COMS 4995 – DESIGN USING C++

ASSIGNMENT 5 – GRAPH LIBRARY

SRINIDHI SRINIVASA RAGHAVAN

UNI: ss5145

I. DESCRIPTION OF THE ARCHITECTURE

This architecture makes use of three main concepts- one each for Graph, DAG and Tree.

GRAPH –

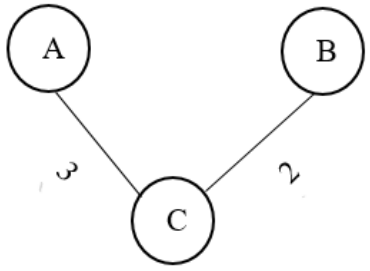
A graph needs a vertex as a requirement. Having an edge is not a requirement of graph, as there can be graphs with just vertices. This is known as a Forest, but there can be no edge without any vertices. Thus, we need something like:

```
template <typename G>
concept bool Graph = requires(G) {
        typename Vertex;
}
```

Every Vertex has an ID and a Value. A Value is simply a pair of String and Integer. The ID represents a unique number associated with the Vertex of the Graph. A smart pointer is associated with every Vertex. Each Vertex in a graph can have multiple parents and multiple children. Thus, the pointer used here is a smart pointer as there is shared ownership between vertices. Similarly, Every Edge of the Graph has an ID, two Vertices – start, end and a weight. A class is used to compile this information of Vertices and Edges.

The graph is represented using adjacency list. Adjacency list is used as it efficiently represents even sparse graphs. The adjacency list is denoted using a Map. A Map is used as retrieving elements based on Key Values is easier in a Map. The Keys are the ID's of the vertices and the corresponding value is a vector of pair of shared pointers of the adjacent vertices with the weight of the edges. Note that, being an undirected graph, there is some redundancy in this architecture. However, adjacency lists prove the most efficient for Directed Graphs.

Consider the following example:



Let the ID's of Vertices A, B and C be 1, 2 and 3 respectively. Then the graph in this representation is given by:

KEY	VALUE
1	(pointer to 1, 0), (pointer to 3, 3)
2	(pointer to 2, 0), (pointer to 3, 2)
3	(pointer to 3, 0), (pointer to 1, 3), (pointer to 2, 2)

Note that every vertex is adjacent to itself. Hence, a pointer to the vertex itself is also added in the vector of adjacent vertices. However, the weight of that edge is zero as there might not be any self-loops. If there is a self-loop in the graph, then the weight of the edge is replaced with the weight of the edge in the Graph.

The working of some of the functions under this architecture is as given:

1. ADD_VERTEX (Graph G, Vertex V): Creates a Vertex pointer to the vertex and adds it to the map with the ID of the Vertex as the Key and a list with the element (pointer to the Vertex, 0) as the Value

2. ADD_EDGE:
    a. ADD_EDGE (Graph G, Vertex V1, Vertex V2): This function assumes that both the vertices already exist in the Graph. It finds the corresponding vector of adjacent vertices of V1 and adds V2 with weight 1 to it and similarly, finds that of V2 and adds V1 to it with a weight of 1. As the weight of the edge is not specified, 1 is used as the default value

    b. ADD_EDGE (Graph G, Edge E): This function simply finds the start and end vertices of the edge and adds it to the map in a similar way as above, with the difference that weight of the edge is substituted with the weight of the edge as mentioned in the Edge

3. TOP: As a Graph, does not have any special Vertex as Top, we simply return the top-most element in the Map

4. REMOVE (Graph G, Vertex V): It deletes the Shared Pointer. Deleting the pointer removes the dependencies to the pointer and thereby deleting the edges with which the other vertices are connected to it

The other functions are unambiguous and can be implemented in a similar fashion

TREE:

A Tree is a very constrained version of the Graph. A concept is defined to find the predicates of the Tree. This concept makes use of the function is_a_Tree. For keeping the architecture simple, the function is not implemented in the fullest way.

A unique pointer of each vertex is used in a Tree. This is because every node has just one parent, thereby avoiding shared membership. We need to define the term adjacent for the Tree. For any edge (a, b), the

node b is said to be adjacent to node a. The definition is not true otherwise as the root does not have a parent

Note that a Tree is created with the Root and the Root is stored separately for efficient retrieval

Some of the descriptions of the functions is as given:

1. ADD_VERTEX (Tree T, Vertex V): Creates a Vertex pointer to the vertex and adds it to the map with the ID of the Vertex as the Key and the corresponding value is an empty list

2. ADD_EDGE:
    a. ADD_EDGE (Tree T, Vertex V1, Vertex V2): This function assumes that both the vertices already exist in the Tree. It first finds if V2 has any other parent than V1. It also finds if there is any vertex V3 such that there is an edge (V2, V3) and (V3, V1). If no such vertex exists, then it implies that there is no cycle. As both the conditions of the Tree are satisfied, the Edge is added in a similar way as that in a Graph. Note that Vertex V2 should not be the root

    b. ADD_EDGE (Tree T, Edge E): The Edge is added only when both the above conditions are satisfied

3. TOP: The top of the Tree is the root. The root is entered

DAG:

A DAG is a type of Graph which is directed with no cycles. Just like a Tree, a concept is defined for DAG but the implementation of the function to test if a Given Graph is a DAG is not implemented

A shared pointer of each vertex is used in a DAG. This is because any node can have multiple parents. The definition of adjacency is the same as that in a Tree.

Some of the descriptions of the functions is as given:

1. ADD_VERTEX (DAG G, Vertex V): Creates a Vertex pointer to the vertex and adds it to the map with the ID of the Vertex as the Key and the corresponding value is an empty list

2. ADD_EDGE:
    a. ADD_EDGE (DAG G, Vertex V1, Vertex V2): This function assumes that both the vertices already exist in the DAG. It finds if there is any vertex V3 such that there is an edge (V2, V3) and (V3, V1). If no such vertex exists, then it implies that there is no cycle and thus, the Edge is added in a similar way as that in a Graph.

b. ADD_EDGE (DAG G, Edge E): The Edge is added only when both the above conditions are satisfied

3. TOP: There is no special node known as the Top, hence, its implementation is similar to that of Graph

Note that for both DAG and a Tree, the list of adjacent nodes to the Vertex V, does not include V as both do not allow self-loops.

II. REASONS WHY THE ARCHITECTURE WON'T HAVE ANY LEAKS

There is no memory leak in this architecture due to the following reasons:

- If a Vertex is removed, then all the dependencies to its pointer are removed
- The right smart pointers are used for the three data structures

III. DESCRIPTION OF THE IMPLEMENTATION

A genuine attempt was made in implementing the above architecture. But due to various errors, I could not go ahead with it. So instead of implementing it as mentioned, I used an implementation without any pointers for the same. Both the codes – one in which an attempt to solve the given architecture and the other in which a simpler implementation is done are attached.