Graph Template Library

November 8, 2022

Atharva Suhas Mulay (2021CSB1076) ,
Karanraj Mehta (2021CSB1100) ,
Sumit Patil (2021CSB1135)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Sravanti Chede

Summary:

- Graph template library to study graphs in C++
- Generic graph library
- Defines several container template classes
- Data structures for graphs, digraphs and weighted graphs
- Many standard graph algorithms
- Nodes can be arbitrary objects

Introduction

An Abstract Data Type in data structures is a kind of data type whose behaviour is defined with the help of some attributes and some functions. Generally, we write these attributes and functions inside a class or a structure so that we can use an object of the class to use that particular abstract data type. Graph is an abstract data structure. They are mathematical data structures that are useful for solving many kinds of problems in computer science

A graph is a relation on a finite set of vertices. Two vertices are related if there is an edge between them. If V and E denote the set of vertices and edges, the graph is represented as G(V, E).
There are several types of graphs in data structures. These include undirected, directed, weighted, multi, complete, simple, bipartite, and cyclic graphs. In our library, we have focused mainly on weighted and unweighted directed and undirected graphs.

- Unweighted Graph: No value associated with the edges
- Weighted Graph: Whose edges have values. All the values associated with the edges are called weights.
- Undirected Graph: All edges are bidirectional
- Directed Graph: All edges are directed from one node to the other.

Graphs can be represented using either an Adjacency list or an Adjacency matrix. In our library, we've used adjacency list representation of graphs because it is more space efficient. In the adjacency list representation, each node is mapped to a list of all its adjacencies or neighbours.

All classes, functions and variable names are lower_case_underscore. Our library provides the following graph types:

- `graph`
  This class implements an undirected graph. It assumes multiple edges between two nodes are not added. It does allow self-loop edges between a node and itself.
- `digraph`
  Directed graphs, that is, graphs with directed edges. Provides operations common to directed graphs.
- `wgraph`
  Weighted undirected graphs. Allows only integer weights to the edges.

  - `wdigraph`
    Weighted directed graphs. Allows only integer weights to the edges.

All graph classes allow any object as node such as `char, int, string` and more. Only int weights are allowed in wgraph and wdigraph classes.

The graph internal data structures are based on an adjacency list representation and implemented using map data structures. In unweighted graphs, each node in the adjacency list is mapped to a list (vector) of its neighbours. In weighted graphs, each node is mapped to a vector of pairs in which the first attribute is the neighbour of that node and the second attribute is the weight of the edge connecting it to that node.

Example

Finds number of connected components in an undirected graph

```
#include <iostream>
#include "graph.h"

 int main() {
   graph<char> g;
   g.add_edge('a', 'b');
   g.add_edge('c', 'd');
   g.add_edge('d', 'e');
   g.add_edge('f', 'g');

   std::cout << "Number of connected components in g are " << g.
  number_of_connected_components() << "\n";
   return 0;
 }
```

Output

```
Number of connected components in g are 3
```

graph Class

A template container class for unweighted undirected graphs. Self-loops are allowed, but multiple edges are not. Nodes can be arbitrary objects.

- Members

graph::number_of_edges

Stores number edges in the graph

Syntax

```
int number_of_edges;
```

Remarks

The member variable is 0 for an empty graph

---

graph::adj

Stores the adjacency list of the graph. It is implemented using map data structure in which each node is mapped to a vector to its neighbours.

Syntax

```
std::map<T, std::vector<T>> adj;
```

T is a template parameter

`graph::add_node`

Adds the specified node to that graph

Syntax

```
void add_node(T u);
```

Parameters

u
Node to be added

Time Complexity

O(log n) worst case time complexity where n is the number of nodes since the adjacency list uses map data structure which takes O(log n) time for insertion.

`graph::add_edge`

Adds an undirected edge between the two specified nodes.

Syntax

```
void add_edge(T u, T v)
```

Parameters

u
First node
v
Second node

Remarks

It is assumed that multiple edges between two nodes are not added.

Time Complexity

O(log n) where n is the number of nodes.

