C++ Generic Graph Library Design Report

Chenqin Xu
`cx2198@columbia.edu`

Kaimao Yang
`ky2371@columbia.edu`

Mingjie Zhao
`mz2646@columbia.edu`

April 30, 2018

Contents

1 Introduction

We present a generic graph library for C++. With this library, user can manage graph structures much more efficiently. Furthermore, our library is easy to use and extend for C++ developers.

There are many different classifications of graphs. We propose a graph hierarchy, and for different uses, we divide graphs into four major categories. For any sub-category, we implemented graph maintenance and visiting functions. For now, we implemented directed-graphs, and it is simple to implement tree or undirected graphs based on our library.

In addition, we implemented some path algorithms to test our library. These algorithms provide generic interfaces, and are also easy to expand.

2 Assumptions

There are some assumptions in our graph library:

- Nodes and edges must be attached to one and only one specific graph.
- We provide several kinds of graph that are easy to tell from the name, and user should manually choose the best graph according to different applications.

3 Node

User can define the node type they want to use. In most senarios, a node belongs to a graph. Therefore, in our design, a node is created by declaring the graph it belongs to and its information. We use an unique identifier for each node in specific graph. After user call function **insert_node**, the function will return back this unique identifier. Afterwards, this handle is used to access the correspoding node.

For simplicity, we do not have a node structure for individual nodes in graph like most previous work. For an individual node, we only have the user-defined type storing informations and the handle assigned by the graph.

The implementation is generic. We use template for user-defined classes. When user declares a graph, the type of nodes must be defined simultaneously. We use concept to check the input types. A user-defined node type must implement the print function.

Here is what the node is defined in a graph:

```
using node_handle = size_t;
node_handle insert_node (V info);
void erase_node (node_handle handle);
```

4 Edge

In directed graphs, an edge is usually attached with 2 nodes: **start** and **end**. Therefore, similar to node, an edge is always attached to a specific graph. We use similar structure here. There is no standalone structure for edge. Instead, after user adds an edge to a specific graph, an edge handle including both ends is returned back.

User can also defined their own types for edges, and such types must satisfy:

- implemented print function;
- must be addable for cost computation.

Here is what the edge is defined in a graph:

```
using edge_handle = pair<node_handle, node_handle>;
edge_handle insert_edge(node_handle handle1, node_handle handle2,
E info);
void erase_edge(edge_handle e);
```

5 Graph Declarations

5.1 General Design

To our knowledge, the most two common types for graph storage are adjacency matrix and adjacency list. Adjacency matrix is easy to find the edge when we have both ends. However, when the graph is sparse, this storage will introduce much useless space. Adjacency list is better for sparse graphs. Considering this, we divide directed graphs into dense graph and sparse graph.

Furthermore, when we choose containers and algorithms to implement these two structures., to maintain a dynamic node list, we need to keep tracking the validness of those identifiers. The most direct way is to use unordered_map. However, if the node number is fixed, this extra costs can be avoided. Therefore, we divide the graph into graphs with fixed number of Nodes and graphs with variable number of nodes (for short, fixed graphs and variable graphs).

In this way, we have four different kinds of graphs. We use inheritance to show the relationships (Fig. 1). The base graph class is an abstract class. Here we define some fundamental members and functions for all graphs, including,

```
template<typename V, typename E>
    class graph {
    public:
        using node_handle = size_t;
        using edge_handle = pair<size_t, size_t>;
        graph() {}
        virtual ~graph()=0;
        virtual node_handle insert_node(V)=0;
        virtual edge_handle insert_edge(node_handle, node_handle,
        E)=0;
        virtual void erase_edge(edge_handle)=0;
        virtual void print_graph()=0;
```

