**Vaishnav Sonawane.**

**SY Common Department**

Library for Graph Data Structure

**Abstract—**

*This paper explains implementation for the graph data structure along with implementation of common graph algorithms in C++. An abstract base class Graph is used to model a graph. The derived classes DirectedGraph and UndirectedGraph inherit from class Graph.*

*The main objective of the project is to reduce the efforts to create Graph Data structure and algorithms associated with it and provide library for Graphs so that users can directly use Graph by importing the header files.*

**Keywords—** *Graph Data Structure, Algorithms, C++, Directed Graph, Undirected Graph.*

1. **INTRODUCTION**

Algorithms and data structures as an essential part of knowledge in a framework of computer science have their stable position in computer science curricula, since every computer scientist and every professional programmer should have the basic knowledge from the area. Our scope here is the higher education in the field of computer science.

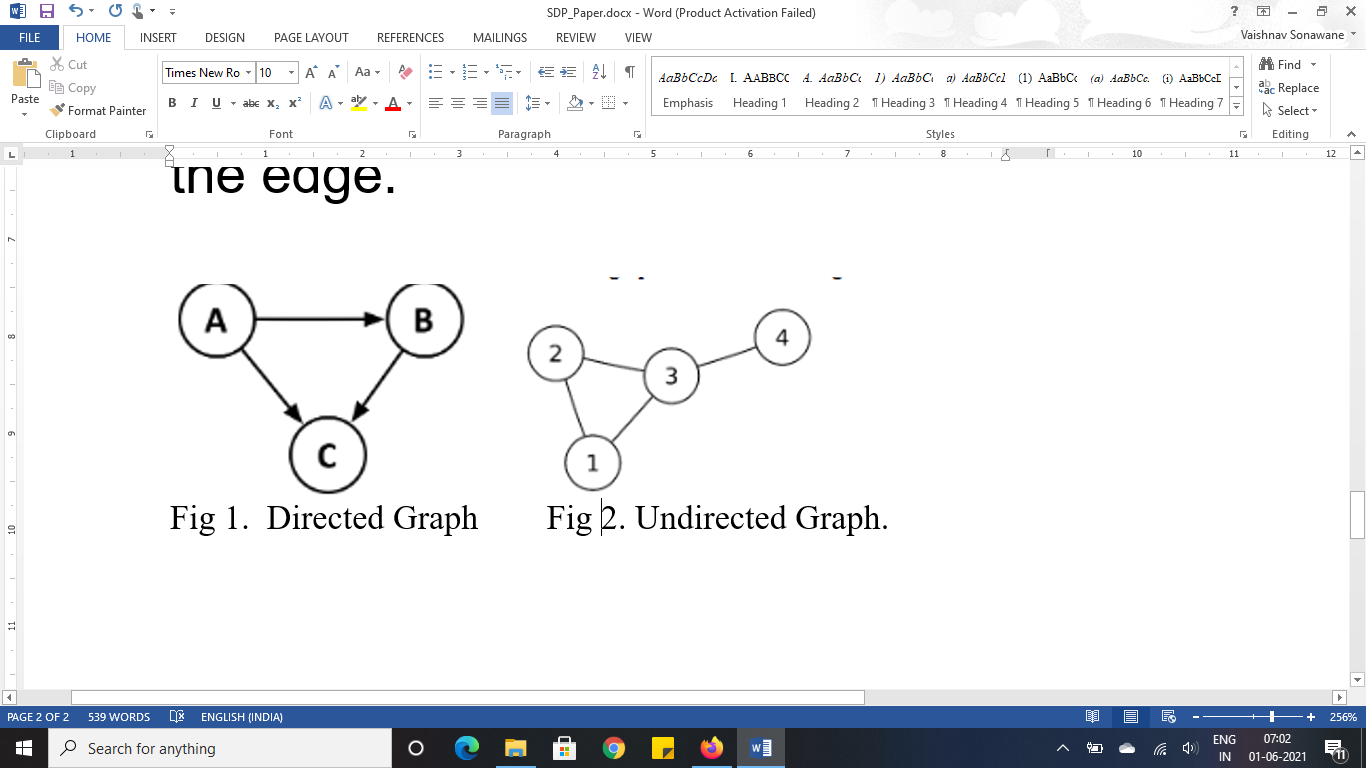
Within the paper, we discuss implementation for the Graph data structure along with implementation of common graph algorithms. An abstract base class Graph is used to model a graph. The derived classes DirectedGraph and UndirectedGraph inherit from class Graph.

Graph is a data structure that consists of finite set of vertices, together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. These pairs are known as edges/lines/arcs for undirected graphs and directed edge / directed arc / directed line for directed graphs. An edge can be associated with some edge value such as a numeric attribute. These attribute will be based on cost or length or capacity. We can represent the vertices externally also with the help of integer indices or references. Graphs are very important in the field of computer science. They are used to model real world systems such as Internet where each node represents a router and each edge represents a connection between the routers. The wireframe drawings in computer graphics are another example of graphs.

