
    bool mstSet[V];

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    // Make key 0 so that this vertex is picked as first vertex.
    key[0] = 0;
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++)
    {
        // Pick the minimum key vertex from the
        // set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of
        // the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not
        // yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent vertices of m
            // mstSet[v] is false for vertices not yet included in MST
            // Update the key only if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // print the constructed MST
    printMST(parent, graph);
}
```

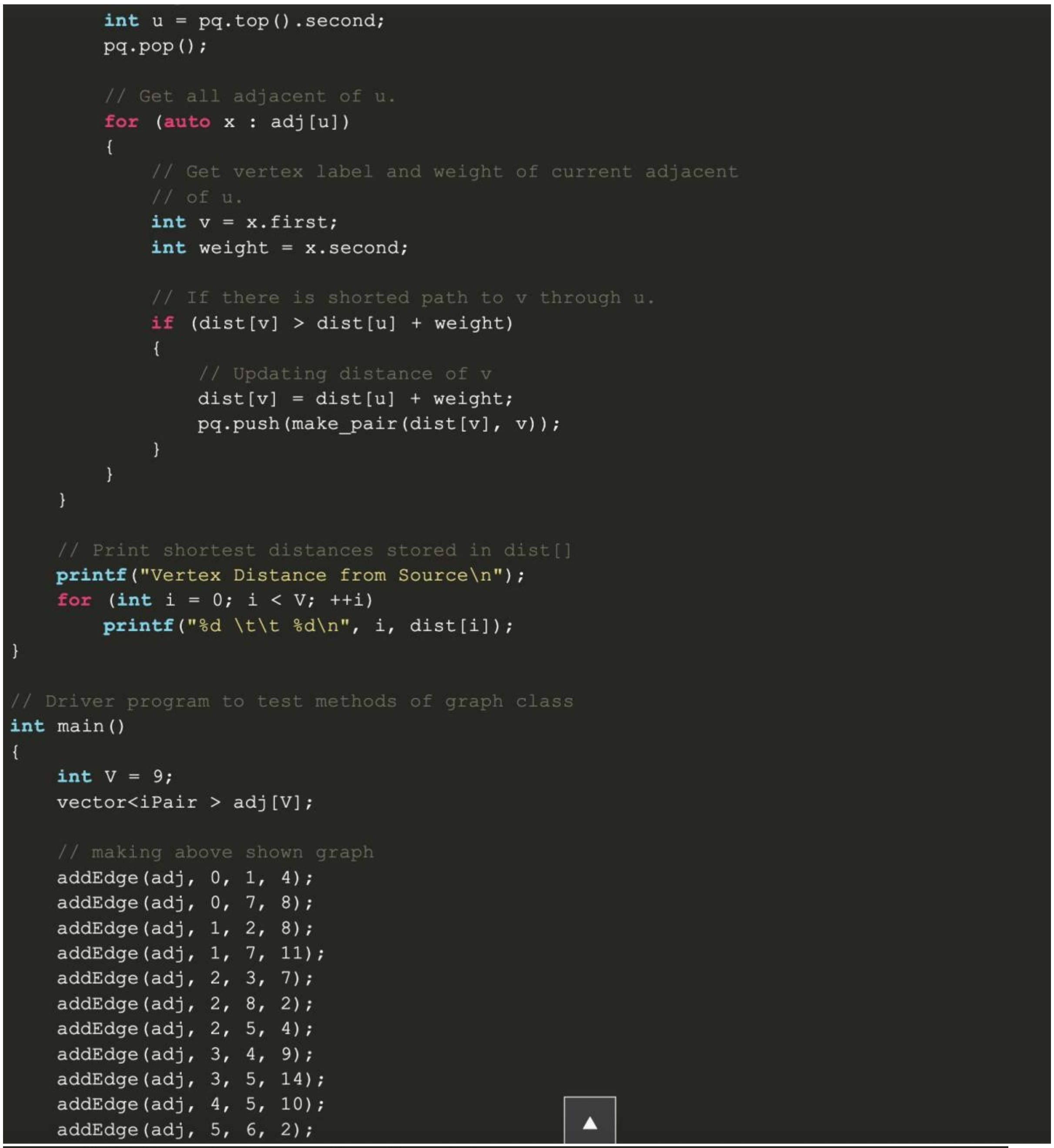


