The **Floyd-Warshall algorithm in C** is a graph algorithm that finds the shortest path between two vertices in a graph in a weighted graph with positive or negative edge weights but without negative cycles. The algorithm is named after the British mathematician **Floyd Warshall**. The algorithm is also known as the **all-pairs shortest path algorithm**.

The algorithm compares all possible paths between two vertices in a graph and finds the shortest path. It does so in O(V3) time even when the graph is sparse.

In this algorithm, we will use a matrix to represent the graph. The matrix will have the following structure:

[

[0, 1, 2, 3, 4],

[1, 0, 5, 6, 7],

[2, 5, 0, 8, 9],

[3, 6, 8, 0, 10],

[4, 7, 9, 10, 0]

]

Then, we will use the following formula to find the shortest path between two vertices:

**d[i][j] = min(d[i][j], d[i][k] + d[k][j])**

**Problem Description**

Write a C program that finds the shortest path between two vertices in a graph and prints the shortest path matrix using Floyd Warshall Algorithm.

**Problem Solution**

1. Ask the user to enter the edges of the graph as a matrix representation.  
2. Print the original matrix.  
3. Pass the matrix to the function floydWarshall to find the shortest path matrix.  
4. Print the shortest path matrix.

**Methods used:**

* **void floydWarshall(int \*\*, int)** – This function will find the shortest path between two vertices in a graph using the Floyd-Warshall algorithm. The parameters are the graph represented as a matrix and the number of rows.

**Program/Source Code**

Here is the source code of a C Program that will find the shortest path between two vertices in a graph using the Floyd-Warshall algorithm. The C program is successfully compiled and run on a Linux system. The program output is also shown below.

1. */\**
2. *\* C Program to find the shortest path between two vertices in a graph*
3. *\* using the Floyd-Warshall algorithm*
4. *\*/*
6. #include <stdio.h>
7. #include <stdlib.h>
9. void floydWarshall(int \*\*graph, int n)
10. {
11. int i, j, k;
12. for (k = 0; k < n; k++)
13. {
14. for (i = 0; i < n; i++)
15. {
16. for (j = 0; j < n; j++)
17. {
18. if (graph[i][j] > graph[i][k] + graph[k][j])
19. graph[i][j] = graph[i][k] + graph[k][j];
20. }
21. }
22. }
23. }
25. int main(void)
26. {
27. int n, i, j;
28. printf("Enter the number of vertices: ");
29. scanf("%d", &n);
30. int \*\*graph = (int \*\*)malloc((long unsigned) n \* sizeof(int \*));
31. for (i = 0; i < n; i++)
32. {
33. graph[i] = (int \*)malloc((long unsigned) n \* sizeof(int));
34. }
35. for (i = 0; i < n; i++)
36. {
37. for (j = 0; j < n; j++)
38. {
39. if (i == j)
40. graph[i][j] = 0;
41. else
42. graph[i][j] = 100;
43. }
44. }
45. printf("Enter the edges: **\n**");
46. for (i = 0; i < n; i++)
47. {
48. for (j = 0; j < n; j++)
49. {
50. printf("[%d][%d]: ", i, j);
51. scanf("%d", &graph[i][j]);
52. }
53. }
54. printf("The original graph is:**\n**");
55. for (i = 0; i < n; i++)
56. {
57. for (j = 0; j < n; j++)
58. {
59. printf("%d ", graph[i][j]);
60. }
61. printf("**\n**");
62. }
63. floydWarshall(graph, n);
64. printf("The shortest path matrix is:**\n**");
65. for (i = 0; i < n; i++)
66. {
67. for (j = 0; j < n; j++)
68. {
69. printf("%d ", graph[i][j]);
70. }
71. printf("**\n**");
72. }
73. return 0;
74. }

**Program Explanation**

1. The program begins with asking the user to input the edges of the graph and represents them as a matrix representation.  
2. Pass the matrix and the number of vertices **n** as input to the function **floydWarshall**.  
3. In the function, loop nest from 0 to size and inside that make another loop from 0 to size where we compare the weights respectively as seen in the program.  
4. Pass the matrix to the function floydWarshall to find the shortest path matrix.  
5. Print the shortest path matrix.