`graph::number_of_nodes`

Returns the number of nodes in the graph

Syntax

```
int number_of_nodes();
```

Return Value

Number of nodes in the graph

Time Complexity

O(1)

---

graph::remove_node

Removes the specified node from the graph

Syntax

```
void remove_node(T u);
```

Parameters

u Node to be removed

Time Complexity

O(n^2 log n) worst case time complexity assuming complete graph where n is the number of vertices

---

graph::remove_edge

Removes the specified edge from the graph

Syntax

```
void remove_edge(T u, T v);
```

Parameters

u, v nodes between which the edge is to be removed

Time Complexity

O(n)

---

graph::degree

Returns the degree of the specified node

Syntax

```
int degree(T u);
```

Parameters

u
Node whose degree is to be found

Time Complexity

O(log n)

---

graph::clear

Empties the graph

Syntax

```
void clear();
```

Time Complexity

O(log n)

---

graph::number_of_connected_components

Finds the number of connected components in the graph

Syntax

```
int number_of_connected_components();
```

Return Value

Returns the number of connected components in the graph

Algorithm

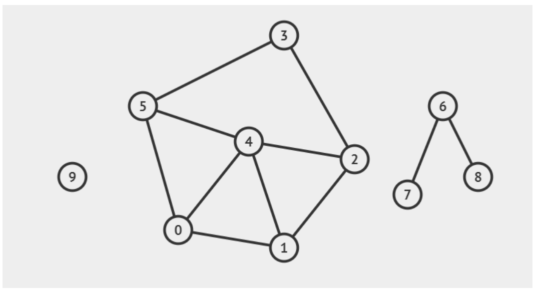
This function calculates the number of connected components by running a series of DFS.

1. Start at one node and perform DFS. All nodes in the connected component containing this node will be found.
2. Find the next unvisited node and run DFS on it, finding the second connected component.
3. If some unvisited nodes are still left, continue from step 2.
4. Finally, return the number of connected components.

Time Complexity

Since this algorithm will not run on the same vertex twice, its time complexity will be same as that of dfs, that is O(V log V + E)

Example



```cpp
 #include <iostream>
#include "graph.h"

int main() {
    graph<int> g;
    g.add_edge(0, 1);
    g.add_edge(0, 4);
    g.add_edge(0, 5);
    g.add_edge(1, 2);
    g.add_edge(1, 4);
    g.add_edge(2, 3);
    g.add_edge(2, 4);
    g.add_edge(3, 5);
    g.add_edge(4, 5);
    g.add_edge(6, 7);
    g.add_edge(6, 8);

    g.add_node(9);


    std::cout<<"The Number of Nodes is : "<<g.number_of_nodes()<<std::endl;
    std::cout<<"The Number of Edges is : "<<g.number_of_edges<<std::endl;
    std::cout<<"The Number of Connected Components is : "<<g.
   number_of_connected_components()<<std::endl;
    std::cout<<"The Degree of 4 is : "<<g.degree(4)<<std::endl;
    std::map<int,std::vector<int>>M= g.adj;
    std::cout<<"\n";
    std::cout<<"The Adjacency List is : "<<std::endl;

    for(auto it:g.adj){
        std::cout<<it.first<<" : ";
        for(auto i:it.second){
            std::cout<<i<<" ";
        }
        std::cout<<std::endl;
    }



    return 0;
}
```

Output

```
The Number of Nodes is : 10
The Number of Edges is : 11
The Number of Connected Components is : 3
The Degree of 4 is : 4

The Adjacency List is :
0 : 1 4 5
1 : 0 2 4
2 : 1 3 4
3 : 2 5
4 : 0 1 2 5
5 : 0 3 4
6 : 7 8
7 : 6
8 : 6
9 :
```

---

graph::bfs

It performs Breadth-First Search traversal on a given graph with given source vertex s.

Syntax

```
std::vector<T> bfs(T s);
```

Parameters

s
Source vertex

Return Value

Returns a vector containing the bfs traversal starting from source vertex s.

Algorithm

The BFS algorithm is defined as follows:

1. Consider an undirected graph with vertices numbered from 1 to n. Initialise queue q as a new queue containing only vertex s, and mark the vertex s as visited.
2. Extract a vertex v from the head of the queue q.
3. Store vertex v in bfs traversal.
4. Iterate in arbitrary order through all such vertices u that u is a neighbour of v and is not marked yet as visited. Mark the vertex u as visited and insert it into the tail of the queue q.
5. If the queue is not empty, continue from step 2.
6. Otherwise, return the bfs traversal of the graph.

