LINK OF THE GITHUB REPO USED: **https://gist.github.com/zayednews/72c04dcb84118d9628d879d18382a336#file-dijkstra-cpp-L172**  
The provided code implements a weighted directed graph using an adjacency matrix and includes methods for adding edges, removing edges, displaying the graph, and finding the shortest path using Dijkstra's algorithm. Let's analyze and document the code structure and functionality of each function.

**Class: Graph**

**Attributes:**

* int \*\*adjMatrix: A 2D array representing the adjacency matrix of the graph, where adjMatrix[i][j] holds the cost/weight of the edge from vertex i to vertex j.
* int vertexCount: The number of vertices in the graph.
* int \*key: An array to keep track of whether a vertex is included in the Minimum Spanning Tree (MST). (0 means not included, 1 means included).
* int \*distance: An array to store the shortest distance from the start node to each vertex.
* int \*parent: An array to store the parent of each vertex in the shortest path tree.

**Methods:**

1. **Constructor: Graph(int vertexCount)**
   * Initializes the graph with vertexCount vertices.
   * Allocates memory for the adjacency matrix and initializes all edges to 0 (no edge).
   * Allocates memory for key, distance, and parent arrays.
2. **Destructor: ~Graph()**
   * Deallocates the memory used by the adjacency matrix and other arrays.
3. **void addEdgeDirectedWeight(int i, int j, int cost)**
   * Adds a directed edge with weight cost from vertex i to vertex j.
   * Validates the vertex indices to ensure they are within the graph's bounds.
4. **void removeEdgeUndirected(int i, int j)**
   * Removes an undirected edge between vertices i and j by setting the corresponding entries in the adjacency matrix to 0.
5. **int isEdge(int i, int j)**
   * Checks if there is an edge from vertex i to vertex j.
   * Returns the weight of the edge if it exists, otherwise returns 0.
   * Prints an error message if the vertex indices are invalid.
6. **void display()**
   * Displays the adjacency matrix of the graph, showing the weights of the edges between vertices.
7. **void initializeState()**
   * Initializes the key, distance, and parent arrays.
   * Sets key to 0 (not in MST), distance to INT\_MAX (infinity), and parent to -1 (no parent).
8. **void showBasicInfo()**
   * Displays the basic information for each vertex, including its key, distance, and parent.
9. **int isAllKeyTrue()**
   * Checks if all vertices are included in the MST.
   * Returns 1 if all vertices are included, otherwise returns 0.
10. **int findMinDistanceNode()**
    * Finds the vertex with the minimum distance that is not yet included in the MST.
    * Returns the index of the vertex with the minimum distance.
11. **void Dijkstra(int startNode)**
    * Implements Dijkstra's algorithm to find the shortest paths from the startNode to all other vertices.
    * Initializes the distance of the startNode to 0.
    * Repeatedly finds the vertex with the minimum distance that is not yet included in the MST, marks it as included, and updates the distances of its adjacent vertices using relaxation.

**Main Function:**

1. **int main()**
   * Creates a graph g with 5 vertices.
   * Adds directed edges with specified weights between the vertices.
   * Displays the adjacency matrix of the graph.
   * Initializes the state arrays (key, distance, parent).
   * Calls Dijkstra's algorithm from the start node 0 to find the shortest paths.
   * Returns 0 to indicate successful execution.

**Summary**

* The code defines a Graph class that represents a directed weighted graph using an adjacency matrix.
* It includes methods for adding and removing edges, displaying the graph, initializing state arrays, and finding the shortest paths using Dijkstra's algorithm.
* The main function demonstrates the creation of a graph, addition of edges, and finding the shortest paths from a specified start node.

This comprehensive structure and detailed documentation provide a clear understanding of how each part of the code functions and interacts with others.