Final Project

DM852 Introduction to Generic Programming

Dennis Andersen – deand17

Department of Mathematics and Computer Science
University of Southern Denmark

June 20, 2022

1 Introduction

The goal of the final project has been to design and implement data structures and algorithms similar to those found in the Boost Graph Library (BGL), a powerful generic library of data structures and algorithms useful for working with graphs, but taking advantage of recent changes to C++, such as the addition of concepts.

Starting from a provided initial implementation containing predefined graph concepts, our first task has been to design and implement a generic configurable adjacency list data structure, which satisfies different constraints, modelled as graph concepts, depending on the requirements given by the user. The idea is to provide a generic interface for accessing the structure of a graph, which is also configurable, while hiding implementation details. That is, depending on the graph concepts requested by the user at compile time, additional functionality may or may not be provided, meaning the user only pays for the functionality needed.

In continuation hereof, our second task has been to implement generic versions of the depth-first search and topological sorting algorithms using the visitor pattern. The idea here is to design our algorithms to take advantage of the generic interface provided by our data structures, enabling more opportunities for reusing the algorithms.

The remainder of this report is outlined as follows. We start by discussing design considerations and important design decisions, then proceed to discuss implementation details and lastly, we end with a summary of our findings in this project.

2 Design Considerations and Choices

Our design choices for this project have mainly been guided by the requirements of each task, as well as by the requirements of the graph concept applicable to the respective task. In addition, our aim has been to keep the design as simple as possible. We start by covering the design of the adjacency list before proceeding to the depth-first search and topological sorting algorithms.

We have used the provided initial implementation as a starting point, which gives us a base design to expand upon. From task 1, we have that the adjacency list must be a template class configurable through three template parameters:

- a `DirectedCategoryT` tag parameter, which can be one of
  - `tags::Undirected` indicating that the adjacency list represents an undirected graph,
  - `tags::Directed` indicating that the adjacency list represents a directed graph with access to only out-edges, or

  - `tags::Bidirectional` indicating that the adjacency list represents a directed graph with access to both out- and in-edges.

- a `VertexPropT` parameter defaulting to `graph::NoProp` which specifies whether the adjacency list should store additional data with each vertex, such as for example a label.

- an `EdgePropT` parameter, also defaulting to `graph::NoProp` which specifies whether the adjacency list should store additional data with each edge, such as for example a label or a weight.

Furthermore, we require that the adjacency list can be both default-, copy-, and move constructed, as well as copy- and move assigned. This can of course be done explicitly, as is the case with the already specified default constructor, however we can also have the compiler implicitly define these for us, which is what we have done for the copy- and move constructors and copy- and move assignment operators.

To represent a graph, the base design uses the following abstractions. Internally, the adjacency list stores two vectors, one for vertices and one for edges. A vertex is represented by a struct `StoredVertexSimple`, which contains a vector of out-edges of that vertex, but to the user, a vertex is represented by a `VertexDescriptor`, which is simply a non-negative integer mapping directly to the index at which that particular vertex is stored in the internal vector of vertices. Similarly, an edge is represented internally by a struct `StoredEdgeSimple` that stores two non-negative integers, a source and a target. To the user, an edge is represented by an `EdgeDescriptor`, which additionally includes the index at which that edge is stored in the internal vector of edges. Access to the stored vertices and edges are then done through various accessor functions exposed via the generic interface.

We now proceed with the subtasks of task 1. In the description of task 1a we are informed that in general, the `DirectedCategoryT` type will be one of the three tag types mentioned above or a type derived from one of these, which we can enforce by adding this as a constraint at the top level using the `std::derived_from` concept. We are also informed that we only need to directly support the `tags::Directed` and `tags::Bidirectional` tags and as a result we have not made any efforts to support undirected graphs in our design. As the provided implementation already satisfies the other requirements of task 1a, we can proceed to task 1b.

Task 1b specifies that the adjacency list must satisfy the requirements of the `MutableGraph` concept, which enables vertices and edges to be added to the graph. The signature of the `addVertex(g)` function allows no way of influencing how a vertex is added to the graph, it merely adds an additional vertex to the graph. In other words, there are no preconditions or requirements associated with adding a vertex and calling the function will always simply add another vertex to the graph. This is not the case for the `addEdge(u, v, g)` function however. Obviously, adding an edge from, to, or between non-existing vertices makes no sense. However, instead of designing complicated error handling, we have done as suggested in the task description and assumed that the following preconditions hold: both `u` and `v` are valid vertex descriptors (i.e. vertices that have already been added to the graph) for `g`; that `u` and `v` are different from each other; and that the edge `(u, v)` does not already exist. In other words, this places the responsibility on the user to ensure that the input to the `addEdge(u, v, g)` function is valid.

