DM852 Generic Programming Final Project

Kevin Joshua Vinther
kevin20
060201

June 2023

1 Introduction

Today, graphs are used in many different areas: networks, databases, and many more. In order to represent graphs, we need to be able to represent the edges and vertices. One way to do this is by using an adjacency list.

In this project, we will implement a generic graph library using an adjacency list. Furthermore, we will implement a depth first search algorithm, and a topological sorting algorithm using the graph.

The generic graph library will be written in C++ and will be restricted to use only the C++ standard library, as well as the Boost Iterator library.

The graph library should be generic, such that it can be used for any type of graph. Furthermore, vertices and edges should be able to hold any type of data, which the user will define themself. The graph should support three different types: directed, undirected and bidirectional.

The goal of this project has been reached when the generic graph library has been implemented, along with the depth first search and topological sorting algorithms.

The code is based on code delivered for this project specifically, thus giving a starting point from which to build on.

2 Design Choices

2.1 Graph

The graph will be implemented as an adjacency list. This means that, depending on the type of graph, each vertex will hold a list of edges.

The generality of the graph library is very important for the project. Thus, the graph should be able to:

1. Hold any type of data
2. Be either directed, undirected or bidirectional

We allow both the vertices and edges to hold any type of data. This will allow them to be used for many different purposes, and not just for the purposes presented in this project (depth first search and topological sorting.) For example, a user would be able to implement weights on the edges, or store usernames on the vertices. There are many possibilities.

We use descriptors for the vertices and the edges to allow for easy handling with the graph. This means, that instead of using the vertices and edges directly, the descriptors act as a type of pointer to the vertices and edges.

An edge should be able to hold information about the soruce and the target vertex. Furthermore, the edge descriptors and vertex descriptors should be identifiable by a unique id.

2.2 Directed, Bidirectional and Undirected

As mentioned before, the graph should be able to support three types of graphs: directed, undirected and bidirectional graphs.

By supporting this, we allow for the graph to be used in multiple different ways.

For the directed graph, each vertex supports a list of edges going out from the vertex. For the bidirectional graph, each vertex supports both a list of edges going out from the vertex, and a list of edges going in to the vertex. For the undirected graph, each two vertices have an out edge to each other. Thus creating an "undirected" graph, where both vertices are connected through out edges.

2.3 Depth First Search

The depth first search algorithm is implemented recursively. It uses visitor pattern to allow for different actions to be taken when a vertex is discovered, finished, or when an edge is traversed. This also allows for an easy way to use the depth first search for other purposes, such as topological sorting.

2.4 Topological Sorting

The topological sorting algorithm is implemented using the depth first search algorithm.

3 Implementation

3.1 Graph

Template Parameters
The generality of the graph is implemented using templates. The parameters for the template are *DirectedCategoryT*, *VertexPropertyT* and *EdgePropertyT*. *DirectedCategoryT* is the type of graph we are using, it can either be *Directed*, *Undirected* or *Bidirectional*. We implement these tags for the direction in the

`graph::tags` namespace. Each of the tags are implemented as structs, with the bidirectional tag inheriting from the directed tag. *VertexPropertyT* and *EdgePropertyT* are the types of the properties which the vertices and edges hold. This is what allows for the edges and vertex to hold values, which the user will define themself.

Adjacency List & Overview
The graph is implemented using an adjacency list. This is represented through the struct *AdjacencyList*.

The adjacency list struct has multiple sub types:

- StoredVertex
  - StoredVertexDirected
  - StoredVertexBidirectional
- StoredEdge
- EdgeDescriptor
- OutEdge
- InEdge
- VertexRange
- EdgeRange
- OutEdgeRange
- InEdgeRange

Vertex
Vertices are represented as the struct *StoredVertex*. If the graph is directed or undirected, the struct *StoredVertexDirected* is used, otherwise the struct *StoredVertexBidirectional* is used. The way this works, is by conditionally using the *using* keyword to define the type *StoredVertex*. This is done using `std::conditional_t` to check which type of graph we are using, and then using the correct struct depending on the result. We could have implemented this using traditional tag dispatching, but this made a cleaner implementation. Using `std::conditional_t` allows us to, at compile-time, decide which struct to use. This gives a very efficient implementation.

