University of Southern Denmark

Department of Mathematics and Computer Science

DM852: Introduction to Generic Programming

Final Project

Authors

Emil Hansen
emiln19@student.sdu.dk

June 18, 2022

Contents

1 Introduction

This project is about developing a generic graph library in *c++*. The graph model which is implemented in the library is an adjacency list, and it is built such that it takes advantage of some of the new features in *c++20*. In addition to the adjacency list, a depth-first search algorithm and a topology call sorting algorithm where to be developed as well, which both should make use of the graph model.
For the development of the adjacency list, a set of the new concepts, from *c++20*, was given which the model had use.

2 Design choices

2.1 Adjacency List

This is a quick overview of how the adjacency list is structured. To model a graph it needs both vertices and edges, both of which will be stored in a list. These two list is the main storage of the model. For the edges, they consists of both a source and a target, which the edge will go in between of. To indicate which vertices that it is going in between, the index of the vertices in the vertex list is used. Then if you want to know which vertex is the source or target, you can just look up the vertex list with the index. For vertices, they can have any number of edges going to them and out of them, meaning that the vertex is either the target or source respectively of the edge. To store this the vertex will have a list of edges, where it will also store the index of the edge from the edge list.

Further additions is also made to the structure, but this is described in the later sections of this paper.

Descriptors for both the vertices and edges is also implemented. These are made to make a default type for vertices and edges for the library. For the vertices they are simply just a *size_t*, which indicated the index of the vertex in the adjacency list. For the edges they have three *size_t*. Two of them which indicate the source and target index of the vertices, and the last for its own index in the adjacency list.

2.2 Direction

The graph should be able to have one of two directions. This is either *directed* or *bidirectional*. The adjacency class has a template parameter *DirectedCategoryT* for indicating which of the tags that the graph should have. For the tags, two empty class with the respective names is made, and their only purpose is to be place as the template parameter.

The difference of a directed graph and a bidirectional graph, is that the bidirectional graph has both in-edges and out-edges for its vertices, whereas a directed graph only

has out-edges. This means that two different data types is needed, depending on which direction the graph has. This is done by having a *StoredVertexSimple* class, which consists of a list for out-edges, and a StoredVertexComplex class which is deriving from the *StoredVertexSimple* and additionally it has a list for in-edges too. Then with some meta-programming using the *conditional_t* function, the function can be given the *is_same* trait class to check which of the directions the graph was given. Depending on which direction it was, the adjacency list will use the proper class as the *StoredVertex* type.

Now for adding a new edge two different methods are needed depending on the direction of the graph, but both has to have the same parameters. The parameters of the functions are, the source vertex, the target vertex and the graph.

If the graph were to be directed, then a new edge should be constructed with going from the source to the target. This edge is then appended to the end of the graphs edge list, and then afterwards the index of the edge in the list is appended to the source's list of out-edges.

If the graph were to be bidirectional, it does all the same steps. However, in addition it would also have to add the index of the edge to the target vertex's in-edges list.

By having both of these methods, it ensures that no matter which of the directions the graph will have, there is a proper way for adding a new edge. But there's a problem as in choosing which methods will be used, as they have the exact same name *addEdge* and the same parameter types. This can be solved be solved by using tag dispatching. So what is done instead is putting the functionality in two helper methods, which have an additional parameter in the parameter list. This parameter will not be used for anything, but it is used to indicate which of the directions the graph needs to have to call that function. Then from the *addEdge* function, the direction of the graph is retrieved, and then the helper function is called with the direction as an additional parameter. Now the correct implementation of adding an edge is called.

Which both of these implemented added, the adjacency list satisfies both the *Mutable* and *BidirectionalGraph* concepts that were provided.

2.3 Property

Another feature that the graph model should have, is labels for the edges and vertices. To do this 2 additional template parameters is added to the structure, as the labels does not have to be of the same type. The template parameters are *VertexPropT* and *EdgePropT*. To implement both of these, the edge and vertex structures have to be changed. What is done is adding a data property for both the edges and vertices, which have the type *EdgePropT* and *VertexPropT* respectively. This property can now be used for labeling the edges and vertices. However, the user might not want any labels in the graph, so

the template parameters are defaulted to an empty class *NoProp*. By using this class by default, it minimizes the size as it just a single byte.

A restriction is placed on the template parameters to be that they have to default initializable. This is done to make sure that a graph of n vertices can be generated directly from the constructor. To make sure that the template parameters is default initilizable the concept *default_initializable* is used, which just takes in the template parameter and returns true if it can be default initialized.

New *addVertex* and *addEdge* functions have to be made, as the user might want to create them with a label. This can be done as the same way as the previous implemented functions of *addVertex* and *addEdge* is implemented, as their parameter list will be different.

