Floyd-Warshall Algorithm

Aim:
The aim of the program is to implement the Floyd-Warshall algorithm to find the shortest paths between all pairs of vertices in a weighted graph. It seeks to efficiently compute the shortest distances between every pair of vertices, even in the presence of negative edge weights, thus facilitating network optimization and pathfinding in various applications.

Theory:
The Floyd-Warshall algorithm is a dynamic programming-based approach to find the shortest paths between all pairs of vertices in a weighted graph. It works efficiently for dense graphs, where the number of edges is close to the square of the number of vertices. The algorithm iteratively constructs a matrix of shortest distances between all pairs of vertices, updating the entries based on intermediate vertices.

Code:
```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
struct Edge {
int src, dest, weight;
};
int** allocateMatrix(int V) {
int **matrix = (int **)malloc(V * sizeof(int *));
for (int i = 0; i < V; i++) {
matrix[i] = (int *)malloc(V * sizeof(int));
for (int j = 0; j < V; j++)
matrix[i][j] = (i == j) ? 0 : INT_MAX;
}
return matrix;
}
void printMatrix(int **matrix, int V, const char *name) {
printf("%s:\n", name);
for (int i = 0; i < V; i++) {
for (int j = 0; j < V; j++) {
if (matrix[i][j] == INT_MAX)
printf("INF\t");
else
printf("%d\t", matrix[i][j]);
}
printf("\n");
}
printf("\n");
}
void floydWarshall(int **graph, int V) {
int **dist = allocateMatrix(V); // Output matrix that will have the shortest distances between every pair of vertices
for (int i = 0; i < V; i++)
for (int j = 0; j < V; j++)
dist[i][j] = graph[i][j];
for (int k = 0; k < V; k++) {
printMatrix(dist, V, "Intermediate Matrix");
```

```
for (int i = 0; i < V; i++) {
for(int j = 0; j < V; j++) {
if (dist[i][k] != INT_MAX && dist[k][j] != INT_MAX && dist[i][k] + dist[k][j] < dist[i][j])
dist[i][j] = dist[i][k] + dist[k][j];
}
}
}
printMatrix(dist, V, "Final Shortest Distance Matrix");
for (int i = 0; i < V; i++)
free(dist[i]);
free(dist);
}
void addEdges(int **graph, struct Edge edges[], int E) {
for (int i = 0; i < E; i++) {
graph[edges[i].src][edges[i].dest] = edges[i].weight;
}
}
int main() {
int V = 4;
int E = 6;
int **graph = allocateMatrix(V);
struct Edge edges[] = {
{0,1,-1},
{0,2,4},
{1,2,3},
{1,3,2},
{1,4,2},
{3,2,5},
{3,1,1},
{4,3,-3}
};
addEdges(graph, edges, E);
printf("Initial adjacency matrix:\n");
printMatrix(graph, V, "Initial Adjacency Matrix");
floydWarshall(graph, V);
for (int i = 0; i < V; i++)
free(graph[i]);
free(graph);
return 0;
}
```

Output:

```
Initial Adjacency Matrix:
0       -1      4       INF
INF     0       3       2
INF     INF     0       INF
INF     INF     5       0

Intermediate Matrix:
0       -1      4       INF
INF     0       3       2
INF     INF     0       INF
INF     INF     5       0

Intermediate Matrix:
0       -1      4       INF
INF     0       3       2
INF     INF     0       INF
INF     INF     5       0

Intermediate Matrix:
0       -1      2       1
INF     0       3       2
INF     INF     0       INF
INF     INF     5       0

Intermediate Matrix:
0       -1      2       1
INF     0       3       2
INF     INF     0       INF
INF     INF     5       0

Final Shortest Distance Matrix:
0       -1      2       1
INF     0       3       2
INF     INF     0       INF
INF     INF     5       0
```

Algorithm Steps:

1. Initialize Distance Matrix:
- Create a distance matrix `dist[][]` of size V x V, where V is the number of vertices in the graph.
- Initialize the elements of the matrix such that `dist[i][j]` represents the weight of the edge from vertex i to vertex j if it exists, or infinity if there's no direct edge.

2. Update Distance Matrix with Direct Edges:
- For each edge (u, v) with weight w in the graph:
  - Update `dist[u][v]` to be w, representing the direct edge weight from vertex u to vertex v.

3. Floyd-Warshall Algorithm Execution:
- Iterate over each vertex 'k' from 0 to V-1 as an intermediate vertex:
  - For each pair of vertices 'i' and 'j' (0 <= i, j < V):
    - Update the shortest distance from vertex 'i' to vertex 'j' using vertex 'k' as an intermediate vertex:
      - `dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])`

- If the distance from 'i' to 'k' plus the distance from 'k' to 'j' is less than the current distance from 'i' to 'j', update the distance accordingly.

4. Final Result:
- After completing all iterations, the `dist[][]` matrix contains the shortest distances between all pairs of vertices.
- The elements `dist[i][j]` represent the shortest distance from vertex 'i' to vertex 'j'.

5. Negative Cycle Detection:
- Check for negative cycles by examining the diagonal elements of the `dist[][]` matrix.
- If any diagonal element `dist[i][i]` is negative, it indicates the presence of a negative cycle reachable from vertex 'i'.

6. Output or Utilization:
- Utilize the `dist[][]` matrix as needed, such as finding shortest paths between specific pairs of vertices or analyzing network properties.

7. Memory Deallocation:
- Free up memory allocated for the distance matrix after its use.

Time complexity:

1. Initialization of Distance Matrix:
- Initializing the distance matrix `dist[][]` takes O(V^2) time as it involves filling a V x V matrix.

2. Floyd-Warshall Algorithm Execution:
- The outer loop iterates V times (for each intermediate vertex).
- Within each iteration of the outer loop, there are two nested loops to update the distances:
  - The inner loops iterate V^2 times (for each pair of vertices).
  - Within the inner loops, updating each distance involves constant time operations.
- Therefore, the total time complexity of the algorithm for this part is O(V^3).

3. Overall Time Complexity:
- Combining the time complexities of initialization and the Floyd-Warshall algorithm execution, the dominant term is O(V^3).
- Thus, the overall time complexity of the Floyd-Warshall algorithm is O(V^3).

Space complexity:

1. Distance Matrix:
- The main component of space consumption is the distance matrix `dist[][]`, which is of size V x V.
- Therefore, the space required for the distance matrix is O(V^2).

2. Auxiliary Variables:
- Apart from the distance matrix, the algorithm uses additional variables such as loop counters and temporary variables.
- These variables typically require a constant amount of space, regardless of the size of the input graph.
- Thus, the space complexity contributed by auxiliary variables is O(1).

3. Total Space Complexity:

- Combining the space required for the distance matrix and auxiliary variables, the dominant term is O(V^2).
- Therefore, the overall space complexity of the Floyd-Warshall algorithm is O(V^2).