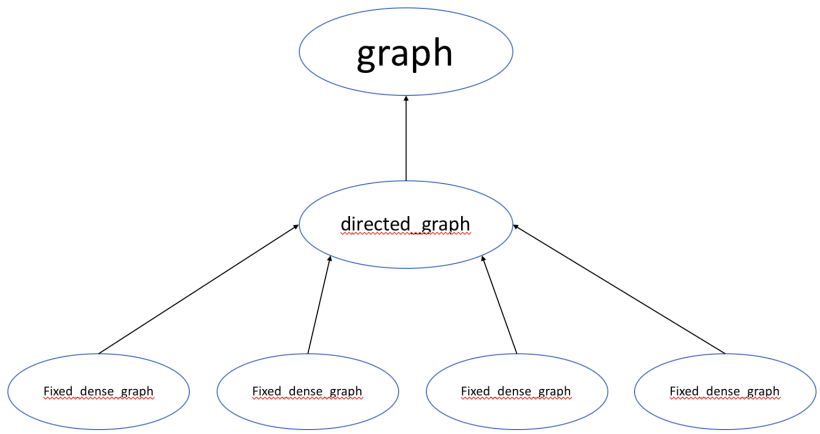

Figure 1: Regression Approach Network Architecture

```
private:
    vector<V> node_vector;
};
```

The **directed_graph** is inherited from base graph class. For now, it does not have any special functions. We put it here is for future inheritance, such as directed acyclic graphs or directed trees.

Next, the four graphs we defined above are all inherited from **directed_graph**. Besides implementing the parent class functions, each graph has other special functions based on the underlying data structures. In the following sections, we will show several aspects of each graph's design.

5.2 Lazy Update Strategy

The lazy update strategy is, when we need to remove a node from **node_vector** or matrix/list, we will not really delete the value. Instead, we have an indicator to check whether this value is valid. And when we need to re-use this space, we can directly update to new value. It is of use in frequently updated graphs.

5.3 Graph Maintenance

For a general graph, we have several fundamental functions to maintain it.

5.3.1 Constructor

The constructor of a graph need to set some variable values and allocate space for the matrix/list.

5.3.2 Insert Node

Because we use a unique identifier for each node, the first step to insert a node is to generate this identifier. After that, the node gets pushed into the graph, where we use a **vector** to save the information of each node. This function returns back this identifier as the handler. The general interface is,

```
node_handle insert_node (V);
```

5.3.3 Erase Node

Erasing node is only allowed in variable graphs. The interface is,

```
void erase_node (node_handle);
```

5.3.4 Insert Edge

To insert an edge, we need to have the both ends of it and the additional information, so the general interface is,

```
edge_handle insert_edge (node_handle, node_handle, E);
```

5.3.5 Erase Edge

Erasing an edge is just the reversion of inserting an edge, so the general interface is,

```
void erase_edge (edge_handle);
```

5.4 Graph Visiting

To implement the graph algorithm, the graph visiting is the most frequency operation. It is necessary for us to design a similar interface for all graph. Visiting a graph includes visiting nodes and visiting edges, and fortunately, there is an approach to design a similar interfaces.

The answer is to use iterator. No matter to visit all the nodes in a graph or to visit all the out-edges of a node, we need to use for-loop to access each element. It is natural to use an iterator like **vector**.

Here we implemented the range-for expression. To execute a range-for, all the operations needed are ***, !=, ++**. In this way, the interface of an iterator is,

```
struct iterator {
        //type members;
        bool operator!=(iterator it) {}
        iterator& operator++() {}
        type operator*() {}
};
```

Then, we need to define a container to use this iterator, and the interface of this container is like,

```
struct container {
        iterator begin()const {}
        iterator end()const {}
};
```

The implementation of this container is different for each graph we proposed.

6 Graph Storage

6.1 Dense Graph

We use adjacency matrix to store graph, because the graph is dense. The element in this matrix is the information of the edge. We use 2-d **vector** to store this structure, so the declaration is,

```
vector<vector<E>> adj_matrix;
```

Here the type E is defined by user.

To mark the validness of each edge, we have another matrix as the flag matrix declared as,

```
vector<vector<bool>> matrix_flag;
```

The increment operation for edges iterator is a litte complicated, because some edges in the adjacency matrix are invalid. We need to check the flag indicator to find next valid edge.