```
// Driver code
int main()
{
    /* Let us create the following graph
        2  3
    (0)--(1)--(2)
    | / \ |
    6| 8/ \5 |7
    | / \ |
    (3)-------(4)
            9     */
    int graph[V][V] = { { 0, 2, 0, 6, 0 },
                        { 2, 0, 3, 8, 5 },
                        { 0, 3, 0, 0, 7 },
                        { 6, 8, 0, 0, 9 },
                        { 0, 5, 7, 9, 0 } };

    // Print the solution
    primMST(graph);

    return 0;
}

// This code is contributed by rathbhupendra
```

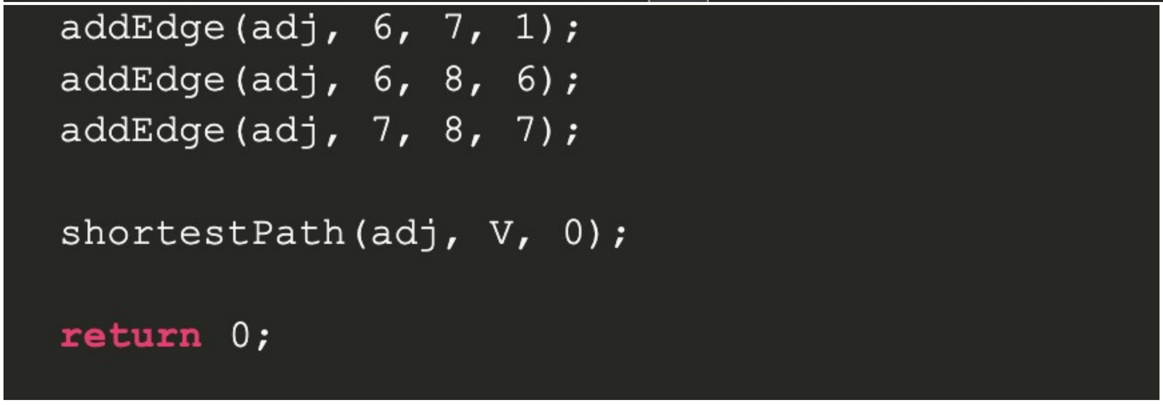
7.2 Programs for Dijkstra's Algorithm



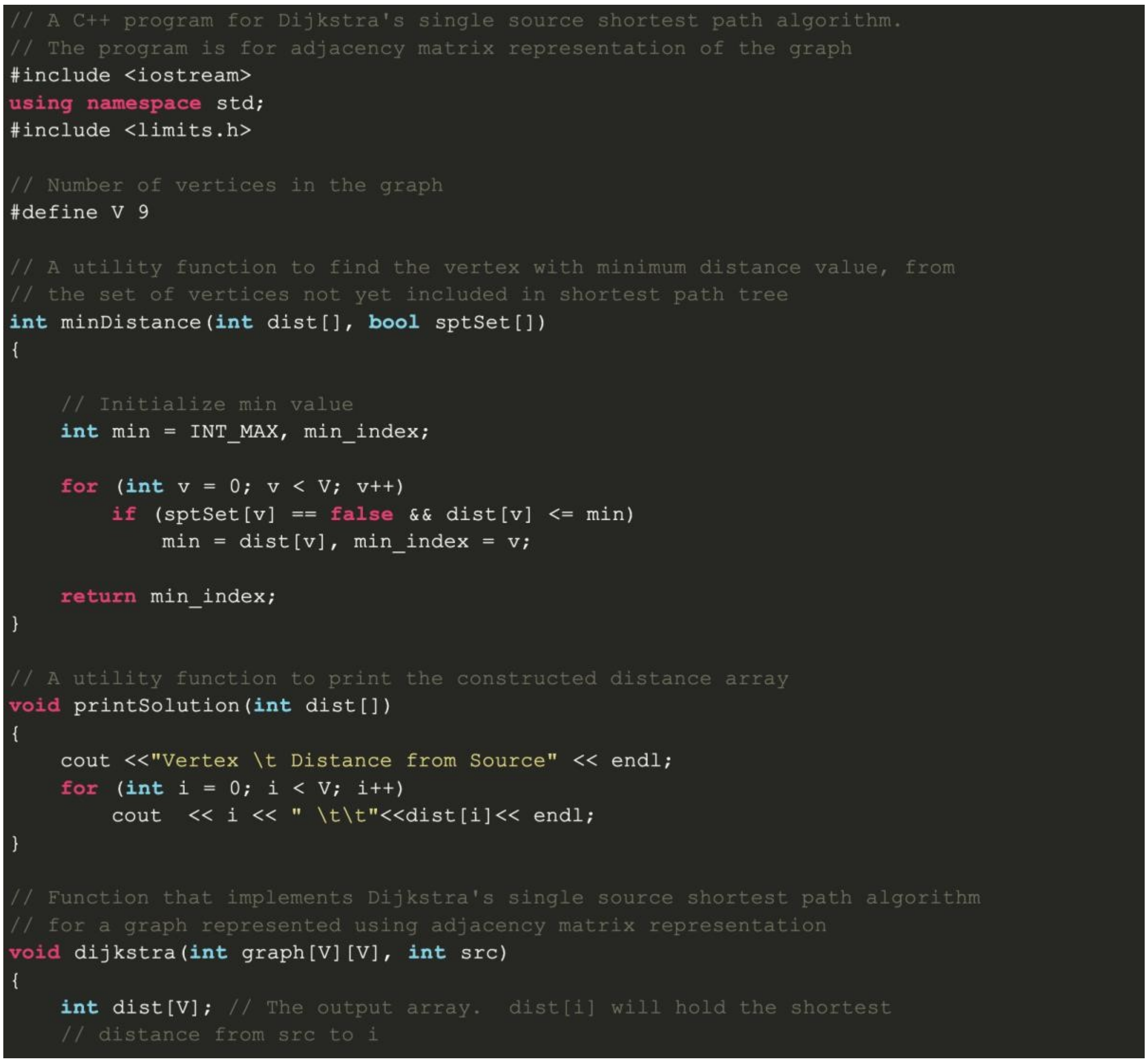
```
// Program to find Dijkstra's shortest path using
// priority_queue in STL
#include<bits/stdc++.h>
using namespace std;
# define INF 0x3f3f3f3f