The vertex struct holds a list of edges. In the case where it is a directed or undirected graph, it only holds out-edges, but in case it is bidirectional, it holds both in- and out-edges. The reason for the undirected graph holding out-edges, is an implementation decision which has been made in order to make an undirected graph work, without using more types than necessary. Thus, in an undirected graph, both vertices connected by an edge see the edge as an out-edge.

VertexDescriptor
As mentioned in Design Choices, we use descriptors for vertices and edges. For VertexDescriptor we simply use an integer (`std::size_t`) to represent the vertex descriptor. As was the purpose, this allows for easier use of the graph, as the user does not need to understand the implementation deeply.

Edge
Edges are represented as the struct *StoredEdge*. This struct holds the source and target vertex descriptors, as well as the edge property. In the default constructor fot the *StoredEdge*, the *EdgeProp* is compared to *NoProp* at runtime, to check if a new *EdgeProp* needs to be initialized. The stored edge is much simpler than the stored vertex, as it does matter which type of graph we are using, it only needs to know which is the source vertex, and which is the target vertex.

EdgeDescriptor
The edge descriptor is not as simple as the vertex descriptor. Instead of simply being an integer, it is a struct containing the source and target vertex descriptors, as well as the identifier for the edge, which is an integer.

InEdge and OutEdge
We implement two types of edges; one for in-edges, and one for out-edges. This is much like StoredEdge, except it only holds one vertex description: the source or target (depending on the type of edge.)

Vertex and Edge Ranges
Both the vertices and edges have been implemented as ranges. This allows for many uses, such as for-each loops and general iterator use. This, for example, allows for us to easily implement the depth first search algorithm, as it iterates over the vertices and edges through the graph. We implement both the vertex- and edge ranges using the boost library, but since the vertex is much simpler, we can just use an existing iterator, while the edge range needs to be implemented as a custom iterator.

OutEdgeRange and InEdgeRange
In addition to VertexRange and EdgeRange, we also implement OutEdgeRange and InEdgeRange. These ranges give additional functionality, and allow for us to implement functions such as *outEdges*, which returns a range of all out edges from a Vertex. The same has been done for *InEdgeRange* which returns the in edges for a given vertex. Furthermore, we implement two edge ranges: *OutEdgeRange* and *InEdgeRange*. This allows for the user to iterate over the out-edges and in-edges of a vertex, respectively. These can be used to find the number of in-edges and out-edges of a vertex, as well as iterating over them.

constexpr

Multiple times in the implemenation, we use the *constexpr* keyword in *if* statements. This allows for compile-time evaluation of *if* statements. We do it, because it is needed for `std::is_same`, which is used multiple times to check the type of graph, and whether or not the edge or vertex has a property.

addEdge
One of the functions in which *constexpr* is used is in the *addEdge* function. When adding an edge, we need to know the type of graph, as to create the edge correctly. If the graph is directed we only set an out edge from the source vertex to the target vertex. If the graph is undirected, we additionally set one more out edge from the target to the source graph (this has been discussed in design choices.) If the graph is bidirectional, we set an *in* edge from the target vertex to the source vertex, as per the requirements of a bidirectional graph.

StoredVertex
In the constructor for *StoredVertex*, whether it is the Directed or Bidirectional vertex, also uses *constexpr* to, at compile time, check whether or not the VertexProp is given to be something else than *NoProp*, which is the default.

Two functions for addEdge and addVertex
In the graph interface, we have two functions for adding edges and vertices. One of them takes a property, and the other does not. The reason for this is that we want to be able to add edges and vertices without properties, but we also want to be able to add them with properties. The compiler will choose the correct function, depending on whether or not the user has given a property.