**Time complexity: O(V3)**  
The time complexity of the algorithm is O(V3), where V is the number of vertices in the graph.

**Space Complexity: O(V2)**  
The space complexity of the algorithm is O(V2), where V is the number of vertices in the graph.

**Runtime Test Cases**

In this case, we enter “4” as the number of vertices as input to find the shortest path between two vertices in the graph using the Floyd-Warshall algorithm.

Enter the number of vertices: 4

Enter the edges:

[0][0]: 0

[0][1]: 12

[0][2]: 45

[0][3]: 2

[1][0]: 1

[1][1]: 0

[1][2]: 45

[1][3]: 32

[2][0]: 77

[2][1]: 43

[2][2]: 0

[2][3]: 2

[3][0]: 42

[3][1]: 3

[3][2]: 88

[3][3]: 0

The original graph is:

0 12 45 2

1 0 45 32

77 43 0 2

42 3 88 0

The shortest path matrix is:

0 5 45 2

1 0 45 3

6 5 0 2

4 3 48 0

# 

# Comparison of Dijkstra’s and Floyd–Warshall algorithms

**Main Purposes:**

* [Dijkstra’s Algorithm](https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/) is one example of a single-source shortest or SSSP algorithm, i.e., given a source vertex it finds shortest path from source to all other vertices.
* [Floyd Warshall Algorithm](https://www.geeksforgeeks.org/dynamic-programming-set-16-floyd-warshall-algorithm/) is an example of all-pairs shortest path algorithm, meaning it computes the shortest path between all pair of nodes.

**Time Complexities :**

* Time Complexity of Dijkstra’s Algorithm: O(E log V)
* Time Complexity of Floyd Warshall: O(V3)

**Other Points:**

* We can use Dijskstra’s shortest path algorithm for finding all pair shortest paths by running it for every vertex. But time complexity of this would be O(VE Log V) which can go (V3 Log V) in worst case.
* Another important differentiating factor between the algorithms is their working towards distributed systems. Unlike Dijkstra’s algorithm, Floyd Warshall can be implemented in a distributed system, making it suitable for data structures such as Graph of Graphs (Used in Maps).
* Lastly Floyd Warshall works for negative edge but no [negative cycle](https://www.geeksforgeeks.org/detect-negative-cycle-graph-bellman-ford/), whereas Dijkstra’s algorithm don’t work for negative edges.

(Or)

#include<stdio.h>

#define MAX 100

void display(int matrix[MAX][MAX], int n);

int adj[MAX][MAX];

int n;

void create\_graph();

int main()

{

int i,j,k;

int P[MAX][MAX];

create\_graph();

printf("\nThe adjacency matrix is :\n");

display(adj,n);

for(i=0; i<n; i++)

for(j=0; j<n; j++)

P[i][j] = adj[i][j];

for(k=0; k<n; k++)

{

for(i=0; i<n; i++)

for(j=0; j<n; j++)

P[i][j] = ( P[i][j] || ( P[i][k] && P[k][j] ) );

printf("\nP%d is :\n",k);

display(P,n);

}

printf("\nP%d is the path matrix of the given graph\n",k-1);

}/\*End of main() \*/

void display(int matrix[MAX][MAX],int n)

{

int i,j;

for(i=0; i<n; i++)

{

for(j=0; j<n; j++)

printf("%3d",matrix[i][j]);

printf("\n");

}

}/\*End of display()\*/

void create\_graph()

{

int i,max\_edges,origin,destin;

printf("\nEnter number of vertices : ");

scanf("%d",&n);

max\_edges = n\*(n-1);

for( i=1; i<=max\_edges; i++ )

{

printf("\nEnter edge %d( -1 -1 ) to quit : ",i);

scanf("%d %d",&origin,&destin);

if((origin == -1) && (destin == -1))

break;

if( origin >= n || destin >= n || origin<0 || destin<0)

{

printf("\nInvalid edge!\n");

i--;

}

else

adj[origin][destin] = 1;

}/\*End of for\*/

}/\*End of create\_graph()\*/