Time Complexity

For regular bfs, the time complexity is O(V + E), where V is the number of vertices and E is the number of edges. However, because of the map implementation of the adjacency list, it becomes O(V log V + E)

`graph::dfs`

It performs Depth First Search traversal on a given graph with a given source vertex u.

Syntax

```
std::vector<T> dfs(T u);
```

Parameters

u
Source vertex

Return Value

Returns a vector containing the dfs traversal starting from source vertex u.

Algorithm

The recursive implementation of DFS is as follows:

1. Start the search at one vertex.
2. After visiting that vertex, we perform a DFS for each adjacent vertex we haven't visited before.
3. This way, we visit all vertices that are reachable from the starting vertex.

The iterative implementation of DFS is as follows:

1. Initialise stack s as a new stack containing only vertex u, and mark the vertex u as visited.
2. Extract v from the top of the stack s.
3. Store vertex v in the dfs traversal.

4. Iterate in arbitrary order through all such vertices u that u is a neighbour of v and is not marked yet as visited. Mark the vertex u as visited and push it into the stack s.
5. If the stack is not empty, continue from step 2.
6. Otherwise, return the dfs traversal of the graph.

Time Complexity

O(V log V + E)

---

graph::cyclic

Checks whether the graph contains a cycle or not

Syntax

```
bool cyclic();
```

Return Value

Returns true if the graph is cyclic, else returns false

Algorithm

It uses DFS to check for cyclicity.

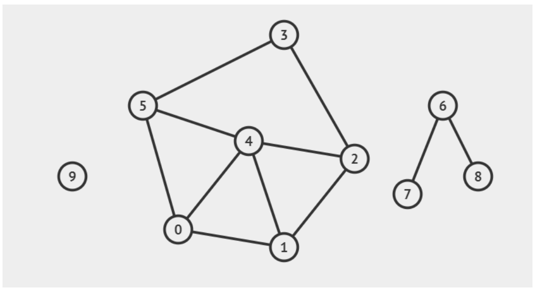
1. Initially all vertices are unvisited.
2. Construct a data structure to store parents (In our implementation, we use a map).
3. For each unvisited node, run DFS and update the parent map.
4. If DFS discovers a visited node, then we have found a cycle and return true.
5. Otherwise return false.

The parent map can be used the print the cycle found.

Time Complexity

Since this algorithm will not run on the same vertex twice, its time complexity will be same as that of dfs, that is O(V log V + E)

Example



```
 #include <iostream>
#include "graph.h"

int main()
{
    graph<int> g;
    g.add_edge(0, 1);
    g.add_edge(0, 4);
    g.add_edge(0, 5);
    g.add_edge(1, 2);
    g.add_edge(1, 4);
    g.add_edge(2, 3);
    g.add_edge(2, 4);
    g.add_edge(3, 5);
    g.add_edge(4, 5);
    g.add_edge(6, 7);
    g.add_edge(6, 8);

    g.add_node(9);

    std::vector<int> V1 = g.dfs(0);
    std::vector<int> V2 = g.bfs(0);
    std::cout << "The DFS Traversal is : ";
    for (auto it : V1)
    {
        std::cout << it << " ";
    }
    std::cout << std::endl;
    std::cout << "The BFS Traversal is : ";
    for (auto it : V2)
    {
        std::cout << it << " ";
    }
    std::cout << std::endl;

    if (g.cyclic())
    {
        std::cout<<"Graph has atleast one cycle."<<std::endl;
    }
    else{
        std::cout<<"Graph doesn't have a cycle"<<std::endl;
    }
    return 0;
}
```

Output

```
The DFS Traversal is : 0 5 3 2 4 1
The BFS Traversal is : 0 1 4 5 2 3
Graph has atleast one cycle.
```

---

graph::is_bipartite

Checks whether the graph is bipartite or not

Syntax

```
bool is_bipartite();
```

Return Value

Returns true if the graph is bipartite, else returns false

Algorithm

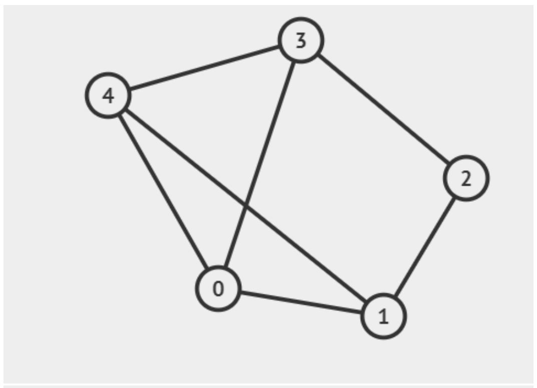
The algorithm uses the fact that a graph is bipartite if and only if it is two-colorable. Here, use BFS to traverse the graph.

