DM852
Introduction to Generic Programming

Andreas Twisttmann Askholm
aaskh20@student.sdu.dk

June 9, 2024

1 Introduction

Graphs are useful structures in many aspects of computer science, they can be used to model many features of the world we live in such as road networks and networks. The use of graphs to model aspects of the real world allows us to analyse and solve complex problems using thoroughly studied algorithms, such as dijkstra's algorithm which finds shortest paths in a graph.

Graphs themselves can be represented in various ways, in this project I have implemented a c++ library for one of those representations, an adjacency list.
This library takes advantage of some of the recent features of the c++ language, to support as many types of graphs as possible in an efficient manner.
To demonstrate a use for the library, an accompanying depth first search and topological sorting algorithm was implemented.

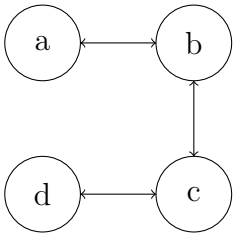
I will describe the design decisions that went into the development of the adjacency list library in the Design Decisions section.
Implementation covers some of the nitty gritty details of the implementation.

2 Design Decisions

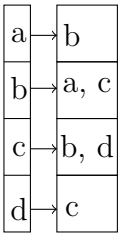
In general a graph is not very useful without any kind of data associated with either vertices or edges. For this reason both vertices and edges are designed as containers to store information, such as distance on an edge and a city name on a vertex for a graph modelling a road network.

2.a Adjacency list

An adjacency list is an efficient and flexible way to represent a graph, hence the reason this representation was chosen.
Efficiency comes both from the need to only consume $\mathcal{O}(V + E)$ space, as



(a) An example of an acyclic simple graph



(b) Traditional adjacency list representation of graph 1a

Figure 1: Illustration showing a graph and its adjacency list representation

---

opposed to an adjacency matrix for example, which uses $\mathcal{O}(V^2)$ space, and the fast traversal of edges connected to a vertex ($\mathcal{O}(degree(V))$ time).
The flexibility comes from its ability to represent any type of graph and its extendability which allows for storage of additional information along with the vertices and edges.

In my implementation I have strayed from the traditional adjacency list with its implicit edges, shown in figure 1b. Instead the implemented adjacency list resembles the one shown figure 2, this approach allowed me to store data on each edge without duplicating the stored data.
This approach, of having explicit edges, meant each vertex should reference a number of edges to retain the relationships modelled in a traditional adjacency list, shown by the arrows in the above illustration.

To further reduce the amount of storage used to store an adjacency list, I decided to store the references to edges only once for undirected and directed graphs, this approach halved the references stored with the vertices.
The edges are still available in the edge list and each vertex in a directed graph will then reference its outgoing edges. To check if two vertices are connected in an undirected graph, this choice necessitates a lookup in the edge list.

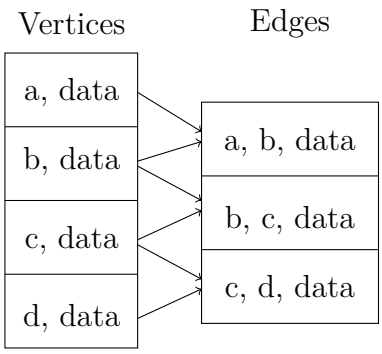


Figure 2: Illustration showing an adjacency list representation of graph 1a, in my implementation.

The reference to an edge held by a vertex, consists of an index in the edge list and the target of the edge (the source for ingoing edges). Having access to source, target and index, makes it easy to construct an edge descriptor without having to access the edge itself.

Descriptors

Throughout the library both edges and vertices are represented by so called descriptors in addition to the actual stored entities. The use of descriptors allowed me to pass around detailed references to the stored entities without copying or moving the potentially large amount of data stored on those.

An edge descriptor is essentially comprised of three indices, an index in the vertex list for each endpoint and an index in the list of edges, where the actual edge and its data is stored.
Likewise a vertex descriptor is essentially an index in the list of vertices. Just as with the edges the actual data associated to a vertex, is contained in this list and can be accessed using the descriptor.