// iPair ==> Integer Pair
typedef pair<int, int> iPair;

// To add an edge
void addEdge(vector <pair<int, int> > adj[], int u,
                                     int v, int wt)
{
    adj[u].push_back(make_pair(v, wt));
    adj[v].push_back(make_pair(u, wt));
}


// Prints shortest paths from src to all other vertices
void shortestPath(vector<pair<int,int> > adj[], int V, int src)
{
    // Create a priority queue to store vertices that
    // are being preprocessed. This is weird syntax in C++.
    // Refer below link for details of this syntax
    // http://geeksquiz.com/implement-min-heap-using-stl/
    priority_queue< iPair, vector <iPair> , greater<iPair> > pq;

    // Create a vector for distances and initialize all
    // distances as infinite (INF)
    vector<int> dist(V, INF);

    // Insert source itself in priority queue and initialize
    // its distance as 0.
    pq.push(make_pair(0, src));
    dist[src] = 0;

    /* Looping till priority queue becomes empty (or all
    distances are not finalized) */
```



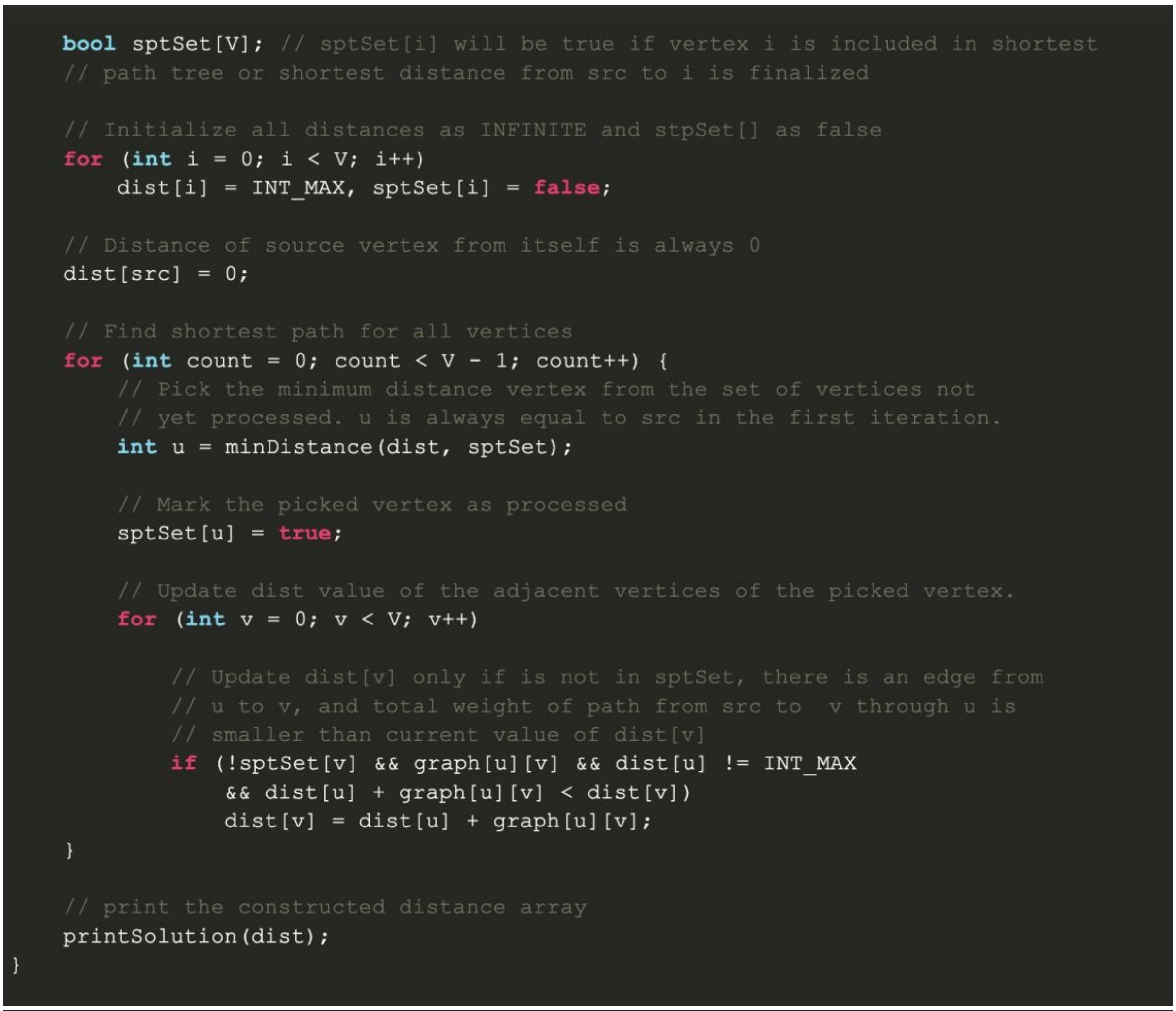
```
        int u = pq.top().second;
        pq.pop();

        // Get all adjacent of u.
        for (auto x : adj[u])
        {
            // Get vertex label and weight of current adjacent
            // of u.
            int v = x.first;
            int weight = x.second;

            // If there is shorted path to v through u.
            if (dist[v] > dist[u] + weight)
            {
                // Updating distance of v
                dist[v] = dist[u] + weight;
                pq.push(make_pair(dist[v], v));
            }
        }
    }

    // Print shortest distances stored in dist[]
    printf("Vertex Distance from Source\n");
    for (int i = 0; i < V; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// Driver program to test methods of graph class
int main()
{
    int V = 9;
    vector<iPair > adj[V];