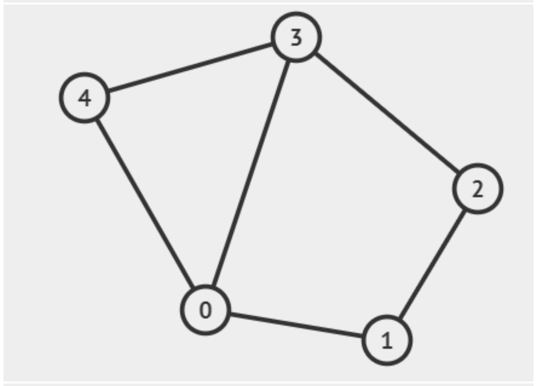
1. For each unvisited vertex assign it one colour, say blue and assign all it's neighbours the opposite colour, say red.
2. When the neighbour of a vertex has already been visited, we check whether its colour is opposite to the current node or not. If yes, continue from step 1, else return false.
3. Finally if all the vertices have been visited, we conclude that the graph is bipartite and return true.

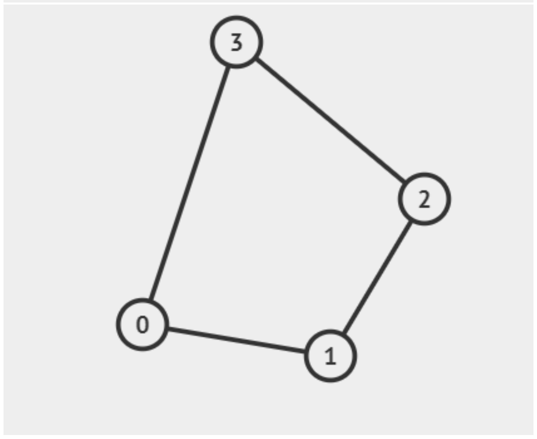
Time Complexity

Since this algorithm will not run on the same vertex twice, its time complexity will be same as that of bfs, that is O(V log V + E)

Example







```cpp
#include <iostream>
#include "graph.h"

int main()
{
    graph<int> g;
    g.add_edge(0, 1);
    g.add_edge(0, 3);
    g.add_edge(0, 4);
    g.add_edge(1, 2);
    g.add_edge(2, 3);
    g.add_edge(3, 4);
    g.add_edge(4, 1);

    std::cout << "The Adjacency List is : " << std::endl;
    for (auto it : g.adj)
    {
        std::cout << it.first << " : ";
        for (auto i : it.second)
        {
            std::cout << i << " ";
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;
    if (g.is_bipartite())
    {
        std::cout << "The Graph is Bipartite" << std::endl;
    }
    else
    {
        std::cout << "The Graph is not Bipartite" << std::endl;
    }
    g.remove_edge(4, 1);
    std::cout << std::endl;
    std::cout << "The Adjacency List after removing edge between 4 and 1 is : " <<
   std::endl;

    for (auto it : g.adj)
    {
        std::cout << it.first << " : ";
        for (auto i : it.second)
        {
            std::cout << i << " ";
        }
        std::cout << std::endl;
    }

    g.remove_node(4);
    std::cout << std::endl;
    std::cout << "The Adjacency List after removing node 4 is : " << std::endl;

    for (auto it : g.adj)
    {
        std::cout << it.first << " : ";
        for (auto i : it.second)
        {
            std::cout << i << " ";
        }
        std::cout << std::endl;
    }

    if (g.is_bipartite())
    {
        std::cout << "The Graph is Bipartite" << std::endl;
    }
    else
    {
        std::cout << "The Graph is not Bipartite" << std::endl;
    }
    return 0;
}
```

Output

```
The Adjacency List is :
0 : 1 3 4
1 : 0 2 4
2 : 1 3
3 : 0 2 4
4 : 0 3 1
```

```
The Graph is not Bipartite

The Adjacency List after removing edge between 4 and 1 is :
0 : 1 3 4
1 : 0 2
2 : 1 3
3 : 0 2 4
4 : 0 3

The Adjacency List after removing node 4 is :
0 : 1 3
1 : 0 2
2 : 1 3
3 : 0 2

The Graph is Bipartite.
```

---

digraph Class

A template container class for unweighted directed graphs. Self-loops are allowed, but multiple edges (parallel) are not. Nodes can be arbitrary objects.

