



Generic Programming Project

UE18CS331

**Bachelor of Technology
in
Computer Science & Engineering**

Generic Graph in C++

Project ID - 43

**Arnav Agarwal
Heeth M Thakkar**

**PES1201801863
PES1201802022**

January - May 2021

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY**

(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

ABSTRACT

A Graph is a non-linear data structure consisting of a finite set of vertices(or vertex) and a set of edges which connect a pair of vertices.

Graphs are used to represent networks and solve many real-life problems. These networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like LinkedIn, Facebook.

In our project we have designed a generic container to represent a weighted and non-directed graph with all basic functionality to add, delete and modify edges. We have also provided breath first search and depth first search iterators to traverse the graph.

The graph container is used to mimic an autonomous system which basically consists of a network of routers that follow the same routing algorithm. As an algorithm customised for the graph container, we have implemented Dijkstra's algorithm which finds the shortest paths from a vertex to all other vertices in the given graph so as to develop a routing table for each router in the network.

Graph

Graph is container that stores vertices and edges connecting them. The edges are non-directed and can be weighted. The graph is implemented as an adjacency list manifested as an `unordered_map` for efficiency. The key of the `unordered_map` indicates a vertex in the graph and the corresponding value is implemented as a set of pairs with the first element in the pair indicating the vertex that the key vertex connects to and the second element having the weight of the edge between them. To allow only unique edges between a pair of vertices, the set is implemented with a customized binary predicate.

Container Properties:

Unordered

The elements in the container do not follow a strict order at all times.

Unique vertex

No two vertex in the container can have equivalent identifiers.

Non-Directed

Links connecting vertex are bidirectional.

Weighted

Links connecting vertex can carry weights.

```
template <
    typename vertex = int,           //Graph::vertex_type
    typename edge = double          //Graph::edge_type
> class Graph;
```

Member Types

member type	definition	notes
vertex_type	first template parameter (vertex)	defaults to int
edge_type	second template parameter (edge)	defaults to double
list_type	type of element in adjacency list <code>std::pair<const vertex, std::set<std::pair<vertex, edge>, uniquePair<vertex, edge>>></code>	
iterator	a forward iterator to list_type	
breadth_first_search_iterator	a forward iterator to list_type following Breath First Search traversal	
depth_first_search_iterator	a forward iterator to list_type following Depth First Search traversal	
size_type		same as size_t

Member functions

(constructor)	default
(destructor)	default
operator=	default

Iterators

begin	Return iterator to beginning of adjacency list
end	Return iterator to end of adjacency list
bfs(graph<vertex, edge>, iterator)	Constructs a breadth_first_search_iterator for the given vertex
bfsend	Checks if breadth_first_search_iterator has completed the traversal
dfs(graph<vertex, edge>, iterator)	Constructs a depth_first_search_iterator for the given vertex
dfsend	Checks if depth_first_search_iterator has completed the traversal

Capacity

empty	Test whether the container is empty
size	Return container size

Modifiers

add_edge	Adds an edge between two vertices
delete_edge	Deletes an edge between two vertices
modify_edge	Modifies the weight of the edge between two vertices

Operations

find	Returns iterator to given vertex
dijkstra	Returns shortest path cost and parent vertex for each vertex from source vertex

Time Complexity for Graph Operations

Operation	Average Case
add_edge	$O(\log n)$
delete_edge	$O(n)$
modify_edge	$O(n)$
find	$O(n)$

graph::dijkstra

```
template<typename vertex, typename edge>
std::map<vertex, std::pair<vertex, edge>> Graph<vertex, edge>::dijkstra(const vertex& src);
```

Given a graph and a source vertex in graph, Dijkstra is used to find shortest paths from source to all vertices in the given graph. There are basically two key operations in dijkstra's:

- ExtractMin : from all those vertices whose shortest distance is not yet found, we need to get vertex with minimum distance.
- DecreaseKey : After extracting vertex we need to update distance of its adjacent vertices, and if new distance is smaller, then update that in data structure.

Above operations can be easily implemented by a set as a set keeps all its keys in sorted order so the minimum distant vertex will always be at beginning, we can extract it from there, which is the ExtractMin operation and update other adjacent vertex accordingly if any vertex's distance become smaller then delete its previous entry and insert new updated entry which is DecreaseKey operation.

Parameters

src
vertex_type object that acts as the source vertex from which shortest path to all other connected vertices is computed.

Return value

Returns a map with key as vertices and value as a pair of (shortest path cost, parent vertex) from the source vertex.

Example

```
Graph<> G;
G.add_edge(0, 2);
G.add_edge(0, 4);
G.add_edge(1, 5, 5);
G.add_edge(1, 6, 10);
G.add_edge(2, 5, 25);
G.add_edge(2, 6, 25);
G.add_edge(3, 5, 25);
G.add_edge(5, 6, 25);
G.add_edge(5, 7, 25);

std::map<int, pair<int, int>> m = G.dijkstra(5);
int s;
std::stack<int> st;
for(pair<int, pair<int, int>> x: m)
{
    std::cout<<x.first<<"\t\t"<<x.second.second<<"\t\t";
    s = x.first;
    while(s!=5)
    {
        st.push(s);
        s = m[s].first;
    }
    st.push(5);
    while(!st.empty())
    {
        std::cout<<"->"<<st.top();
        st.pop();
    }
    std::cout<<endl;
}
```

Output

0	25	->5->2->0
1	5	->5->1
2	25	->5->2
3	25	->5->3
4	25	->5->2->0->4
5	0	->5
6	15	->5->1->6
7	25	->5->7

Complexity

$O(E \log(V))$ – Since insertion and deletion of set is logarithmic in time

Where E – Number of edges

V – Number of vertices

References:

<https://www.cplusplus.com/reference/>

<https://en.cppreference.com/w/>

<https://users.cs.northwestern.edu/~riesbeck/programming/c++/stl-iterator-define.html>