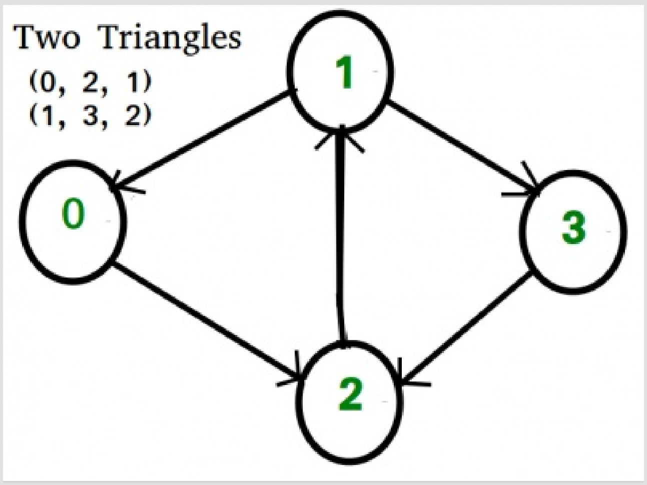
Number of Triangles in Directed and Undirected Graphs

Given a Graph, count number of triangles in it. The graph is can be directed or undirected.

Example:

```
Input: digraph[V][V] = { {0, 0, 1, 0},
                        {1, 0, 0, 1},
                        {0, 1, 0, 0},
                        {0, 0, 1, 0}
                      };
Output: 2
Give adjacency matrix represents following
directed graph.
```



We have discussed a method based on graph trace that works for undirected graphs. In this post a new method is discussed with that is simpler and works for both directed and undirected graphs.

The idea is to use three nested loops to consider every triplet (i, j, k) and check for the above condition (there is an edge from i to j, j to k and k to i)
However in an **undirected graph**, the triplet (i, j, k) can be permuted to give six combination (See previous post for details). Hence we divide the total count by 6 to get the actual number of triangles.
In case of **directed graph**, the number of permutation would be 3 (as order of nodes becomes relevant). Hence in this case the total number of triangles will be obtained by dividing total count by 3. For example consider the directed graph given below

Following is C++ implementation.

```
/* C++ program to count triangles in a graph.  The program is
   for adjacency matrix representation of the graph.*/
#include<bits/stdc++.h>

// Number of vertices in the graph
#define V 4

using namespace std;

// function to calculate the number of triangles in a simple
// directed/undirected graph.
// isDirected is true if the graph is directed, its false otherwise
int countTriangle(int graph[V][V], bool isDirected)
{
    int count_Triangle = 0;  // Initialize result

    // Consider every possible triplet of edges in graph
    for (int i=0; i<V; i++)
    {
        for (int j=0; j<V; j++)
        {
            for (int k=0; k<V; k++)
            {
               // check the triplet if it satisfies the condition
               if (graph[i][j] && graph[j][k] && graph[k][i])
                  count_Triangle++;
             }
        }
    }

    // if graph is directed , division is done by 3
    // else division by 6 is done
    isDirected? count_Triangle /= 3 : count_Triangle /= 6;

    return count_Triangle;
}

//driver function to check the program
int main()
{
    // Create adjacency matrix of an undirected graph
    int graph[][V] = { {0, 1, 1, 0},
                       {1, 0, 1, 1},
                       {1, 1, 0, 1},
                       {0, 1, 1, 0}
                     };

    // Create adjacency matrix of a directed graph
    int digraph[][V] = { {0, 0, 1, 0},
                        {1, 0, 0, 1},
                        {0, 1, 0, 0},
                        {0, 0, 1, 0}
                       };

    cout << "The Number of triangles in undirected graph : "
         << countTriangle(graph, false);
    cout << "\n\nThe Number of triangles in directed graph : "
         << countTriangle(digraph, true);

    return 0;
}
```

Output:

```
The Number of triangles in undirected graph : 2

The Number of triangles in directed graph : 2
```

Comparison of this approach with previous approach:

Advantages:

- No need to calculate Trace.
- Matrix- multiplication is not required.
- Auxiliary matrices are not required hence optimized in space.
- Works for directed graphs.

Disadvantages:

- The time complexity is O(n^3) and can’t be reduced any further.