- Members

For number_of_edges, adj, add_node, number_f_nodes, remove_node, clear, bfs, dfs see graph class.

---

digraph::add_edge

Adds a directed edge from the first node to the second

Syntax

```
void add_edge(T u, T v);
```

Parameters

u
First node v
Second node

Remarks

Edge is directed from u to v (first node to second). It is assumed that multiple edges between two nodes are not added. The order in which the nodes occur is important.

Time Complexity

O(log n)

---

digraph::remove_edge

Removes the specified edge from the graph

Syntax

```
void remove_edge(T u, T v);
```

Parameters

u, v nodes between which the edge is to be removed

Remarks

The order in which the nodes occur is important.

Time Complexity

O(n)

`digraph::in_degree`

Stores the in degree of all nodes in the graph

Syntax

```
std::map<T, int> in_degree;
```

Remarks

in_degree maps each node to its in degree in the graph.It takes O(log n) time.

`digraph::out_degree`

Returns the out degree of the specified node

Syntax

```
int out_degree(T u);
```

Parameters

u
Node whose out degree is to be found

Remarks

Returns the out degree of the specified node

`digraph::cyclic`

Checks whether the graph contains a cycle or not

Syntax

```
bool cyclic();
```

Return Value

Returns true if the graph is cyclic, else returns false

Algorithm

If the directed graph has a topological sort then it is a DAG. Otherwise, it contains a cycle.

`digraph::topological_sort`

Finds a topological order of the given directed graph if it exists

Syntax

```
std::vector<T> topological_sort();
```

Return Value

Returns a vector containing a topological sort of the given directed graph. As a topological sort is possible only for a DAG, if the graph contains cycles it returns an empty vector.

Algorithm

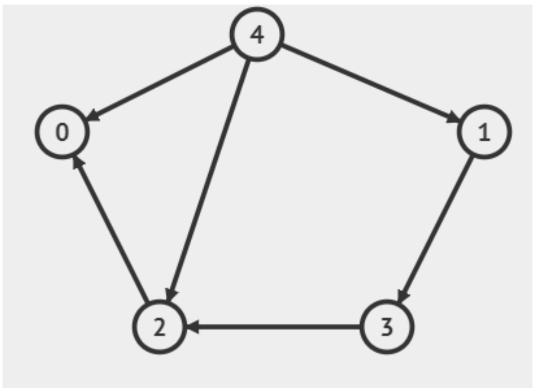
It uses Kahn's algorithm to find a topological sort of the given DAG.

1. Initialise a queue q with all the nodes whose in degree is 0.
2. Extract vertex v from the front of the queue.
3. Add v to the topological order list.
4. Decrement the in degree of all its neighbours by 1. If the in degree of some node becomes 0, insert it into the tail of the queue.
5. If the queue is not empty continue from step 2.
6. Otherwise, return the topological order list if it contains all the nodes else return an empty list.

Time Complexity

Since this algorithm will not run on the same vertex twice, its time complexity is calculated in a way similar to bfs, that is O(V log V + E).

Example



```
#include <iostream>
#include "graph.h"

int main()
{
    digraph<int> g;
    g.add_edge(1, 3);
    g.add_edge(2, 0);
    g.add_edge(3, 2);
    g.add_edge(4, 0);
    g.add_edge(4, 1);
    g.add_edge(4, 2);


    std::vector<int> V = g.topological_sort();
    std::cout << "The Topological sort is : ";
    for (auto it : V)
    {
        std::cout << it << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Output

```
The Topological sort is : 4 1 3 2 0
```

---

digraph::SCCs

Finds all strongly connected components in the given directed graph.

Syntax