6.2 Sparse Graph

For any sparse graph, using an adjacency list is more efficient than an adjacency matrix. It is natural to come up with idea using **unordered_map** because the row in adjacency list is not fixed. However, as we know, in C++, the underlying data structure of **unordered_map** is a hash table. Besides, it also needs strategy to avoid conflicts. Therefore, we figured a simpler way just using **vector**. The interface is,

```
vector<vector<edge>> adj_list;
```

Here **edge** is a nested **struct** in **graph**. For the adjacency list, in addition to the information, we also need to store **end** of each edge. A nested **struct** is convenient.

The constructor of sparse graph will not allocate space. Instead, when inserting a node, we push a vector into **adj_list**.

Inserting an edge is just to push a **edge** to the vector corresponding to the start node. However, when we want erase an edge, we need to traverse this whole vector. This method is expected because the structure of adjacency list.

6.3 Fixed Graph

Sometimes, using a graph with fixed number of nodes is enough, while the maintenance cost is much smaller. Therefore, it is necessary for us to design a time-efficient class for such needs.

The constructor will have an argument that decides the number of nodes that cannot be changed later. Then, it allocates space for**node_vector** and adjacency matrix/list.

In fixed graphs, adding node or adding edge are easy. When we need to erase an edge, the laze update policy is applied. Erasing nodes are not allowed.

6.4 Variable Graph

The design is more complicated for variable graph. Firstly, when we need to erase a node, we need to remove all the edges related to this node. Secondly, after we remove a node, we need to figure an efficient approach to recycle this identifier and manage the resources.

To solve these two problem, we have two policies:

- Lazy update policy: Like we talked before, all the erasement of an edge use lazy update policy, here is the same situation. After we erasing the node, we will mark this identifier invalid as well as all the edges related to it. This lazy update policy will have O(E) running time. We will not dynamically resize the matrix by ourselves, since **vector** can do this resource management efficiently. Then, when a new node is added afterwards, there will be no extra costs.

- Recycling and Mapping: This is also another trick. In a graph, the unique identifier is generated one-by-one, which means this will not be recycled. However, for the matrix, it is impossible for us to maintain a large 2-d **vector** with many zero values. Therefore, we must find a way to let the new node to use these empty location generated by erased nodes. Then, we came up with this idea. We use a map here to map the identifier (not recycled) to the real index stored in the graph. Similarly, we can use **vector** to replace this map, but map is better to handle frequent erasing and inserting. With this map, we can let the index used repeatedly. To get the re-used index, we use **stack**, so we can get the index in O(1) time.

7 Algorithm Design

Our implemented algorithms are all generic. We use template and concept.

For BFS and DFS algorithm, in fact it only differs in the container type. Besides, in these algorithms, the operations related to the containers are just **top(), oop(), push()**. In this way, we can use a generic template for these two algorithms that accept an argument indicating the container type, then different algorithm can be achieved using same code. The generic interface is,

```
bool generic_pathexists(G g, typename G::node_handle s, typename
G::node_handle e, C& container) {}
```

and the BFS is just,

```
bool bfs_pathexists(G g, typename G::node_handle s, typename
G::node_handle e) {
    _queue<typename G::node_handle> q;
    return generic_pathexists(g, s, e, q);
}
```

DFS is,

```
bool dfs_pathexists(G g, typename G::node_handle s, typename
G::node_handle e) {
    stack<typename G::node_handle> q;
    return generic_pathexists(g, s, e, q);
}
```

It is easy for this template to be used in other algorithm just by changing the container type.

We also implemented a shortest path algorithm using Bellman-Ford algorithm, that is also very easy to implement with our graph library.

8 For Developers

Our library is very friendly to developers from three aspects:

- simple interface and short codes: we provide simple interfaces in graphs and algorithms, so the developers will never need to remember long argument list or types.
- graph hierachy: the hierachy we provided make expanding our base library to more functional library easy and achievable. Developers can add new class to our graph hierachy.
- generic programming: our most codes are generic, that means developers can easily reused our codes in a wide variety of situations.

9 Future Work

Our library still can be improved in these ways:

- add some exception handling
- add graph visualization
- use **array** to substitute **vector** for fixed graphs: thanks for our classmates pointing this out in our presentation!