DM852 Exam project

Niels Peter Roest[1 []]

SDU Odense

Abstract. In this paper i will describe my solution to the final project in DM852 [1]

Keywords: Computer Science · Generics · Programming.

1 Intro

In this project, we had to implement graph functionality and 2 graph algorithms, namely Depth first search (DFS) and Topological sort. [1].

2 Definitions

Let's start with the definitions of a Vertex and Edge. For this project, we were given an unfinished Adjacency List (AL), and allowed to use this as a base for our solution. In my AL, verticies are defined by their index in the verticies list. So if a verticies is named 0, its index 0 in the list. A vertice contains a list of outgoing edges (OutEdgeList), potentially a user-defined property, and potentially incoming edges (for bidirectional). Firstly, an OutEdge is defined to be the index of an edge relation (StoredEdge), and OutEdgeList is defined to be a vector of OutEdges. An edge relation is a struct containing a source and target vertice, and its refered to by its index in the edges list.

So the vertex stores edges, and for this assignment, it had to store variable amounts of information based on user settings. So to facilitate this, i used a "std::contitional_t" which checks if the user selected this to be a directed or bidirectional list at compile time. If it was directed, we only store outgoing edges (StoredVertexSimple), but if its bidirectional, we select a struct with both outgoing and ingoing verticies (StoredVertexComplex). The output of this is the struct StoredVertexT. I then again use the conditional to check for custom properties, if the user wanted to store an additional property in the vertice, we use definition StoredVertexE, which inherits StoredVertexT, and has a custom property. Otherwise it just uses StoredVertexT with no custom properties.

For users, a vertex is returned as a VertexDescriptor, this VertexDescriptor is just the index in the vertex list. Additionally, an edge is returned as a EdgeDescriptor, this is a struct that contains a source, target and the edge relations index. The astute reader will have noticed that verticies storing edge relations is complicated, since you have to now lookup the relation, and filter by if you are source or target, to get the adjacent vertex. But this is a small price

to pay for O(1) lookup time on edge relations from a vertice, if it didnt do this, you would have to search the edge list when returning an EdgeDescriptor from the OutEdgeList, which is expensive.

3 Constructors

For constructors, the assignment was to have a copy constructor, copy assignment, move constructor, move assignment. Since all the AL stores is two lists (vertice and edge list), it was sufficient to just call std::copy and std::move on said lists.

4 Iterators

There are 2 iterators, an OutEdgeRange, and an InEdgeRange, these two iterators to the exact same thing, except for the InEdgeRange switching the return source and target, so i'll only describe the OutEdgeRange. The OutEdgeRange works by storing an AdjacencyList, and the index of this vertice. It then calls an iterator which simply iterates over the outgoing edges list of the vertice, and returns it. Its important to note that it returns EdgeDescriptors, which means it needs the index of this relation, and is also where the previously mentioned edge definitions is important for fast operation (since we have the index, and can just lookup the target in O(1)).

5 MutableGraph

For a MutableGraph, there is a provided concepts.hpp file which defines what operations should be supported. Namely, addVertex and addEdge. These functions are relatively simple. The function addEdge, creates a new edge between 2 vertices, adds the relation to the list of edge relations, and add's the relation index to their lists of outgoing (and ingoing if bidirectional) edges. The addEdge function also uses tag dispatcching to find the correct function to execute. There are 4 addEdge functions, a normal one that works with directed and nothing else. An additional one with bidirectional, that adds an ingoing edge also. And finally a duplicate of each of these, that also stored a property if applicable. This is checked using the requirement std::same_as, for checking if the VertexProp is not NoProp (where NoProp means no property, so it stores nothing. The normal functions are still callable with a property, if the property is default initializeable, this is checked using the requirement std::default_initializable.

6 IncidenceGraph

For an IncidenceGraph there has to be an outEdges and outDegree function. Where outEdges returns an iterator over outgoing edges, and an outDegree counts the number of outgoing edges. So the outEdges just returns the iterator over OutEdges, and the outDegree just gets the size of the outedges list.

7 BidirectionalGraph

A bidirectional graph also counts ingoing edges. This means that the vertex must also store ingoing edges (implemented as previously described). It must also have an inEdges and inDegree, jus like the IncidenceGraph, and these functions work the same, but for the incoming edges instead.

8 PropertyGraph

A property graph concept describes the need for the array index operator []. So, to facilitate this for both verticies and edges got operators for both non-const and const (a requirement for the PropertyGraph). These operators just access the given index in the edge/vertice list, and return its property. They all require that their respective property isnt NoProp, because then the verticies/edges dont contain the property field (as described in definitions).

9 MutablePropertyGraph

MutableGraph requires that addVertex and addEdge also accept a property field, and adds said property to the vertex. So i have duplicated said functions, which now also set a property. These new functions also have a requires field, which states that the respective property isn't NoProp, so the field actually exists (according to my definitions).

10 Depth first search (DFS)

For DFS, we were given the exact algorithm to implement. Thanks to generics, the algorithm i implemented looked almost the exact same as in the pseudocode. With some exceptions, like having to address source(v) and target(v) instead of the nicer (u,v) that the algorithm uses. But other than the fact that mine is a tiny bit more verbose, its the exact same.

11 Topological sort

In topological sort, we had to record the time that each vertex was visited, and output it to a OutputIterator. Firstly i record the startVertex call, this is because we only record vertex we visit, and we never go back to the original vertex. So to ensure correctness, i always record the absolute first vertex in the startVertex call of the generic visitor. Then each visited node/vertice is recorded in the call treeEdge, such that i record the "time" that i visited each vertex (by time i mean the order of operations in this case). Thats what the assignment wanted us to do, so thats what it does, and it works.

12 Tests

For testing i used a main.cpp file, which mainly tests that the generics are valid by requiring concepts for various configurations of the AL, but also runs the functions and writes their results to the terminal, so the developer can check that they are valid.

13 Conclusion

I haved implemented an Adjacency List that supports a multitude of functionality and configurations with respect to directed and bidirectional graph, and custom properties on verticies and edges. Additionally these configurations are optimized such that not needed data isn't included, such as verticies having no custom property field when a custom property isn't set, saving memory space. Depth first search was also implemented correctly, along with a Toplogical sort algorithm. Everything was tested with respect to concepts being implemented correctly and functionality working as intended.

References

1. Jakob Lykke Andersen. "DM852 Introduction to Generic Programming Spring 2022 Final Project".