```
std::vector<std::vector<T>> SCCs();
```

Return Value

Returns a list of lists or, more precisely, a vector of vectors containing all SCCs in the given directed graph.

Algorithm

It uses Kosaraju's algorithm to find SCCs of the given Directed graph

1. Create an empty stack S and do dfs traversal of the graph.
2. In dfs traversal, after calling the recursive dfs for adjacent vertices of a vertex, push the vertex to the stack. This will store all the vertices in the stack in order of their finishing times.
3. Reverse directions of all edges to obtain the transpose graph.
4. One by one, pop the top vertex from S while S is not empty. Perform the dfs traversal assuming the popped vertex as source.
5. The dfs starting from the popped vertex will give the strongly connected components of the popped vertex

Time Complexity

Since it performs dfs which takes O(V+E) and then reversing the graph also takes O(V+E), so time complexity of this algorithm is O(V+E).

---

digraph::number_of_SCCs

Finds the number of strongly connected components in the given directed graph. It calls digraph::SCCs and returns the size of the vector returned

Syntax

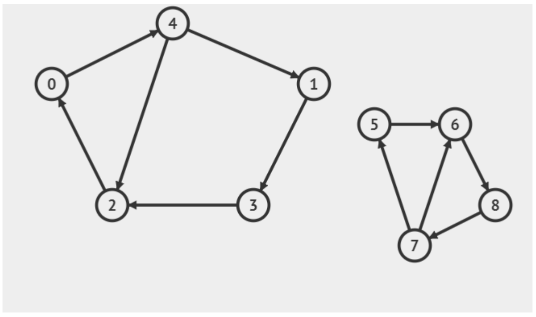
```
int number_of_SCCs();
```

Return Value

Returns the number of strongly connected components in the given directed graph

---

Example



```cpp
#include <iostream>
#include "graph.h"

int main()
{
    digraph<int> g;
    g.add_edge(0, 4);
    g.add_edge(1, 3);
    g.add_edge(2, 0);
    g.add_edge(3, 2);
    g.add_edge(4, 1);
    g.add_edge(4, 2);
    g.add_edge(5, 6);
    g.add_edge(6, 8);
    g.add_edge(7, 5);
    g.add_edge(7, 6);
    g.add_edge(8, 7);


    std::cout<<"The no. of strongly connected component is "<<g.number_of_SCCs()<<std
    ::endl;
    std::cout<<"The Strongly Connected components are : "<<std::endl;
    std::vector<std::vector<int>> V1 = g.SCCs();
    for (auto i : V1)
    {
        for (auto j : i)
        {
            std::cout<<j<<" ";
        }
        std::cout<<std::endl;
    }

    return 0;
}
```

Output

```
The no. of strongly connected component is 2
The Strongly Connected components are :
5 7 8 6
0 2 3 1 4
```

---

wgraph Class

A template container class for weighted undirected graphs. Allows only integer weights to the edges.

- Members

For number_of_nodes, number_of_edges, add_node, remove_node, remove_edge, degree, and clear see graph class.

---

wgraph::adj

Stores the adjacency list of the graph. It is implemented using map data structure in which each node is mapped to a vector of pairs in which the first attribute is the neighbour of that node and the second attribute is the weight of the edge connecting it to that node.

Syntax

```
std::map<T, std::vector<std::pair<T, int>>> adj;
```

---

wgraph::add_edge

Adds an undirected edge between the two specified nodes with specified weight. Weight must be an integer. If weight is not mentioned, the default weight is used, which is 1.

Syntax

```
void add_edge(T v, T w, int k);
```

Parameters

u
First node v
Second node k
Weight

Time Complexity

O(log n) where n is the number of nodes

---

wgraph::bellman_ford

Finds the shortest distance from every vertex to all other vertices, and also detects negative weight cycle

Syntax

```
std::vector<std::vector<int> bellman_ford();
```

Return Value

Returns a list of lists or, more precisely, a vector of vectors containing shortest distances from every vertex to every other vertex for the given directed graph.

Algorithms

Our implementation of bellman ford algorithm is described as follows:

1. Perform the following operations for every vertex of the graph taking them as source.
2. Initialise distance from the source to all vertices as infinite and distance to the source itself as zero.
3. Create a map dist which stores all vertices as keys and sets their distances as infinite except dist[src], where src is the source vertex.
4. Do this operation |V| -1 times, it will calculate the the shortest distances.
   - If dist[v] > dist[u] + weight of edge uv, then update dist[v] to dist[v] = dist[u] + weight of edge uv.
5. Again traverse every edge and do following for each u-v
   - If dist[v] > dist[u] + weight of edge uv, then graph contains Negative weight cycle

---

`wgraph::dijkstra`

Find the shortest distance between a starting vertex s and all other vertices.

Syntax