2.b DFS topological sort

To test and demonstrate the capabilities of the adjacency list library both a depth first search and a topological sorting algorithm was implemented. The depth first search uses a visitor approach, as described in the boost library documentation[1].
The search is essentially a recursive depth first exploration of all vertices, which marks the vertices depending on the status of the visit (not seen, exploring, finished).

The visitor approach is a way to separate an algorithm from the traversal of the underlying structure. This approach makes the implemented depth first search reusable for other applications[2] in addition to topological sorting.
The topological sort itself consists of recording when a vertex is being marked 'finished' in a depth first search, and ordering the vertices accordingly. I decided to order the vertices in ascending order of finishing time as opposed to the traditionally descending order.

3 Implementation

To achieve the support for both directed and bidirectional graphs, without including unnecessary methods and while keeping the interface consistent, i.e. calling `addEdge(u,v,g)` should add an edge to the graph as long as it is mutable, regardless of the directionality of the edges.
The following expression is evaluated at compile time, to adopt the compiled code to the type of graph for which it is used.

```
1 constexpr static bool isBi =
  ↪  std::derived_from<DirectedCategoryT,
  ↪  graph::tags::Bidirectional>;
```

[1] https://www.boost.org/doc/libs/master/libs/graph/doc/depth_first_search.html

[2] https://en.wikipedia.org/wiki/Depth-first_search#Applications

Based on this check the type of vertex is decided, and the appropriate versions of `addEdge(u,v,g)` is included in the compiled library.

In total six different versions of `addEdge(u,v,g)` has been implemented. Three versions to accommodate different edge properties, each duplicated to support the different directionalities.
The different templated versions was constrained using the keyword `requires`, along with functions such as `std::copyable<EdgePropT>` and `std::is_default_constructible<EdgeProp>::value`, to check the properties of the data stored in the edges.
If a constraint on a method is not fulfilled, it will not be available in the compiled version of the library.
`addVertex(g)`, which adds a vertex to a graph, is also implemented three times, each version being constrained in the same manner as `addEdge(u,v,g)`.

Iterators

Iterators for the different edge lists makes it easy to iterate through the edges, for instance:

```
1 for (auto e : outEdges(u,g))
2     std::cout << e.tar << std::endl;
```

goes through all outgoing edges of vertex `u` in graph `g`, and prints the index of the vertex at the other end.
In order to implement iterators I relied on the boost iterator library, in particular the `boost::iterator_adaptor`. This approach allowed me to specialize the core functionality of the boost iterator to my needs.
To adapt the underlying iterator I created a new dereference method for the iterators, since the dereferencing of an iterator should return an edge descriptor instead of the actual elements being iterated over.
In addition to the dereference method, the constructors should be updated

to construct iterators on edgelists.
The specialisation was done for the list storing the actual edges, as well as the outedge and inedge lists contained in the vertices.
The iterators was used to construct ranges (from begin to end), in order to achieve the functionality illustrated in the code snippet above.

DFS

Both the DFS and toposort implementations are constrained to the types of graphs that fulfil the concept of Godfs. This constraint stems from the expectation that certain methods are available when running the DFS.

Concepts in c++ is a feature that allows templated functions to be constrained to template parameters satisfying the concept, in this case the concept of Godfs.
The concept of Godfs itself is a concept that requires a graph to fulfil multiple other concepts. These other concepts require the underlying graph library to provide the functions used throughout the dfs algorithm, such as `edges(g)` and `outEdges(v,g)` .

Visitor

The visitor used for topological sorting derives from a so called nullvisitor. The nullvisitor has all the functions required by the DFS implementation, but provides no actual functionality. Building on this visitor allows me to only implement the functions needed to perform the sorting using dfs, notably the `finishVertex(v,g)` function.

4 Conclusion

The use of concepts and requirements provides better and more meaningful error messages. Combined with derivations these features enables the development of complex algorithms in a concise and easily readable manner.
While the multiple implementations of some of the methods does require some extra work, it results in a much more versatile library. The constraining of methods, results in a compiled version which only includes the functionalities required for its current use.