For task 1c, we have to ensure that the adjacency list satisfies the requirements of the `IncidenceGraph` concept when applicable, that is, when the user requests that the adjacency list model either a `tags::Directed` or `tags::Bidirectional` graph. The `IncidenceGraph` concept specifies two additional required functions: `outEdges(v, g)`, which returns an `OutEdgeRange` for the vertex `v` and `outDegree(v, g)`. In addition, there is also the specific requirement that the iterator for the `OutEdgeRange` models a `std::bidirectional_iterator`.

Starting with the `outEdges(v, g)` function, we simply model an out-edge as a struct `OutEdge` containing the id of the target vertex. The out-edges of a vertex can then be modelled as a vector of `OutEdge`s for each incident edge, as modelled in the provided `StoredVertexSimple`. To then return an `OutEdgeRange`, we simply have to return a range defined by an iterator to the first out-edge of `v` and a past-the-end iterator indicating the end of the out-edge range. Our decision for modelling this has been to simply use the same approach as used in the provided `EdgeRange::iterator`, which leverages the `boost::iterator_adaptor`. The `boost::iterator_adaptor` allows us to specify an underlying iterator, which we can then adapt to serve our desired purpose. In addition, by using an underlying data structure which supports for example a `std::random_access_iterator`, as is the case with the vector we use to store out-edges, we can have the iterator we adapt also support random access. And because a `std::random_access_iterator` is also a `std::bidirectional_`- `iterator`, we thereby satisfy that requirement. The design of the `outDegree(v, g)` function then follows directly from this, as it simply needs to return the size of the vector storing the out-edges of vertex `v`.

Building upon task 1c, task 1d introduces the `BidirectionalGraph` concept, which specifies the in-edge counterparts to `outEdges(v, g)` and `outDegree(v, g)` of a vertex `v`, namely: `inEdges(v, g)`, which returns an `InEdgeRange` for the vertex `v` and `inDegree(v, g)`, with the same requirements as their out-edge counterparts. That is, the `InEdgeRange::iterator` should model a `std::bidirectional_iterator`. However, this added functionality is only required when the `tags::Bidirectional` tag is given, so to ensure this we have followed the steps outlined in the task and added a variation of `StoredVertexSimple`. This variation holds an additional vector of in-edges, with an in-edge being defined in the same way as an out-edge, but specifying the source of the in-edge and using a different name to distinguish the two. To enable selection of the appropriate functionality depending on the given tag, we can use partial specialization together with a dummy template parameter to specify which variant of `StoredVertexSimple` to use. We can then define two internal variants of the `addEdge` function, one specific to each variant of `StoredVertexSimple` and have the public variant of `addEdge` choose the appropriate function using tag dispatching. Lastly, for the `InEdgeRange::iterator` we have used the exact same approach as described above for the `OutEdgeRange::iterator`. Although this appears to result in code duplication, the common part of each range iterator is actually the use of the `boost::iterator_adaptor`, from which they all inherit. In other words, this is more a case of specialization than of code duplication.

The final feature our adjacency list needs to support is introduced in task 1e and is to use the two template parameters `VertexPropT` and `EdgePropT` to allow the user to specify the type of object that should be stored together with each vertex and edge respectively, and the requirements for this functionality is modelled by the `PropertyGraph` and `MutablePropertyGraph` concepts. However, this functionality should only be enabled if needed, meaning we let both template parameters default to `graph::NoProp`.

Now when `VertexPropT` is different from `graph::NoProp`, each vertex must have an instance of the given type. Similarly, when `EdgePropT` is different from `graph::NoProp`, each edge must have an instance of the given type. As a result, we expand our internal representation of stored vertices with two additional partial specializations, literally copies of what we already have, but that also contain a field for storing a property. We use the same approach and partially specialize our representation of a stored edge to also accommodate an edge property. The appropriate storage method will then be selected depending on the template arguments.

With our modified internal storage setup, we can now proceed to handle the requirements of the `MutablePropertyGraph` concept. The `MutablePropertyGraph` concept requires the addition of property variants of the `addVertex` and `addEdge` functions, that is: `addVertex(vp, g)` for adding a vertex with a property and `addEdge(u, v, ep, g)` for adding an edge with a property. The non-property variants are then only required to be valid if the given property is default constructible.

