***Function used in the graph implementation with their definition***

**void bfs(int adj[][no\_vertices], int start)**

**{**

**FILE \*BFS = fopen("BFS.txt", "w");**

**int \*visited, \*queue;**

**int front = 0, rear = -1;**

**visited = (int\*)calloc(no\_vertices, sizeof(int));**

**queue = (int\*)malloc(no\_vertices \* sizeof(int));**

* **FILE \*BFS = fopen("BFS.txt", "w");**: Opens a file named "BFS.txt" in write mode. It creates a file pointer **BFS** to write the BFS traversal results.
* **int \*visited, \*queue;**: Declares two integer pointers **visited** and **queue** to keep track of visited vertices and store the vertices to be processed in the BFS algorithm.
* **int front = 0, rear = -1;**: Initializes two variables **front** and **rear** for the front and rear indices of the queue.
* **visited = (int\*)calloc(no\_vertices, sizeof(int));**: Allocates dynamic memory using **calloc** to create an array **visited** of size **no\_vertices** (number of vertices) and initializes all elements to 0. This array will track which vertices have been visited during the BFS traversal.
* **queue = (int\*)malloc(no\_vertices \* sizeof(int));**: Allocates dynamic memory using **malloc** to create an array **queue** of size **no\_vertices** (number of vertices) to store the vertices to be processed in the BFS traversal.

queue[++rear] = start;

visited[start] = 1;

fprintf(BFS, "BFS: ", start);

* **queue[++rear] = start;**: Adds the **start** vertex to the queue by incrementing **rear** first and then assigning **start** to **queue[rear]**.
* **visited[start] = 1;**: Marks the **start** vertex as visited by setting **visited[start]** to 1.
* **fprintf(BFS, "BFS: ", start);**: Writes "BFS: " followed by the **start** vertex to the file **BFS** to indicate the starting vertex of the BFS traversal.

while (front <= rear)

{

start = queue[front++];

printf("%d\t", start);

fprintf(BFS, " %d\t", start);

for (int i = 0; i < no\_vertices; i++)

{

if (adj[start][i] == 1 && visited[i] != 1)

{

queue[++rear] = i;

visited[i] = 1;

}

}

}

* **while (front <= rear)**: Executes the following code block while there are still vertices in the queue to be processed.
* **start = queue[front++];**: Dequeues a vertex from the queue and assigns it to **start** by first accessing **queue[front]** and then incrementing **front**.
* **printf("%d\t", start);**: Prints the value of **start** to the console.
* **fprintf(BFS, " %d\t", start);**: Writes the value of **start** followed by a tab to the file **BFS** to record the BFS traversal sequence.
* **for (int i = 0; i < no\_vertices; i++)**: Iterates over all vertices in the graph.
* **if (adj[start][i] == 1 && visited[i] != 1)**: Checks if there is an edge between **start** and vertex **i** in the adjacency matrix **adj** and if vertex **i** has not been visited yet.
* **queue[++rear] = i;**: Enqueues vertex **i** by incrementing **rear** first and then assigning **i** to **queue[rear]**.
* **visited[i] = 1;**: Marks vertex **i** as visited.

**free(visited);**

**free(queue);**

**fclose(BFS);**

**}**

* **free(visited);**: Frees the memory allocated for the **visited** array.
* **free(queue);**: Frees the memory allocated for the **queue** array.
* **fclose(BFS);**: Closes the file **BFS** after writing the BFS traversal results.

This code performs a BFS traversal on the given graph represented by the adjacency matrix **adj** starting from the **start** vertex. It writes the BFS traversal sequence to a file named "BFS.txt" and prints it to the console. The **visited** array and **queue** array are used to keep track of visited vertices and the vertices to be processed in the BFS traversal, respectively.

