DM852
Introduction to Generic Programming

Spring 2022

Final Project

This is the final project in the exam of DM852.

Introduction

In the lectures we have discussed the design of a generic graph library, similar to the Boost Graph Library[1], but taking advantage of recent changes in the C++ language. In this project you will be working on extending an initial implementation, both with data structures and algorithms:

1. A generic configurable adjacency list.
2. Depth-first search and topological sorting.

The initial implementation can be downloaded from the example repository and includes

- a codified version of the relevant graph concepts,
- a simple adjacency matrix class,
- a bit of the adjacency list class template,
- simple input/output facilities for graphs, and
- supporting infrastructure for the graph library.

The documentation can be found as comments in the code.

[1] `https://www.boost.org/doc/libs/1_79_0/libs/graph/doc/table_of_contents.html`

Formalities

- All code must be your own, or be provided as part of the course material.
- Your code must be valid C++20 (i.e., compile with `-std=c++20` on GCC).
- To complete the project you need at least GCC version 10, but using GCC 11 (or 12) seems to provide much better error messages in some cases involving unsatisfied constraints.
- The tasks refer to several concepts we have seen in the lectures. For the **`std::`** concepts you can see the lecture slides, and the documentation at `https://en.cppreference.com/w/cpp/header/concepts`. The **`graph::`** concepts are documented in the corresponding file in the provided code.
- You are only allowed to use the C++ standard library and the Boost Iterator library[2], as long as underlying intend of the assignment is followed. You are **not** allowed to directly use any other Boost libraries than the Iterator library.

  If you are in doubt if it is ok to use a particular library feature you should ask.
- You may not work in groups.
- You should test your implementation your self, and all your test files should be put into `exam/test/`.

  Hint for testing: you can use `static_assert` with concepts to check if a certain type fulfils a certain concept.
- Submission is done through the IMADA Git server where you have a dedicated exam repository, called "Repo_exam_⟨*username*⟩", which must be used for submitting. You must place code in the exact files specified, as your submission will be evaluated in part automatically.

  The deadline for submitting is

  Monday, 20 June, at 10:00 CEST

  Whatever is in your exam repository on the `master` branch at the deadline is considered your submission.

[2] `http://www.boost.org/doc/libs/1_79_0/libs/iterator/doc/`

Report

File: `exam/report.pdf`
You must hand in a PDF file with your report of at most 10 pages in total. The report must

- introduce the problem you are solving,
- describe important design choices,
- describe important implementation choices, including application of techniques learned in the course,
- summarize your findings.

Importantly, the evaluation of your submission is based primarily on your report, and only to a little extend on the code.

Task 1: Adjacency List

File: `exam/src/graph/adjacency_list.hpp`
That is, in client code one should be able to write only the include directive `#include <graph/adjacency_list.hpp>` and compile with a suitable `-Ipath` flag, to access your implementation.

Your first task is to implement a configurable adjacency list data structure. You may use the code in the provided file as a starting point.

Your implementation must be a class template `AdjacencyList` in the namespace `graph`. The class template must have 3 parameters:

1. `DirectedCategoryT`, a tag type. See Task 1a–d.
2. `VertexPropT`, which must default to `graph::NoProp`. See Task 1e.
3. `EdgePropT`, which must default to `graph::NoProp`. See Task 1e.

You do not need to support graphs with parallel edges or with loop edges.

Overall requirements for the adjacency list:

- `std::default_initializable`, it can be default constructed,
- `std::copyable`, it can be copy constructed and copy assigned, and
- it has a move constructor and move assignment operator.

Note: copyability and moveability might be implicitly fulfilled by the compiler-generated special member functions, but depending on your design choices you may have to manually implement these methods.

Task 1a: Directed and Undirected Graphs

The first template parameter, `DirectedCategoryT`, for your adjacency list must be a tag type that indicates whether the data structure represents an undirected graph, a directed graph (with access to only out-edges), or a directed graph with access to both out- and in-edges. The type given will in general[3] be respectively either

- `graph::tags::Undirected`,
- `graph::tags::Directed`, or
- `graph::tags::Bidirectional`.

For this project you may however assume that `graph::tags::Undirected` is not used, i.e., your implementation only needs to support directed graphs, but in two variations.

Note: The provided code already supports this requirement.

Task 1b: Vertices, Edges, and Basic Operations

- Implement the requirements for the concepts
  - `graph::VertexListGraph` and
  - `graph::EdgeListGraph`

  but such that `VertexRange::iterator` and `EdgeRange::iterator` model `std::bidirectional_iterator`.
  Note: the provided code already fulfils this requirement.

- Implement a function `getIndex` that for each vertex `v` in a graph `g` returns a unique number in the range $[0;$ `numVertices(g)`$[$. It must be callable as `getIndex(v, g)`, and must return some integer type.
  Note: the provided code already fulfils this requirement.

- Implement the requirements from `MutableGraph` for adding vertices and edges. That is for a graph `g` and vertex descriptors `u`, `v` the following expressions must be valid:
  - `addVertex(g)`: returns a vertex descriptor representing the newly added vertex.
  - `addEdge(u, v, g)`: returns an edge descriptor representing the newly added edge.

  For the `addEdge` function you may require the following pre-condtions:
  - Both `u` and `v` are valid vertex descriptors for `g`.

[3] In general the directed-tag just needs to be derived from one of the three base classes.

  - `u` and `v` are different.
  - No edge $(\mathtt{u}, \mathtt{v})$ exist already in `g`.

Depending on design and implementation choices, you may have to provide custom implementations of copy and move operations for your graph.