Overloading the [] operator
The [] operator is used to access a vertex prop or an edge prop from the graph. This has been done by overloading the [] operator for the vertex and edge ranges. Thus, the user can use the [] operator on a graph to access the vertex prop or the edge prop because of this overloading.

3.2 Depth First Search

For depth first search, we take heavy inspiration from the pseudocode in Appendix A. This provides us with both the function *DFS* and *DFS-VISIT*. Thus, this implementation uses visitor pattern, as the pseudocode does. The visitor design pattern allows for us to implement various functions which are called at different points in the algorithm. These functions are called by the algorithm, and thus the user can implement them and do whatever they want in their implementation.

The visitor pattern is very important for our implementation of topological sorting, as we will see later.

The most important part about the implementation of depth first search is the *dfsVisit* function. This function is called recursively, and it is the function which actually does the depth first search. It is called by the *dfs* function, which

sets up the colour of the vertices, and calls *dfsVisit* on all vertices which are white (as, if a vertex is white, it is because it has not yet been processed).

3.3 Topological Sorting

We implement topological sorting using the depth first search algorithm. We do this by overriding the *finishVertex* function, which is a part of the visitor pattern.

The *finishVertex* function uses an iterator, *iter* which is an argument to the function to insert the vertex into the beginning of a list. This list becomes the sorted list of vertices.

We sort it by using the depth-first search with the new *finishVertex* function. As it is added recursively, the vertices are added in reverse order.

4 Evaluation and Testing

In order to test whether or not the implementation works as intended, I have written a number of tests. These tests check whether specific functions work as intended. The tests are:

- `test_directed_graph_creation()` tests if you can successfully create a directed graph
- `test_bidirectional_graph_creation()` test if you can succesfully create a bidirectional graph
- `test_undirected_graph_creation()` test if you can succesfully create an undirected graph
- `test_topo_sort()` tests if the topological sorting works as intended, i.e. if it returns the correct order of vertices
- `test_getIndex()` tests the *getIndex()* function
- `test_default_constructor()` tests if you can create a graph with the default constructor succesfully
- `test_copyable` tests if the graph is copyable

5 Conclusion

In this project we have implemented a generic graph library using the adjacency list representation. This implementation allows for the creation of generic graphs, with or without properties, which can be both directed, undirected and bidirectional. Using this graph library we have implement depth first search, and in turn using depth first search, we have implemented topological sorting. When testing the implementation, I have unfortunately not been able to get

the tests to work as intended. I was not able to find the cause because of time constraints, and the error messages seemed rather cryptic. However, although these did not work as intended, I believe the cause to be a minor one, as the implementation theoretically should work.

A Depth First Search Algorithm

```
DFS(G, Visitor):
        initialize 'colour'
        for each vertex u in V:
            colour[u] := WHITE
            Visitor.initVertex(u, G)
        end for
        for each vertex u in V:
            if colour[u] = WHITE:
                Visitor.startVertex(u, G)
                DFS-VISIT(G, Visitor, colour, u)
        end for

DFS-VISIT(G, Visitor, colour, u):
        Visitor.discoverVertex(u, G)
        colour[u] := GRAY
        for each v in Adj[u]:
            Visitor.examineEdge((u, v), G)
            if(colour[v] = WHITE):
                Visitor.treeEdge((u, v), G)
                DFS-VISIT(G, Visitor, colour, v)
            else if(colour[v] = GRAY):
                Visitor.backEdge((u, v), G)
            else if(colour[v] = BLACK):
                if d[u] < d[v]:
                    Visitor.forwardOrCrossEdge((u,v), G)
                Visitor.finishEdge((u, v), G)
        end for
        colour[u] := BLACK
        Visitor.finishVertex(u, G)
```

Figure 1: Depth First Search Algorithm