**void dfs(int adj[][no\_vertices], int visited[], int start)**

**{**

**printf("%d\t", start);**

**visited[start] = 1;**

**for (int i = 0; i < no\_vertices; i++)**

**{**

**if (visited[i] != 1 && adj[start][i] == 1)**

**{**

**dfs(adj, visited, i);**

**}**

**}**

**}**

* **void dfs(int adj[][no\_vertices], int visited[], int start)**: This line declares the **dfs** function, which takes three parameters: **adj** (the adjacency matrix representing the graph), **visited** (an array to track visited vertices), and **start** (the starting vertex for the DFS traversal).
* **printf("%d\t", start);**: This line prints the value of the **start** vertex to the console. It represents visiting the current vertex during the DFS traversal.
* **visited[start] = 1;**: This line marks the **start** vertex as visited by setting **visited[start]** to 1 in the **visited** array.
* **for (int i = 0; i < no\_vertices; i++)**: This line starts a loop to iterate over all vertices in the graph.
* **if (visited[i] != 1 && adj[start][i] == 1)**: This line checks if vertex **i** has not been visited yet (**visited[i] != 1**) and if there is an edge between the **start** vertex and vertex **i** in the adjacency matrix (**adj[start][i] == 1**).
* **dfs(adj, visited, i);**: This line recursively calls the **dfs** function to perform DFS traversal starting from vertex **i**. This is done to explore the unvisited adjacent vertices of the **start** vertex.

The **dfs** function follows a recursive approach to explore the graph depth-first. It starts by visiting the **start** vertex, marks it as visited, and then recursively explores the unvisited adjacent vertices by calling **dfs** on each of them. This process continues until all reachable vertices from the **start** vertex have been visited. The **visited** array keeps track of visited vertices to avoid revisiting them.

**FLOYD\_WARSHALL ALGORITHM FOR SHORTEST PATH**

The Floyd-Warshall algorithm is a dynamic programming algorithm used to find the shortest path between all pairs of vertices in a weighted directed graph. It is named after Robert Floyd and Stephen Warshall, who independently developed the algorithm in 1962.

The algorithm works by iteratively considering all vertices as potential intermediate vertices along the path between two vertices and updating the shortest path if a shorter path is found. Here's the basic idea of the Floyd-Warshall algorithm:

1. Initialize a 2D matrix called the "distance matrix" with the initial distances between vertices. If there is an edge between vertices i and j, the distance matrix should store the weight of that edge. If there is no edge, the distance should be set to infinity.
2. Perform a series of iterations, considering all vertices as potential intermediate vertices in the path between any pair of vertices.
3. For each iteration, update the distance matrix by considering the possibility of going through the current intermediate vertex. If the distance from vertex i to vertex j through the intermediate vertex k is shorter than the current distance between i and j, update the distance matrix accordingly.
4. After all iterations are completed, the distance matrix will contain the shortest distances between all pairs of vertices.
5. If you also need to reconstruct the actual shortest paths, you can use an additional matrix called the "next matrix" to store the next vertex on the shortest path from i to j. This matrix is updated during the iterations to keep track of the intermediate vertices along the shortest paths

**function floydWarshall(graph):**

**n = number of vertices in the graph**

**distanceMatrix = initializeDistanceMatrix(graph)**

**nextMatrix = initializeNextMatrix(graph)**

**for k from 1 to n:**

**for i from 1 to n:**

**for j from 1 to n:**

**if distanceMatrix[i][j] > distanceMatrix[i][k] + distanceMatrix[k][j]:**

**distanceMatrix[i][j] = distanceMatrix[i][k] + distanceMatrix[k][j]**

**nextMatrix[i][j] = nextMatrix[i][k]**

**return distanceMatrix, nextMatrix**

**void floydWarshall(int adj[][no\_vertices], int numVertices, int source, int destination)**

**{**

**int distance[no\_vertices][no\_vertices];**

**int i, j, k;**

**FILE \*path = fopen("Shortest Path.txt", "w");**

**fclose(path);**

**path = fopen("Shortest Path.txt", "a");**

* **void floydWarshall(int adj[][no\_vertices], int numVertices, int source, int destination)**: This line declares the **floydWarshall** function, which takes four parameters: **adj** (the adjacency matrix representing the graph), **numVertices** (the number of vertices in the graph), **source** (the source vertex), and **destination** (the destination vertex).
* **int distance[no\_vertices][no\_vertices];**: This line declares a 2D array **distance** to store the shortest path distances between all pairs of vertices.
* **FILE \*path = fopen("Shortest Path.txt", "w");**: This line opens a file named "Shortest Path.txt" in write mode. It creates a file pointer **path** to write the shortest path distances.
* **fclose(path);**: This line closes the file **path** to prepare it for writing.
* **path = fopen("Shortest Path.txt", "a");**: This line opens the file **path** again, but in append mode, to continue writing to it.