Task 1c: Accessing Out-Edges

Implement the requirements for `graph::IncidenceGraph`, but with the requirement that `OutEdgeRange::iterator` models a `std::bidirectional_iterator`.

Task 1d: Bidirectional Graphs

When the directed-tag is `graph::tags::Bidirectional` the user requests that the graph fulfil the requirements for `graph::BidirectionalGraph`. Implement this functionality, but with the requirement that `InEdgeRange::iterator` models a `std::bidirectional_iterator`.

However, when `graph::tags::Directed` is given as directed-tag this functionality is not needed. Using techniques from the course (e.g., tag-dispatching and various meta-programming constructs on types), implement the requirements for `graph::BidirectionalGraph` such that extra memory and computation needed for bidirectionality is only required when that tag is given, and not when only using `graph::tags::Directed` as the directed-tag. This can for example be done in the following steps:

1. Design a variation of the internal vertex storage (`StoredVertexSimple` in the provided code) that holds both out-edges and in-edges.
2. Reimplement the type alias for `StoredVertex` such that the directed-tag is used to decide which vertex storage implementation is used.
3. Implement an alternative `addEdge` function that should be used when the graph is bidirectional.
4. Reimplement `addEdge` to dispatch, at compile-time, to the right function.

Task 1e: Basic Support For Vertex and Edge Properties

It is often useful to store additional data as properties on vertices and edges. Use the two additional template type parameters for your adjacency list, `VertexPropT` and `EdgePropT`, for the user to specify which type of object should be stored for each vertex and edge respectively. Both of these property types must default to `graph::NoProp` which signifies that no extra objects should be stored.

Implement the requirements for `MutablePropertyGraph` and thereby the properties of `PropertyGraph`. However, you may require that `VertexPropT` and `EdgePropT` can be copied, i.e., fulfils `std::copyable`. But it is a bonus if you require only `std::moveable`. If a property type is not copyable, but only moveable, then the whole graph will at most be moveable too.

When `VertexPropT` is different from `graph::NoProp`:

- Each vertex must have an instance of the given type.
- The expression `addVertex(g)` is only required to be valid if `VertexPropT` is default constructible.
- The expression `addVertex(vp, g)` is valid, for `vp` being of type `VertexPropT`. It adds a vertex with the given property.

Similarly, when `EdgePropT` is different from `graph::NoProp`:

- Each edge must have an instance of the given type.
- The expression `addEdge(u, v, g)` is only required to be valid if `EdgePropT` is default constructible.
- The expression `addEdge(u, v, ep, g)` is valid, for `ep` being of type `EdgePropT`s. It adds an edge with the given property.

For reading and modifying vertex/edge properties, implement overloads of `operator[]` such that `g[v]` returns a reference to the label on vertex `v`, and `g[e]` returns a reference to the label on edge `e`. Both `const` and non-`const` overloads must be implemented.

Task 2: Depth-First Search and Topological Sorting

A depth-first algorithm can be used to implement other algorithms, such as topological sorting and strongly connected components. Your task is to implement a generic depth-first search, and then use it to implement topological sorting. For implementing the DFS you should use the visitor-based approach described in the lectures and used in Boost.Graph[4].

Task 2a: Depth-First Search

File: `exam/src/graph/depth_first_search.hpp`
Based on the book Introduction to Algorithms and the Boost Graph library[5], we can describe a visitor-based depth-first search with the following pseudo code.

[4] `http://www.boost.org/doc/libs/master/libs/graph/doc/DFSVisitor.html`

[5] `http://www.boost.org/doc/libs/master/libs/graph/doc/depth_first_search.html`

```
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):
      Visitor.forwardOrCrossEdge((u, v), G)
    Visitor.finishEdge((u, v), G)
  end for
  colour[u] := BLACK
  Visitor.finishVertex(u, G)
```

Using the graph concepts and the **`getIndex`** function, implement the following function template for a depth-first search in namespace **`graph`**.

```
template<typename Graph, typename Visitor>
void dfs(const Graph &g, Visitor visitor);
```

An example of a visitor that does nothing can be found in the provided code (**`DFSNullVisitor`**). You can also use the provided adjacency matrix for testing your implementation.

Task 2b: Topological Sort

File: `exam/src/graph/topological_sort.hpp`
From Introduction to Algorithms we know that a topological sort of a directed acyclic graph can be computed using a depth-first search:

1. Run DFS, and record the finishing time of each vertex.
2. Sort the vertices according to descending finishing time.

However, with our generic DFS algorithm we use visitors instead of recording the finishing times explicitly. In this design we notice that it is easier to construct the reverse of a topological sort. We therefore simply write the reverse order out and leave it up to the user to perform the final reversal.

1. Implement a DFS visitor called
   `template<typename OutputIterator> TopoVisitor`
   that holds an output iterator, and assigns each vertex to it as they are being finished. For implementing the visitor you may derive from the provided `DFSNullVisitor`.

2. Implement the following function template for reverse topological sorting of a DAG. The vertex sequence is written to the given output iterator. You may assume it is only called with a DAG.

   ```
   template<typename Graph, typename OutputIterator>
   void topoSort(const Graph &g, OutputIterator oIter);
   ```

For a graph `g` of type `Graph` we should then be able to compute a real topological sort as:

```
std::vector<typename graph::Traits<Graph>::VertexDescriptor> vs;
vs.reserve(numVertices(g));
graph::topoSort(g, std::back_inserter(vs));
std::reverse(vs.begin(), vs.end());
```