KIET Group of Institutions, Ghaziabad

Computer Science and Information Technology

Design and Analysis of Algorithm

Project-based learning

Project

On

Dijkastra’s and Algorithm

Odd Semester

(2024-25)

Submitted To: Vinay sir

Submitted By: Pratik Kumar

Branch/Sec : CSIT 5B

Roll No: 2200290110128

Index

ACKNOWLEDGEMENT

I’ve got this golden opportunity to express my kind gratitude and sincere thanks to Mr. Abhinav Juneja, Head of Department of Computer Science and Information Technology and to our mentor Mr. Praveer Dubey for the time and efforts they provided throughout the year. Their useful advice and suggestions were helpful to me during the project’s completion. I’m also indebted to every person responsible for the making up of this project directly or indirectly.
I must also acknowledge our deep debt of gratitude to each one of my colleagues who led this project to come out in the way it is. It’s my hard work and untiring sincere efforts and cooperation to bring out the project work. Last but not least, I would like to thank my parents for their sound counselling and cheerful support. They have always inspired us and kept our spirit up.

Pratik Kumar
CSIT 5B
2200290110128

INTRODUCTION

PROBLEM STATEMENT:

In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.
The problem of finding the shortest path between two intersections on a road map may be modeled as a special case of the shortest path problem in graphs, where the vertices correspond to intersections and the edges correspond to road segments, each weighted by the length or distance of each segment.

MOTIVATION:

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a weighted graph, which may represent, for example, a road network. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.
Dijkstra's algorithm finds the shortest path from a given source node to every other node. It can be used to find the shortest path to a specific destination node, by terminating the algorithm after determining the shortest path to the destination node. For example, if the nodes of the graph represent cities, and the costs of edges represent the average distances between pairs of cities connected by a direct road, then Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. A common application of shortest path algorithms is network routing protocols, most notably IS-IS (Intermediate System to Intermediate System) and OSPF (Open Shortest Path First). It is also employed as a subroutine in algorithms such as Johnson's algorithm.

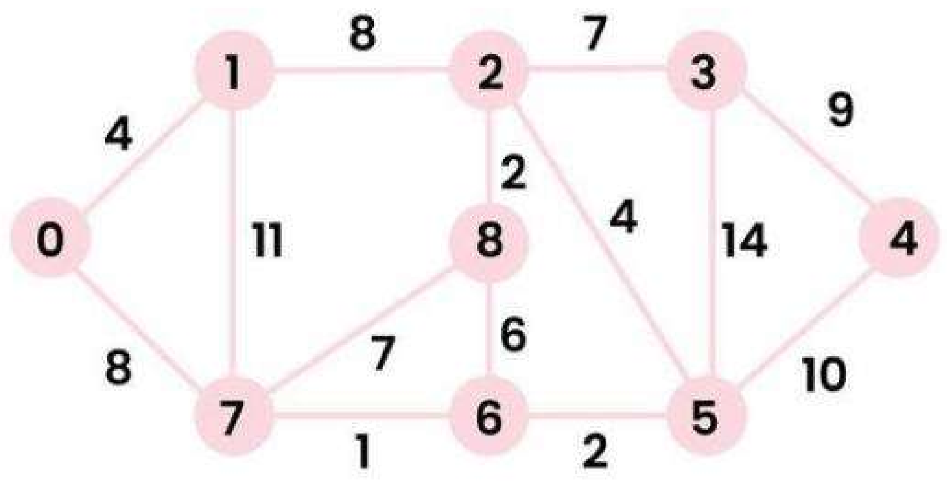
Algorithm (with Analysis)

