C++ Graph Library Reference Manual

April 30, 2018

1 Introduction

This is the manual for our C++ Graph library. It is a simple C++ class library for the computation and analysis of graphs like maps, social networks, web pages. This library provides convenient and efficient functionality for creating and accessing different directed graphs with no complex interfaces. In this version, we implement graph classes on directed graphs (which can also be used as the basis of undirected graphs) with no multiple edges or loops, which probably is the most common and widely used model for graph. It presents you with the promising future of creating efficient graphs with just a few lines of codes.

2 Graphs

In this section, we introduce different graphs in this library, including their hierarchy organization, important structures and classes, and some basic operations.

2.1 Graph

Our library presents graphs in a hierarchy relationship. A virtual base class **graph** is the base for all graphs mentioned below. A class **directed_graph** inherits from **graph**, which presents all directed graphs. Then there are four types of directed graphs:

- directed dense graph: **directed_dense_graph**.
- directed sparse graph: **dierected_sparse_graph**
- directed dense graph with fixed size: **fixed_directed_dense_graph**
- directed sparse graph with fixed size: **fixed_directed_sparse_graph**

By "dense" we mean that the number of edges is close to the maximal number of edges and "sparse" means that a graph has only a few edges[1]. A graph has

[1] https:en.wikipedia.orgwikiDense_graph

fixed size if and only if the maximum total number of vertices is a constant. All directed graphs have similar basic operations like inserting a new node or edge, erasing an existing edge, printing the graph information and so on, while they have different implementations. This allows absence in defining specific graph types in some basic algorithms, which we will introduce in detail in section 3.

2.2 Node

A node is a fundamental unit for a graph to be formed. It usually includes some information about some objects. Our design allows the node to be defined by the user to include specific node information. Replace the content of class **city** with your specific node information in the user_structure.h file to generate node types. The user-defined node type has to have a **to_string** function, which returns all information in this node as a string.

```
struct city {
    string name;
    city(string input): name(input) {}
    string to_string() {
        return name;
    }
};
```

Our design generate new nodes from graph using **insert_node** function rather than from the node itself, which means that one node needs to belong to one graph for future use. This function will return the unique identifier which is a handler that can be used to access this node in the future.

2.3 Edge

An edge is an ordered pair of nodes. We identify the source node as **from** node and the target node as **to** node in the code. Our design allows the edge to be defined by the user to include specific edge information. Replace the content of class **dis** to generate edge types. The user-defined edge type has to have a **to_string** to return all information in this node as a string, a **get_val** function to return the weight of this edge, and a **operator+** function to return the sum of this edge's weight and another edge's weight.

```
struct dis {
    size_t miles;
    dis(){}
    dis(size_t input): miles(input) {}
    string to_string() {
        return std::to_string(miles);
    }
    const size_t get_val() const{
        return miles;
    }
    dis operator+(dis d) const{
```

```
        return dis(miles + d.miles);
    }
};
```

An edge is generated only from a graph using **insert_edge**, to ensure that edges are always attached to a specific graph. This function will return a pair of from and to nodes, which is a handler that can be used to access this edge in the future.

2.4 Graph operations

Basic operations for **directed_dense_graph**, **directed_sparse_graph**, **fixed_directed_dense_graph** and **fixed_directed_sparse_graph** are quite similar. They all use **<V,E>** as template, while V stands for the **city** struct of nodes and E stands for the **dis** struct of edges mentioned previously. They all use **node_handle** as **size_t**, **edge_handle** as **pair<node_handle, node_handle>** and **cost_type** as **E**. The only difference for using these graph classes is that for fixed graph, either dense or sparse, we can call the constructor with or without arguments (with an int argument to set the maximum number of nodes in this graph), while for unfixed graph we can only call the constructor with no argument, because we cannot set the size as a constant.

2.4.1 node_handle insert_node (V info)

Insert new node into the graph. Input is the information of the new node, returns the node handler.

```
//create a directed dense graph ddg
directed_dense_graph<V, E> ddg = directed_dense_graph<V, E>();
//insert a node with name "New York" into fddg
//the handler of this node is v0
auto v0 = ddg.insert_node(city("New_York"));
```

2.4.2 size_t node_cnt()

Count the number of nodes in the graph. No input arguments. Returns the size of current nodes.

2.4.3 V handle_to_info (node_handle v)

Get the node information of given node handler. Inputs a node handler, returns the corresponding node of this handle.

2.4.4 E end_to_edge (node_handle h1, node_handle h2)

Find the edge information of given node handlers. Inputs are two node handlers, the handlers of the source and target nodes. Returns the possible edge between them.

2.4.5 edge_handle insert_edge (node_handle v1, node_handle v2, E info)

Insert new edge of two nods into the graph. Inputs two node handlers and the information of the edge from the first node to the second node. Returns the handler of the newly inserted edge.

2.4.6 void erase_edge (edge_handle e)

Erase existed edge. Inputs the handler of the edge which needs to be deleted.

2.4.7 void print_graph()

Print information of all nodes and edges in this graph.

2.4.8 out_edges out(node_handle v)

Get the information of all edges whose source node is the node of given node handler. Inputs the node handler, returns all the edges starts from it.

```
//do something to all the edges starting from node v in graph g
for (auto& e: g.out(v)) {
            //do something
        }
```

2.4.9 const V& operator[](node_handle v)const

Access to the node of given node handler. Inputs the node handler, returns the node information.

3 Algorithms

We provide implementations of several basic and useful graph algorithms in **agorighms.h**.

3.1 pathexists

bool bfs_pathexists(G g, typename G::node_handle s, typename G::node_handle e)
bool dfs_pathexists(G g, typename G::node_handle s, typename G::node_handle e)
Do bfs/dfs on the graph from given node handlers to know whether there is a path between them. Inputs the graph and two node handlers of source and target nodes. Return whether there is a path between them using bfs/dfs.

3.2 findpath

shared_ptr<path<G, typename G::node_handle>> bfs_findpath(G g, typename G::node_handle s, typename G::node_handle e)
shared_ptr<path<G, typename G::node_handle>> dfs_findpath(G g, typename G::node_handle s, typename G::node_handle e)
Do bfs/dfs on the graph from given node handlers to the the path. Inputs the graph and two node handlers of source and target nodes. Return the path between them using bfs/dfs.

4 For Developers

Our library supports expansion for different graph types. The virtual class **Graph** has several virtual functions, which are basic operations for any types of graphs. Developers are able to add up any new Graph like tree or undirected graph or graph with capacity from this base **Graph**, which is quite convenient and simple to operate on our clear hierarchy structure. Please refer to our tutorial for examples in expanding the structure.