**Directed Graph**: A directed graph is a graph where all the edges are directed from one vertex to another. The order of vertices in the pairs in the edge set matters in this type of graph. Thus, A is adjacent to B only if the edge set consists of a pair (A, B). In directed graph edges are drawn as arrows indicating the direction [4]. A directed graph is sometimes called a digraph or a directed network. Directed graph can be cyclic or acyclic. Cycle is a path along the directed edges from a vertex to itself. Example of directed graph is shown in the Fig 1.

**Undirected Graph**: Undirected graph is a graph where all the edges are bidirectional. The order of vertices in the pairs in the edge set doesn’t matter in this type of graph. In undirected graph edges are drawn as straight lines [5]. Example of undirected graph is shown in the Fig. 2

**Weighted Graph**: A weighted graph is a graph where each edge has an associated numerical value, called weight. Weighted graphs may be either directed or undirected. The weight of the edge is often referred to as the “cost” of the edge.



In this project implementation of Weighted Directed graph and Weighted Undirected graph is done. An abstract base class Graph is used to model a graph. The derived classes DirectedGraph and UndirectedGraph inherit from class Graph.

1. **LITERATURE REVIEW**

Software implementations for graph structures can represent a graph as an adjacency list or a sparse adjacency matrix. There are no standards regarding the implementation details of adjacency list or adjacency matrix structures. Notable parties propose different implementations for adjacency lists [1]. Guido van Rossum suggests using a hash table with arrays, where the arrays indicate edges to vertices. Cormen et al. [2] suggest using an array of singly linked lists. Goodrich and Tamassia propose dedicated objects for both vertices and edges, allowing for easily adding information to both vertices and edges. Each of the implementation suggestions has advantages and disadvantages. One can easily come up with more variations, but their trade-offs in the context of graph applications are not clear.

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions. It is a library of container classes, algorithms, and iterators [5]. It implements the Data structures like Stacks, Queues, Vectors, Arrays, Lists, Maps etc. STL also provides methods, algorithms and iterators to perform operation of this common Data structures.

C++ doesn’t provide any standard library for the implementation of complex Data structures like Graph and Tress.

1. **FUNCTIONALITIES**

* Graph Data structure using adjacency list method.
* Directed and Undirected Graph implementation.
* Adding edge of specific weight in graph.
* Removing edge in graph
* Checking Cycle in graph.
* Printing the graph.
* Breadth first search traversal of graph.
* Depth first traversal of graph.
* Get the strongly connected components of a Directed Graph.
* Minimum Spanning tree
  + Prim’s Algorithm.
* Finding shortest path in Graph.
  + Dijkstra’s Algorithm.

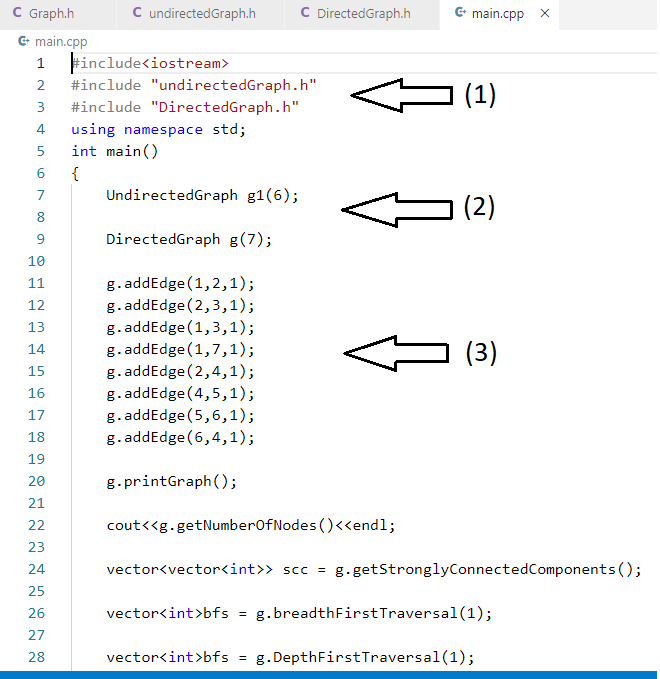
1. **METHODS and ALGORITHMS**
2. **Adjacency List:**

An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array[ ]. An entry array[i] represents the list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs.