Because `graph::NoProp` is simply an empty struct, it is default constructible, meaning we can relay this requirement to the user by adding a constraint on the non-property variants that the specified property must be default constructible. For the property variants, we are furthermore informed that it is desirable if we only require a property to be movable. We can handle this by including a constructor specialized for this requirement in each of our stored vertex and stored edge partial specializations and then always pass the property using `std::move` internally. We can then add as a constraint to the property variants that the property is required to be movable.

Accessing and modifying vertex and edge properties is modelled by the `PropertyGraph` concept and this requires us to add overloads for the array subscript operator `operator[]`. That is, for `g[v]` where `v` is a `VertexDescriptor` and `g[e]` where `e` is a `EdgeDescriptor`, we need to return a reference to property of vertex `v` or edge `e` respectively. However, because we internally keep our stored vertices and stored edges in their own respective vectors, the descriptor is, as mentioned in the beginning, exactly the index at which we keep our internal representation of that vertex or edge, meaning we only have to do a lookup in the appropriate vector. In addition, each operator has the precondition that its descriptor must be valid, meaning again we place the responsibility on the user to ensure that the input given to the operators are valid. Similarly, we also do not perform any bounds checking when invoking the operators.

This covers the design of the adjacency list and task 1. For task 2, task 2a and task 2b ask us to implement the depth-first search and topological sorting algorithms respectively using the visitor pattern. The pseudo code for depth-first search using the visitor pattern is included in the description of task 2a and can be more or less directly translated into C++ by using the generic interface (provided by for example the adjacency list from task 1). The main thing to note is that we keep track of the colour of a vertex using a vector internal to the algorithm and thus separate from the vertex itself and by extension the adjacency list. In other words, we do not alter the graph data structure in any way. If the colour a vertex is assigned from running the algorithm is important, then a specialized visitor can be defined that for examples stores this as a property in the vertex.

For task 2b, part one, we are asked to design a depth-first search visitor holding an output iterator, to which we assign each vertex as it is finished by the depth-first search algorithm. As our visitor is allowed to derive from the provided `DFSNullVisitor`, and already does so in the provided initial implementation, we only need to implement the `finishVertex` function and its implementation is trivial. All that remains is then part two of task 2b, which is the implementation of the (reverse) topological sorting algorithm, and all we have to do here is simply call our depth-first search algorithm using the visitor from part 1 of task 2b, where we as a precondition assume that the graph is a directed acyclic graph (DAG).

This concludes our discussion of the design of tasks 1 and 2 of this project. In the next section, we cover some of the details regarding the implementation of tasks 1 and 2.

3 Implementation

In this section, we cover how we have translated the design decisions described in the previous section into the implementation. Starting with task 1 and its introduction, we have opted not to explicitly define copy- and move constructors nor copy- and move assignment operators and left it to the compiler to implicitly generate these.

For task 1a, we have used a `requires` clause together with the `std::derived_from` concept to enforce that the `DirectedCategoryT` type is derived from either `tags::Undirected` or `tags::Directed`. We only need these two, as `tags::Bidirectional` is already derived from `tags::Directed`.

The initial implementation of functions `addVertex` and `addEdge` in task 1b is trivial, but we expand their functionality in later subtasks.

For tasks 1c and 1d, we have implemented `OutEdgeRange` and `InEdgeRange` respectively in the same way as `EdgeRange` is implemented, with the main difference being that we let the `iterator` for `OutEdgeRange` and `InEdgeRange` take an additional argument, which is source and target respectively. Thus in both cases, we use the `boost::iterator_adaptor` to implement the `iterator` and then adjust the `dereference` function to return the appropriate `EdgeDescriptor`. Similarly, the `begin` and `end` functions have been adjusted to refer to the out-edges and in-edges respectively of the vertex `v` given as the argument. Also, we can recall from the design section that because the underlying data structure that the `iterator` refers to is a vector, we can adapt it to be a `std::random_access_iterator` ensuring that we model a `std::bidirectional_iterator` as required.

For task 1d specifically, we also have the requirement that the functionality of `tags::Bidirec-tional` should be available only when that tag (or a tag derived from it) is given. As mentioned in the design section, we have followed the steps outlined in the description and added a variation of `StoredVertexSimple`. Using partial specialization with template parameters `DirectedCatego-ryT1` and `Dummy = void`, we can then choose the appropriate variant with `StoredVertexSimple-<DirectedCategoryT>`. For adding edges, we have simply added two private functions `addEdge-Impl` and then change the public `addEdge` function to use tag dispatching to call the appropriate implementation function based on the `DirectedCategoryT` tag.