The algorithm uses a min-priority queue data structure for selecting the shortest paths known so far. Before more advanced priority queue structures were discovered, Dijkstra's original algorithm ran in $\Theta(|V|^2)$ time, where $|V|$ is the number of nodes. The idea of this algorithm is also given in Leyzorek et al. 1957. Fredman & Tarjan 1984 proposed using a Fibonacci heap priority queue to optimize the running time complexity to $\Theta(|E|+|V|\log|V|)$. This is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights. However, specialized cases (such as bounded/integer weights, directed acyclic graphs etc.) can be improved further. If preprocessing is allowed, algorithms such as contraction hierarchies can be up to seven orders of magnitude faster.
Dijkstra's algorithm is commonly used on graphs where the edge weights are positive integers or real numbers. It can be generalized to any graph where the edge weights are partially ordered, provided the subsequent labels (a subsequent label is produced when traversing an edge) are monotonically non-decreasing.

The idea is to generate a SPT (shortest path tree) with a given source as a root. Maintain an Adjacency Matrix with two sets,

- one set contains vertices included in the shortest-path tree,
- other set includes vertices not yet included in the shortest-path tree.

At every step of the algorithm, find a vertex that is in the other set (set not yet included) and has a minimum distance from the source.



Naive Algorithm

Initialization:

Create a distance[] array to store the shortest distances from the source vertex to all other vertices. Initialize all distances as infinity, except the source, which is set to 0.

Create a visited[] array to track whether a vertex has been processed (initialized to false for all vertices).

Relaxation:

For each vertex in the graph:

Find the unvisited vertex with the smallest distance (requires a linear scan over all vertices).

Mark it as visited.

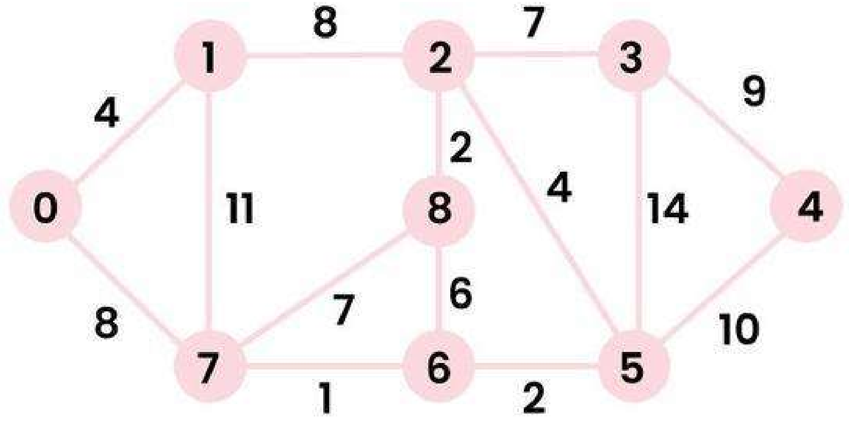
For each unvisited neighbor of the current vertex, update its distance if a shorter path is found.

Repeat:

Continue this process until all vertices have been visited or the smallest distance in distance[] is infinity (indicating disconnected vertices).

Algorithm

- Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
- Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE . Assign the distance value as 0 for the source vertex so that it is picked first.
- While sptSet doesn't include all vertices
- Pick a vertex u that is not there in sptSet and has a minimum distance value.
- Include u to sptSet .
- Then update the distance value of all adjacent vertices of u .
- To update the distance values, iterate through all adjacent vertices.
- For every adjacent vertex v, if the sum of the distance value of u (from source) and weight of edge u-v , is less than the distance value of v , then update the distance value of v .
- Note: We use a boolean array sptSet[] to represent the set of vertices included in SPT . If a value sptSet[v] is true, then vertex v is included in SPT , otherwise not. Array dist[] is used to store the shortest distance values of all vertices.



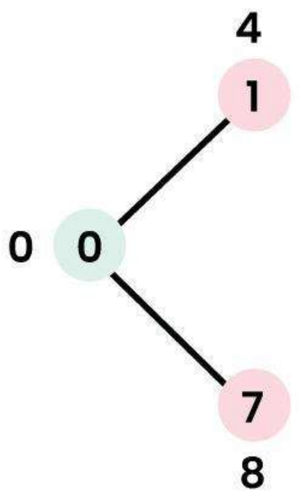
Step 1:

The set sptSet is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite.