1. **Checking Cycle in graph:**
2. Create a recursive function that initializes the current index or vertex, visited, and recursion stack.
3. Mark the current node as visited and also mark the index in recursion stack.
4. Find all the vertices which are not visited and are adjacent to the current node. Recursively call the function for those vertices, if the recursive function returns true, return true.
5. If the adjacent vertices are already marked in the recursion stack then return true.
6. Create a wrapper class, that calls the recursive function for all the vertices and if any function returns true return true. Else if for all vertices the function returns false return false.
7. **Breadth First Search:**
8. Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
9. If no adjacent vertex is found, remove the first vertex from the queue.
10. Repeat 1 and 2 until the queue is empty.
11. **Depth First Search:**
12. Create a recursive function that takes the index of node and a visited array.
13. Mark the current node as visited and print the node.
14. Traverse all the adjacent and unmarked nodes and call the recursive function with index of adjacent node.
15. **Finding Strongly Connected Components of Graph (Kosaraju’s algorithm):**
16. Create an empty stack ‘S’ and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack. In the above graph, if we start DFS from vertex 0, we get vertices in stack as 1, 2, 4, 3, 0.
17. Reverse directions of all arcs to obtain the transpose graph.
18. One by one pop a vertex from S while S is not empty. Let the popped vertex be ‘v’. Take v as source and do DFS. The DFS starting from v prints strongly connected component of v. In the above example, we process vertices in order 0, 3, 4, 2, 1 (One by one popped from stack).
19. **Dijkstra’s Algorithm:**
20. skeeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
21. Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
22. While *sptSet* doesn’t include all vertices   
     **a)** Pick a vertex u which is not there in *sptSet* and has minimum distance value.

**b)** Include u to *sptSet*.   
 **c)** Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

1. **Prim’s Algorithm:**
2. Create a set *mstSet* that keeps track of vertices already included in MST.
3. Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
4. While mstSet doesn’t include all vertices   
    **a)** Pick a vertex *u* which is not there in *mstSet* and has minimum key value.   
    **b)** Include *u* to mstSet.   
    **c)** Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u-v* is less than the previous key value of *v*, update the key value as weight of *u-v*
5. **APPLICATION OUTPUT:**



1. User importing Header files.
2. User can directly create graph by creating objects of respective classes.
3. User can add edge anytime while coding
4. User can use all the methods implemented for the graph.
5. **CONCLUSION**

In this paper we explained the implementation of a project named ‘Library for Graph data Structure’. Project implements the Graph data structures and most frequently used algorithms associated with Graph Data structure. User is able to use implemented Graph and all methods by importing necessary header files. By doing this the user is able to save the time to implement Graph and algorithms every time in his/her code.

1. **FUTURE SCOPE**

Printing method of graph can be included using some advance animation libraries in order to give the user animated view of graphs.

Applications of the algorithms can be included so that user can get proper understanding of where the algorithms can be used.

Other types of Graphs also can be added including Directed acyclic graphs, Multigraphs, Sparse and dense graphs.

1. **ACKNOWLEDGEMENT**

I am overwhelmingly thankful to all the teachers and mentors who helped to create this project. Not only the guidance, but the overall learning experience from the professors helped me learn and understand a lot of things. My guide helped me through the thick and thins of project, her opinions and guidance was the driving force to me. Her suggestions and useful technical insights helped to solve my problems whenever I got stuck.

1. **REFERENCES**

**[**1] “Adjacency list” Wikipedia. Wikimedia Foundation, accessed 10   
december 2014.

[2] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E.   
Leiserson. Introduction to Algorithms(2nd ed.). McGraw-Hill Higher   
Education, (2002).

[3] K. Mehlhorn, P. Sanders, Algorithms and Data Structures (Springer-Verlag, Berlin Heidelberg, 2008)

[4] J. Genči, Possibilities to Solve Some of the Slovak Higher Education Problems Using Information Technologies, In proceedings of: 10th IEEE International Conference on Emerging eLearning Technologies and Applications, ICETA 2012, Stará Lesná, The High Tatras, Slovakia, November 8-9, 2012

[5] [Depth-first search and linear graph algorithms](https://scholar.google.com/scholar?oi=bibs&cluster=15533190727229683002&btnI=1&hl=en)

R Tarjan - SIAM journal on computing, 1972

[6] [Data structures and network algorithms](https://scholar.google.com/scholar?oi=bibs&cluster=15312592582565753124&btnI=1&hl=en)

RE Tarjan - 1983

[7] Falley. P “Categories of Data Structures”, Journal of   
Computing Sciences in Colleges - Papers of the Fourteenth Annual   
CCSC Midwestern Conference and Papers of the Sixteenth Annual   
CCSC Rocky Mountain Conference. Volume 23 Issue 1, October   
2007. PP. 147-153, 2007-10-01

[8] <https://www.geeksforgeeks.org/graph-and-its-representations/>

[9] <https://www.freecodecamp.org/news/a-gentle-introduction-to-data-structures-how-graphs-work-a223d9ef8837/>

[10]<https://www.geeksforgeeks.org/the-c-standard-template-library-stl/>