```
std::map<T, int> dijkstra(T s);
```

Parameters

s
Starting vertex

Return Value

Returns a map that maps all vertices to their shortest distance from the starting vertex.

Remarks

The algorithm will work if and only if the weights of all edges are non-negative.

Algorithms

Our implementation of dijkstra's algorithm is described as follows:

1. Create a map of predecessors and a map of distances. Initialize the map of distances with infinity (INT_MAX). Set distance of the starting vertex s from itself as 0.
2. Create a min priority queue q of pairs in which the first attribute is distance from s and the second attribute is a graph node. Initialize it with 0, s.
3. Extract the v and d_v (distance of v from s) from the top of the min priority queue q. If the queue is empty, halt and continue from step 6.
4. If d_v and d[v] (distance map) are equal, halt and continue from step 3, otherwise continue.
5. For all neighbours e of v, if d[v] + weight of edge v, e / (v, e) (for wdigraph) is less than distance of e from s than update the distance of e from s. Push the pair of distance of e from s and e in the priority queue q. Update predecessor of e to v.
6. Finally, return d.

Time Complexity

Time complexity depends on finding vertex with smallest distance from s and the time taken for updating distance map. The time complexity is O(E log V).

`wgraph::path`

Finds the shortest path from a starting node to an ending node

Syntax

```
std::vector<T> path(T from, T to);
```

Parameters

from
Starting node
to
Ending node

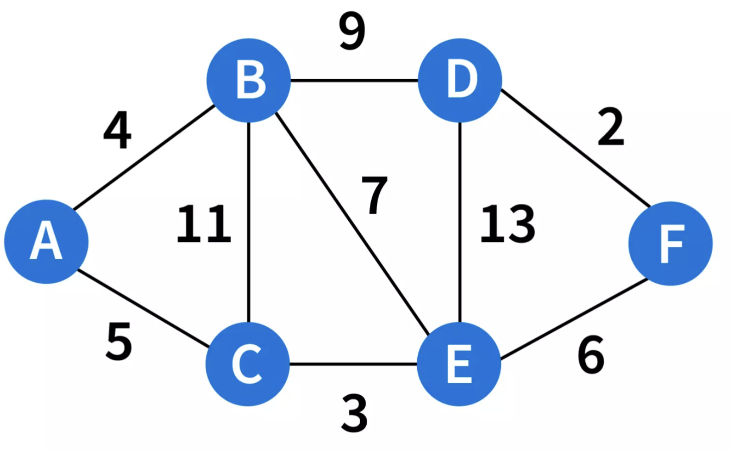
Return Value

Return a vector containing the shortest path from the starting node to the ending node

Remarks

Its internal implementation uses Dijkstra's algorithm. It calls the function dijksta.

Example



```
#include <iostream>
#include "graph.h"

int main()
{
    wgraph<char> g;
    g.add_edge('A', 'B', 4);
    g.add_edge('B', 'D', 9);
    g.add_edge('D', 'F', 2);
    g.add_edge('F', 'E', 6);
    g.add_edge('E', 'C', 3);
    g.add_edge('C', 'A', 5);
    g.add_edge('B', 'C', 11);
    g.add_edge('E', 'D', 13);
    g.add_edge('E', 'B', 7);

    std::cout << "By Dijkshtra's Algorithm.... " << std::endl;
    std::cout << std::endl;

    std::map<char, int> V = g.dijkstra('A');
    std::cout << "Shortest Distance From A to all other nodes : " << std::endl;
    for (auto it : V)
    {
        std::cout << it.first << " : " << it.second << std::endl;
    }
    std::cout<<std::endl;
    std::vector<char> V1 = g.path('A', 'F');
    std::cout << "Shortest Path From A to F is : " ;
    for (auto it : V1)
    {
        std::cout<<it<<" ";
    }
    std::cout<<std::endl;

    return 0;
}
```

Output

```
By Dijkshtra's Algorithm....

