Dijkstra

Step 1: Libraries and define constants

```
#include <iostream>
#include <climits>
using namespace std;

const int MAX = 100;        // Maximum number of vertices
const int INF = INT_MAX;    // Represents "infinity" for initial distances
```

Purpose:

- MAX limits the graph size.
- INF marks unreachable paths initially.

Step 2: Declare global structures

```
int graph[MAX][MAX];
int parent[MAX];
```

Purpose:

- graph[i][j]: weight of edge between i and j (or INF if no edge).
- parent[v]: keeps track of the previous vertex in the shortest path.

Step 3: Initialize graph

```
void initGraph(int n)
{
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
         graph[i][j] = (i == j) ? 0 : INF;
}
```

Purpose:

- Diagonal = 0 (distance to itself).
- Non-diagonal = INF (no edge initially).

Step 4: Add edges

```
void addEdge(int u, int v, int w)
{
   graph[u][v] = w;
   graph[v][u] = w; // undirected graph
}
```

Purpose:

- Insert an edge (u,v) with weight w.
- Works both ways since the graph is undirected.

Step 5: Find unvisited vertex with minimum distance

```
int minDistance(int dist[], bool visited[], int n)
{
   int minVal = INF;
   int minIndex = -1;
   for (int v = 0; v < n; v++)
   {
      if (visited[v] == false && dist[v] < minVal)
      {
         minVal = dist[v];
         minIndex = v;
      }
   }
   return minIndex;
}
```

Purpose:

- Helps Dijkstra pick the “next best” vertex to explore.

Step 6: Dijkstra’s Algorithm

```
void dijkstra(int src, int n)
{
   int dist[MAX];      // shortest distance estimates
   bool visited[MAX];  // visited flag

   // Step 6a: Initialize
   for (int i = 0; i < n; i++)
   {
      dist[i] = INF;
      visited[i] = false;
      parent[i] = -1;
   }
   dist[src] = 0; // distance to source = 0

   // Step 6b: Relax vertices one by one
   for (int count = 0; count < n - 1; count++)
   {
      int u = minDistance(dist, visited, n);
```

```
        if (u == -1) break;  // No more reachable nodes

        visited[u] = true;

        for (int v = 0; v < n; v++)
        {
            if (visited[v] == false && graph[u][v] != INF &&
                dist[u] + graph[u][v] < dist[v])
            {
                dist[v] = dist[u] + graph[u][v];
                parent[v] = u;
            }
        }
    }

    // Step 6c: Print results
    cout << "Shortest distances from source " << src << ":\n";
    for (int i = 0; i < n; i++)
        cout << "To " << i << " = " << dist[i] << "\n";
}
```

Purpose:

- Step 6a: Initialize distances and parents.
- Step 6b:
  - ➥ Pick the unvisited vertex with the smallest distance.
  - ➥ Update its neighbors if a shorter path is found.
- Step 6c: Print final shortest distances.

Step 7: Path reconstruction

```
void printPath(int dest)
{
    if (parent[dest] == -1)
    {
        cout << dest << " ";
        return;
    }
    printPath(parent[dest]);
    cout << dest << " ";
}
```

Step 8: Main function

```
int main()
{
    int n = 5;
    initGraph(n);

    // Step 8a: Add edges
    addEdge(0, 1, 10);
    addEdge(0, 4, 5);
    addEdge(1, 2, 1);
    addEdge(1, 4, 2);
    addEdge(2, 3, 4);
    addEdge(3, 0, 7);
    addEdge(3, 2, 6);
    addEdge(4, 2, 9);
    addEdge(4, 3, 2);

    // Step 8b: Run Dijkstra
    int src = 0;
    dijkstra(src, n);

    // Step 8c: Print path to destination
    int dest = 3;
    cout << "\nPath from " << src << " to " << dest << ": ";
    printPath(dest);

    cout << "\nCost = (see above output)\n";

    return 0;
}
```

Purpose:

inprotected.com