**for (i = 0; i < numVertices; i++) {**

**for (j = 0; j < numVertices; j++) {**

**if (adj[i][j] == 0)**

**distance[i][j] = INT\_MAX;**

**else**

**distance[i][j] = adj[i][j];**

**}**

**}**

These nested loops initialize the **distance** array by copying the weights from the **adj** matrix. If there is no direct edge between two vertices (**adj[i][j] == 0**), the corresponding **distance** is set to **INT\_MAX** to represent infinity.

**for (k = 0; k < numVertices; k++) {**

**for (i = 0; i < numVertices; i++) {**

**for (j = 0; j < numVertices; j++) {**

**if (distance[i][k] != INT\_MAX && distance[k][j] != INT\_MAX &&**

**distance[i][k] + distance[k][j] < distance[i][j]) {**

**distance[i][j] = distance[i][k] + distance[k][j];**

**}**

**}**

**}**

**}**

These nested loops perform the main steps of the Floyd-Warshall algorithm. They iteratively update the **distance** array by considering all possible intermediate vertices **k**. If the path from vertex **i** to vertex **j** through vertex **k** is shorter than the current shortest path (**distance[i][k] + distance[k][j] < distance[i][j]**), the **distance** array is updated with the new shorter distance.

**if (distance[source][destination] == INT\_MAX) {**

**printf("No path from node %d to node %d\n", source, destination);**

**fprintf(path, "No path from node %d to node %d\n", source, destination);**

**} else {**

**printf("Shortest distance from node %d to node %d: %d\n", source, destination, distance[source][destination]);**

**fprintf(path, "Shortest distance from node %d to node %d: %d\n", source, destination, distance[source][destination]);**

**}**

* **if (distance[source][destination] == INT\_MAX)**: This condition checks if the shortest path distance between the **source** and **destination** nodes is equal to **INT\_MAX**. If it is, it means that there is no path between the nodes.
* **printf("No path from node %d to node %d\n", source, destination);**: If there is no path between the nodes, this line prints a message indicating that there is no path from **source** to **destination**.
* **fprintf(path, "No path from node %d to node %d\n", source, destination);**: This line writes the same message to the file **path** to record that there is no path from **source** to **destination**.
* **else**: If there is a path between the nodes, this block of code is executed.
* **printf("Shortest distance from node %d to node %d: %d\n", source, destination, distance[source][destination]);**: This line prints the shortest distance between the **source** and **destination** nodes.
* **fprintf(path, "Shortest distance from node %d to node %d: %d\n", source, destination, distance[source][destination]);**: This line writes the same shortest distance to the file **path** for recording purposes.
* **fclose(path);**: This line closes the file **path** after writing the necessary information. It ensures that the file is properly closed and resources are released.

Overall, this code snippet checks if there is a path between two nodes in the graph and prints/writes the corresponding shortest distance or a message indicating that there is no path.

**while(info.s!=-1 && info.d!=-1)**

**{**

**printf("Enter an Edge from node (0 to %d) to node (0 to %d): ", no\_vertices, no\_vertices);**

**scanf("%d%d", &info.s, &info.d);**

**adj[info.s][info.d] = 1;**

**adj[info.d][info.s] = 1;**

**}**

* **while(info.s!=-1 && info.d!=-1)**: This line initiates a while loop that continues until the user enters **-1** for both **info.s** and **info.d**. This condition is used to exit the loop.
* **printf("Enter an Edge from node (0 to %d) to node (0 to %d): ", no\_vertices, no\_vertices)**: This line prompts the user to enter an edge by specifying the range of valid node indices (**0** to **no\_vertices - 1**).
* **scanf("%d%d", &info.s, &info.d)**: This line reads two integers from the user, representing the starting node (**info.s**) and the destination node (**info.d**) of the edge.
* **adj[info.s][info.d] = 1;**: This line sets the adjacency matrix element at position **[info.s][info.d]** to **1**, indicating the existence of a directed edge from node **info.s** to node **info.d**.
* **adj[info.d][info.s] = 1;**: This line sets the adjacency matrix element at position **[info.d][info.s]** to **1**, indicating the existence of a directed edge from node **info.d** to node **info.s**. This line ensures that the graph is undirected (if an edge is created from **info.s** to **info.d**, there is also an edge created from **info.d** to **info.s**).

The code snippet allows the user to enter edges between nodes in the graph by specifying the node indices. The adjacency matrix **adj** is updated accordingly, with **1** indicating the presence of an edge between the specified nodes. The loop continues until the user enters **-1** for both the starting and destination nodes, allowing multiple edges to be entered consecutively.