In task 1e, we have implemented the functionality for supporting vertex properties by adding an additional template parameter `VertexPropT1` to `StoredVertexSimple` and have then made two additional partial specializations that each have a `VertexPropT1` field; one with only out-edges and one with both out- and in-edges. In addition, for these two variants we have added explicit constructors that take a `VertexPropT1` as an rvalue and store it using `std::move`. We have used the same approach to support edge properties, meaning we have partially specialized `StoredEdgeSimple` using template parameters `EdgePropT1` and `Dummy = void`, also adding a constructor that specifically stores the property of an edge using `std::move`. The appropriate variants for storing a vertex and an edge are then determined as `StoredVertexSimple<DirectedCategoryT, VertexPropT>` and `StoredEdgeSimple<EdgePropT>` respectively. In continuation hereof, for the property variant functions `addVertex(vp, g)` and `addEdge(u, v, ep, g)` we have simply added additional private implementation functions and then use tag dispatching as before to choose the appropriate function for each case. In addition, to ensure that a property is movable, each function `requires` the `std::movable` concept for the property type specified. Accessing the property of a vertex or an edge is simply implemented as a lookup in the appropriate internal vector.

Finally, for task 2, the implementation of the depth-first search algorithm in task 2a is essentially a one-to-one implementation of the pseudo-code given in the task description, where we use `getIndex` function for indexing vertices, the range functions `vertices` to get a range of vertices and `outEdges` to get a range of the out-edges of a vertex. For task 2b part one, as we derive from the `DFSNullVisitor`, we only need to provide an implementation of the `finishVertex` function, which simply needs to dereference the output iterator, be assigned, and then incremented. Task 2b part two, then simply calls our `dfs` function with the `TopoVisitor`.

This concludes the details regarding the implementation of tasks 1 and 2. Next, we briefly cover the tests we have made to verify the desired functionality.

3.1 Tests

To test our implementation, we have made six tests to verify the requirements of the different subtasks.

The first test, `test_init_copy_move` shows that the adjacency list can be default-, copy-, and move constructed, as well as copy assigned and move assigned. For this test, we have configured the adjacency list as `AdjacencyList<graph::tags::Directed, std::string, std::string>`, meaning it also shows that the `MutablePropertyGraph` concept is satisfied, although we have another test showing that also.

The second test, `test_mutable_directed_no_prop` demonstrates that the adjacency list satisfies the `MutableGraph` and `IncidenceGraph` concepts, through the addition of vertices and edges and using out ranges to show that out-edges are available for each vertex.

The third test, `test_mutable_bidirectional_no_prop` demonstrates that the adjacency list satisfies the `MutableGraph` and `BidirectionalGraph` concepts, through the addition of vertices and edges and using both out ranges and in-ranges to show that both out-edges and in-edges respectively are available for each vertex.

The fourth test `test_mutableprop_directed_w_prop` and fifth test `test_mutableprop_bi-directional_w_prop` are identical to the second and third test respectively with the only difference being that vertex and edge properties are used as well.

The sixth and final test `test_topo_sort` demonstrates the implementation of the topological sorting algorithm using the example from Figure 22.7 in Introduction to Algorithms, p. 613.

A makefile has been included in the test directory and running `make` followed by `make test` will first build each test file and then run each test in the order given above.

4 Summary

In this project, starting from a provided initial implementation, we have designed and implemented first a generic configurable adjacency list providing a generic interface for accessing the structure of a graph. We have implemented this using techniques from the course, such as partial template specialization and tag dispatching. We have also taken advantage of the recent addition of concepts, both user defined concepts, such as the provided graph concepts, but also standard library concepts and we have used these to define different behaviours of the adjacency list under different circumstances. In addition, we have also taken advantage of the `boost::iterator_adaptor` library feature, which has enabled a much more simplified and streamlined implementation of the various iterators. Second, we have implemented generic versions of the algorithms depth-first search and topological sorting using the visitor pattern and these have been implemented to take advantage of the generic interface provided by the data structure modelling the graph, such as the implemented adjacency list.

There are several takeaways from this project. First of all, when designing a generic library, ensuring interoperability between the data structures and algorithms is essential. As is the design and layout of the internals, but equally so the interface provided. Not only will this ensure one of the main objectives of reusability, but it will also help make the implementation much simpler. This leads us to our second point, because even though the configurability of the adjacency list requires us to handle several different configuration options, C++ provides clever and relatively simple ways to do this, resulting in an implementation that is much easier to understand. And this is of course a good thing, because the third and final takeaway we have is that although it may be relatively straightforward to learn to use C++, learning to understand it and use it well is an entirely different thing. The depth and versatility of the language is truly impressive and this project has left us with the impression that we have barely scratched the surface.