Now pick the vertex with a minimum distance value. The vertex 0 is picked, include it in sptSet . So sptSet becomes {0}. After including 0 to sptSet , update distance values of its adjacent vertices.

Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8.

The following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.

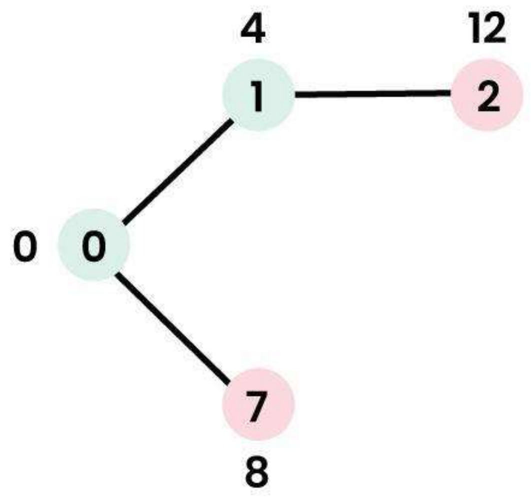


Step 2:

Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet .

So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1.

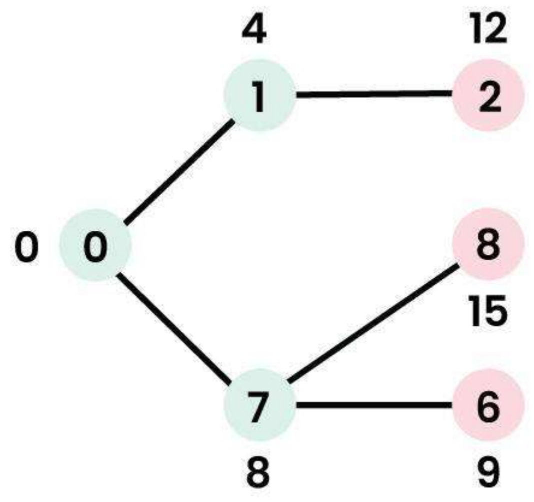
The distance value of vertex 2 becomes 12 .



Step 3:

Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}.

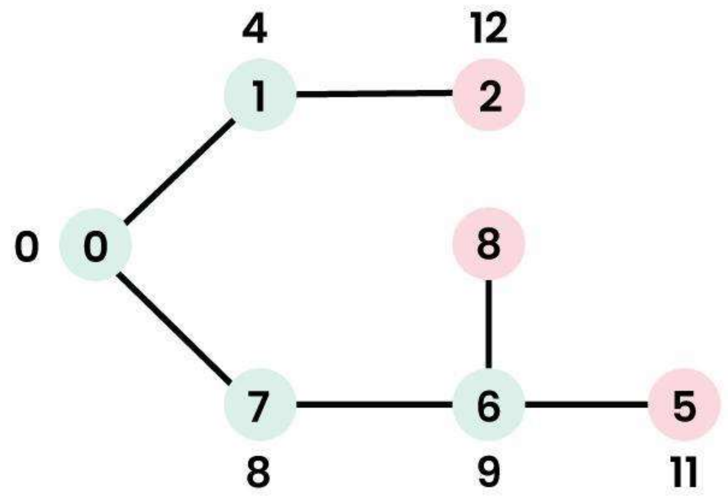
Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