An additional feature is also implemented to quickly access the label for either a vertex or a label. This is done by overloading the subscript operator, so if it is provided with a *VertexDescriptor* it will provide the label for that vertex by looking up the index in the vertex list. Likewise is done for the *EdgeDescriptor*. This makes it so that the label can easily be retrieved, and changed. To keep correct constness const variations are also implemented, such that if the adjacencylist is initalized as a const the labels can not be changed.

With this implemented, the adjacency list satisfies both the *PropertyGraph* and the *MutablePropertyGraph* concepts.

2.4 Iterators and ranges

To retrieve multiple parts of the adjacency list and iterator over the different parts of the adjacency list, different ranges are implemented. These range can then return iterators, that will iterator over different parts of the adjacency list. There is therefore 4 different ranges implemented, the *EdgeRange*, VertexRange, *OutEdgeRange* and *InEdgeRange*. They all do exactly what they are called. EdgeRange consists of all the edges in the graph, VertexRange consists of all the vertices graph, OutEdgeRange consists of all the out-edges of a vertex and InEdgeRange consists of all the in-edges of a vertex.

To make the implementation part easy, the boost::iterator library is used. It provides a framework for easily building iterators, by using or deriving from one of their iterators, and only implementing a few methods. For VertexRange the *counting_iterator* is used, as it simply just have to count because the *VertexDescriptor* is just a number. However for the other three iterators, it is a little more complicated, as they all should return *EdgeDescriptor* but none of the lists consists of this type. What is done here instead is using the *iterator_adaptor*, which allows us to decided which type the iterators should be dereferenced to. This enables our library to convert all the different ranges to *EdgeDescriptor*. This is mostly what is done. When initializing one of the ranges the list is given as a reference, such that it is able to tell how far in the list that it has reached. This list is of

course dependent on which range class it is. All the other methods is handled by the boost library.

2.5 Copy and movable

For convenience and efficiency both copy and move constructors and assignments are implemented. The move constructor and assignment is the simple implementation where it moves the resources from one object to the other, to save resources on the heap and time. It does this by moving the two lists to the other object. The copy constructor and assignment does the copying from one object to the other, my going through each item in the list and creating an identical item in the other. However a problem occurs once again here where needs to add edges and vertices, because the adjacency list that it is copying from might or might not have properties, meaning that it is not known which method of adding is needed to be called. Therefore tag dispatching is once again used, to implement both methods for both vertices and edges, such that the correct implementation of *addEdge* and *addVertex* is called by having the correct number of parameters.

3 Discussion

3.1 Implementation

With all that is implemented as of right now, is like everything goes. In this I mean everything is accessible for everything, also things that shouldn't be accessible. As an example the methods for getting the number of in-edges for a vertex, can be called on a graph which is directed. This shouldn't be possible.

The way of handling this is done with requires concept at the functions. Here the way example from above, I've used the *same_as* concept and checked if the *DirectedCategory* of the adjacency list is *Bidirectional*. Then if it is, it will evaluate to true and only if it evaluates to true the function will be adjacency list. This means that the if the adjacency list is *Directed*, then these function won't be part of the adjacency list.

The same concept is done with the functions that requires the adjacency list to have a have property on either the vertex or edge. Here instead it compares the *VertexProp* or *EdgeProp* with *NoProp*, and if they are not the same it is evaluated to true.

The last place where the requires concept method is used is on the adjacency list's *EdgeProp* and *VertexProp* template parameter. Here it checks if both are copyable and default initializable, as both are criterias for the adjacency list. Copyable is wanted as it is wanted it both ensures that the type can both be copied and moved. For default initilizable is wanted for the vertex property, as the adjacency list should be constructable with a number and then default initialize that number of vertices.

3.2 Tests

For testing the implementation of the adjacency list, I've made some unit tests to check if the implementation I've made is correct. All of these are sectioned up into the different concepts, for the different graph types. It is just a few unit tests I've made to verify that they are correct. This could always have been done in a greater case, but as the implementations is quite simple I've decided against in developing a huge amount of unit tests.

Another form of testing is the development of the depth-first search and topological sort. Here the implementation was easy as the book *Introduction to Algorithms* has pseudo-code for the algorithms. Then running the algorithms on different adjacency list graphs also helped in testing the implementation of the adjacency list.

The last method of testing I've done is doing *static_assert* for the different concepts on different types of adjacency lists. This is done to verify that some of adjacency lists don't have access to functions that is part of a concept that itself is not part of, and that all the graphs have the correct functions.

4 Conclusion

With all of these implementations, I've implemented a generic graph library which is made with a adjacency list structure. It follows the concepts which were provided, and has all the necessary functionalities.