Shortest Distance From A to all other nodes :
A : 0
B : 4
C : 5
D : 13
E : 8
F : 14

Shortest Path From A to F is : A C E F
```

---

wdigraph Class

A template container class for weighted directed graphs. Allows only integer weights to the edges. Allows only integer weights to the edges.

- Members

For number_of_nodes, number_of_edges, in_degree, out_degree, add_edge, add_node, remove_edge see digraph class. For bellman_ford, dijkstra, path see wgraph class.

---

wdigraph::add_edge

Adds a directed edge from the first node to the second with a specified weight. Weight must be an integer. If weight is not mentioned, the default weight is used, which is 1.

Syntax

```
void add_edge(T v, T w, int k);
```

Parameters

u
First node
v
Second node
k
Weight

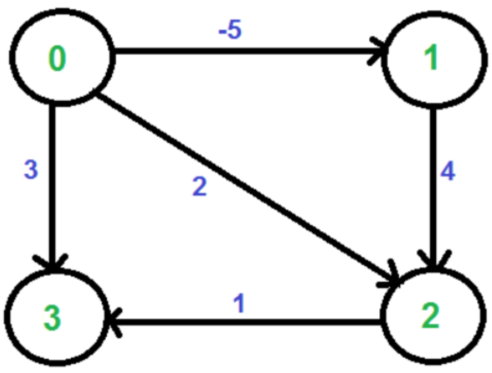
Remarks

Edge is directed from u to v (first node to second). It is assumed that multiple (parallel) edges are not added. The order in which the nodes occur is important.

Time Complexity

O(log n) where n is the number of nodes

Example



```
#include <iostream>
#include "graph.h"

int main()
{
    wdigraph<int> g;
    g.add_edge(0, 1, -5);
    g.add_edge(0, 3, 3);
    g.add_edge(0, 2, 2);
    g.add_edge(2, 3, 1);
    g.add_edge(1, 2, 4);

    std::cout<<"By Bellman Ford Algorithm.... "<<std::endl;
    std::cout << std::endl;

    std::vector<std::vector<int>> V = g.bellman_ford();
    std::cout << "Shortest Distance From all nodes to all other nodes : " << std::
   endl;

    std::cout << std::endl;
    std::cout << "  ";
    for (auto i : g.adj)
    {
        std::cout << i.first << " ";
    }
    std::cout << std::endl;
    auto it = g.adj.begin();
    for (auto i : V)
    {
        std::cout << it->first << " ";
        for (auto j : i)
        {
            if (j == INT_MAX)
            {
                std::cout << "N ";
            }
            else
            {
                std::cout << j << " ";
            }
        }
        std::cout << std::endl;
        it++;
    }
    std::cout << std::endl;

    return 0;
}
```

Output

```
By Bellman Ford Algorithm....

Shortest Distance From all nodes to all other nodes :

  0 1 2 3
0 0 -5 -1 0
1 N 0 4 5
2 N N 0 1
3 N N N 0
```

Applications

1. Used for solving flow problems, which encompass real life scenarios like the scheduling of airlines.
2. Directions in a google map works on the concept of finding shortest path using Dijkstra's algorithm
3. Graph Coloring concept can be used to solve the most popular puzzle game, Sudoku and many other board games by converting the board/puzzle into a graph.
4. Search engines such as Google let us navigate through the World Wide Web without any problem using graph theory by first creating a web graph

Conclusions

In this project, we have created a library which provides data structures for graphs, digraphs, weighted graphs and related algorithms. It will be very useful for studying and analyzing sparse graphs. It can be used to simulate graphs problems such as Sudoku, Snakes & Ladders and many other real life problems.

Bibliography and citations

Acknowledgements

We wish to thank our instructor, Dr Anil Shukla and our Teaching Assistant Sravanthi Chede for their guidance and valuable inputs provided during the project.

References

[1] RodionGork Adamant-pwn, Jakobkogler. Algorithms for competitive programming. Technical report, CP Algorithms, 2014.

[2] Gregory Gutin Jørgen Bang-Jensen. Digraphs theory, algorithms and applications. 2001.

[3] Visual Go Software. visualing data structures through animation. Technical report, 2010.

[4] Ronald Rivest Clifford Stein Thomas H. Cormen, Charles E. Leiserson. Introduction to algorithm. 1989.

[5] Wikipedia. Graph theory. Technical report, Wikipedia, 2022, September 27.