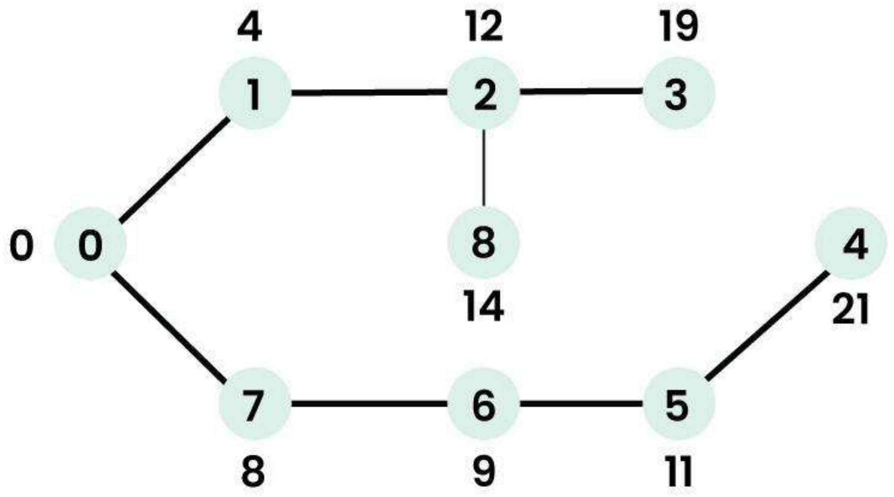
Step 4:

Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6} .

Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until sptSet includes all vertices of the given graph. Finally, we get the following Shortest Path Tree (SPT).


4
12
19
1
2
3
0
0
8
4
14
21
7
6
5
8
9
11

Program:

```
// 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\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's code
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 } };

    // Function call
    dijkstra(graph, 0);

    return 0;
}
```

Output:

```
Vertex      Distance from Source
0            0
1            4
2            12
3            19
4            21
5            11
6            9
7            8
8            14
```

Applications of Dijkstra’s Algorithm:

- Google maps uses Dijkstra algorithm to show shortest distance between source and destination.
- In computer networking , Dijkstra’s algorithm forms the basis for various routing protocols, such as OSPF (Open Shortest Path First) and IS-IS (Intermediate System to Intermediate System).
- Transportation and traffic management systems use Dijkstra’s algorithm to optimize traffic flow, minimize congestion, and plan the most efficient routes for vehicles.
- Airlines use Dijkstra’s algorithm to plan flight paths that minimize fuel consumption, reduce travel time.
- Dijkstra’s algorithm is applied in electronic design automation for routing connections on integrated circuits and very-large-scale integration (VLSI) chips.

Conclusion for Dijkstra's Algorithm

Dijkstra's algorithm is a fundamental graph traversal method for finding the shortest path from a source vertex to all other vertices in a graph. It is widely used due to its correctness and applicability to various real-world scenarios, such as routing, navigation, and network optimization.

Key points:

It guarantees the shortest path for graphs with non-negative edge weights.

The algorithm’s efficiency can be significantly improved with advanced data structures like a priority queue.

While it works well for smaller or sparsely connected graphs, its performance can degrade for large and dense graphs.

However, Dijkstra's algorithm is not suitable for graphs with negative edge weights; in such cases, algorithms like Bellman-Ford are preferred.

Future Scope and Enhancements

Handling Negative Weights:

Extend the algorithm's applicability to graphs with negative weights using techniques like Johnson's algorithm (which combines Dijkstra and Bellman-Ford).

Parallel and Distributed Computing:

Implement parallel versions of Dijkstra's algorithm for faster computation on large-scale graphs, especially in distributed systems.

Dynamic Graphs:

Enhance the algorithm to adapt dynamically to graph changes (e.g., edge updates or vertex insertions), which is crucial for dynamic routing in networks.

Heuristic Approaches:

Integrate heuristic methods like A* search, which combines Dijkstra's principles with heuristics for improved performance in specific applications like pathfinding in games.

Graph Representation:

Optimize graph representation (e.g., adjacency list or sparse matrix) to reduce memory overhead and processing time.

AI and Machine Learning:

Use machine learning models to predict edge weights or recommend initial shortest paths for more efficient solutions.

Real-World Applications:

Further explore its applications in fields like robotics (path planning), logistics (route optimization), and energy grids (network flow analysis).