    // making above shown graph
    addEdge(adj, 0, 1, 4);
    addEdge(adj, 0, 7, 8);
    addEdge(adj, 1, 2, 8);
    addEdge(adj, 1, 7, 11);
    addEdge(adj, 2, 3, 7);
    addEdge(adj, 2, 8, 2);
    addEdge(adj, 2, 5, 4);
    addEdge(adj, 3, 4, 9);
    addEdge(adj, 3, 5, 14);
    addEdge(adj, 4, 5, 10);
    addEdge(adj, 5, 6, 2);
```



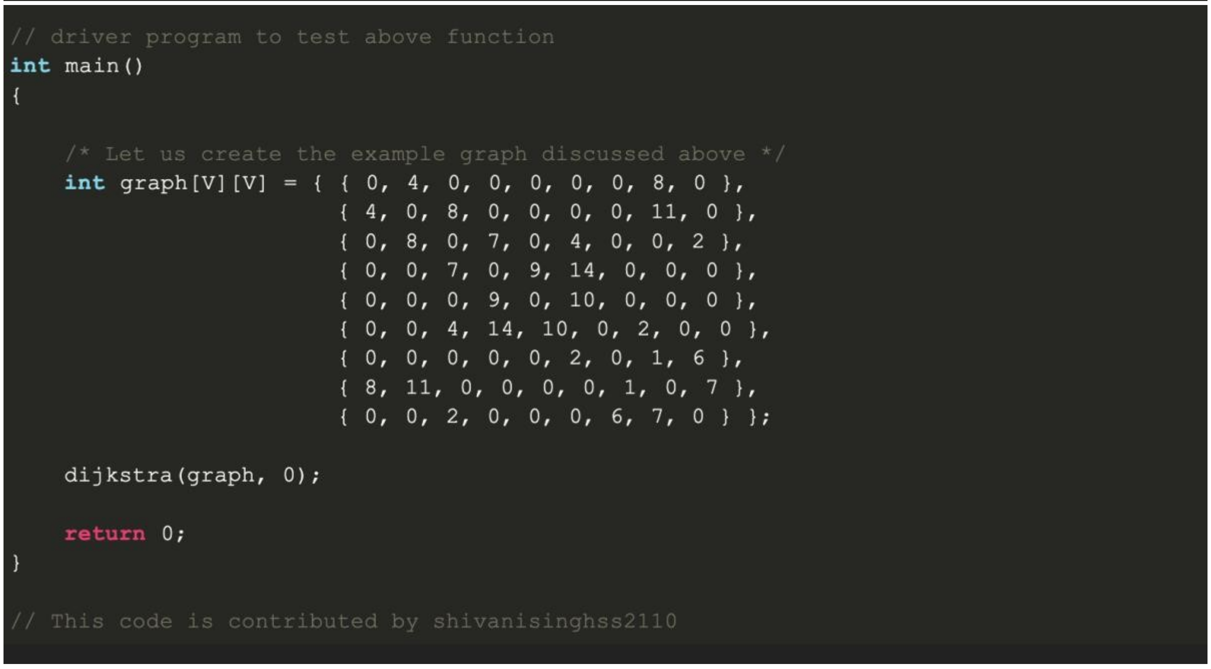
```
    addEdge(adj, 6, 7, 1);
    addEdge(adj, 6, 8, 6);
    addEdge(adj, 7, 8, 7);

    shortestPath(adj, V, 0);

    return 0;
```



```
// A C++ program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph
#include <iostream>
using namespace std;
#include <limits.h>

// Number of vertices in the graph
#define V 9

// A utility function to find the vertex with minimum distance value, from
// the set of vertices not yet included in shortest path tree
int minDistance(int dist[], bool sptSet[])
{

    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}

// A utility function to print the constructed distance array
void printSolution(int dist[])
{
    cout <<"Vertex \t Distance from Source" << endl;
    for (int i = 0; i < V; i++)
        cout  << i << " \t\t"<<dist[i]<< endl;
}

// Function that implements Dijkstra's single source shortest path algorithm
// for a graph represented using adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
    int dist[V]; // The output array.  dist[i] will hold the shortest
    // distance from src to i
```



```
    bool sptSet[V]; // sptSet[i] will be true if vertex i is included in shortest
    // path tree or shortest distance from src to i is finalized

    // Initialize all distances as INFINITE and stpSet[] as false
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum distance vertex from the set of vertices not
        // yet processed. u is always equal to src in the first iteration.
        int u = minDistance(dist, sptSet);

        // Mark the picked vertex as processed
        sptSet[u] = true;

        // Update dist value of the adjacent vertices of the picked vertex.
        for (int v = 0; v < V; v++)

            // Update dist[v] only if is not in sptSet, there is an edge from
            // u to v, and total weight of path from src to  v through u is
            // smaller than current value of dist[v]
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }

    // print the constructed distance array
    printSolution(dist);
}
```



```
// driver program to test above function
int main()
{

    /* Let us create the example graph discussed above */
    int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                        { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                        { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                        { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                        { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                        { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                        { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                        { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                        { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

    dijkstra(graph, 0);

    return 0;
}

// This code is contributed by shivanisinghss2110
```