Programming Assignment 5

Topological Sort

Objectives

In this assignment, you will develop a graph data structure built on top of an adjacency list representation. Furthermore, you will implement an algorithm for determining a topological ordering of a graph.

Report & Turn In

There is no report for this assignment. However, you will likely find some questions on HW3 easier after completing this assignment.

`!IMPORTANT` By submitting code to Mimir and/or Canvas you acknowledge and are bound by the Aggie Honor Code:

> *On my honor, as an Aggie, I have neither given nor received unauthorized aid on this academic work*
>
> *An Aggie does not lie, cheat, or steal, or tolerate those who do.*

Your submission will be taken in place of a digital/physical signature.

Turn in your code to Mimir. You should submit the following files:

- `graph.h`

Please also submit a zipped or tarball version of these files to Canvas.

Provided Materials

Starter code is provided on Github: `https://github.tamu.edu/csce221/pa5`

You probably only need to modify `graph.h` .

A sample main, data, and makefile are provided for testing your code locally.

Graph Data Structure

In this part you should implement a `Graph` data structure which is defined based on an additional, templated type `Vertex` . The implementation of the Graph class should be based on **adjacency lists**, see the starter code in `graph.h` .

Inside the `Vertex` class you probably do not need to do anything. Notice, *Vertex* does not contain a default constructor. You should not need one, but may define one if you deem it necessary. Vertex also does not contain an `operator<<` . Again, you probably do not need one, but if you deem it necessary feel free to implement it.

Inside the `Graph` class:

- `int size(void)` - return the number of Verticies in the graph.
- `void buildGraph(std::istream&)` - take a graph from the provided stream and build the corresponding graph data structure. The format is discussed below.
- `void displayGraph(std::ostream&)` - display the graph to the stream. The format is discussed below.

- `Vertex<T> at (T)` - return the Vertex whose label matches (`==`) the given `T` . Throw an exception if no such Vertex exists.
- `void compute_indegree(void)` - Compute and assign the indegree to each `Vertex` in the `Graph`
- `bool topological_sort(void)` - Compute and assign the topological number to each `Vertex` in the `Graph` . Return `true` if successful and `false` on failures (i.e. a cycle exists in the graph). The topological ordering should be increasing such that a node that comes first in the ordering will have a lower value than nodes that come later.
- `void print_top_sort(ostream&, bool)` - print the nodes in topological ordering into the provided `ostream` . Notice this function is partially implemented with the optional trailing newline logic. Please keep this logic as it is may impact your ability to pass test cases.

> *You may assume that the graph will not change after it is constructed by* `buildGraph`

Design Comments

In building your graph, you are encouraged to utilize standard library data structures. However, an efficient implementation is is critical to passing the test cases. Be careful in your operations; accidental copying of large data structures is a common error and will make it difficult to pass Mimir test cases (within the provided time constraints).

Also be sure you understand the underlying implementation of the data structures. `std::map` is an ordered map, usually based on *red-black* or *AVL* trees. By contrast `std::unordered_map` is usually based on *hash tables* and has much better performance.

Graph input format

The input stream will be line (`\n`) delimited. The first token on each line will be the vertex label (**L**) and the remaining tokes will be the vertices which contain an edge from **L** to these vertices. See the example below for more info.

A word of warning, a Vertex label may first appear as a starting label (**L**) or as a receiving label. For example, labels `2` `4` and `5` appear first in the adjacency list of `1` in the example below. Be able to handle both cases.

While you should be ok to assume that tokens will be space delimited, it will be better practice to use `operator>>(std:istream&, T)` to allow the templated class to define how tokens are processed. One common design pattern is to:

- split the input stream into lines
- pass a single line into a `std::stringstream` object
- repeatedly use `operator>>` to extract single tokens from the `std::stringstream` , until the line is completely read (`eof` is set on the `std::stringstream` object)

The graph is assumed to be sparse and unweighted. Do not attempt to linearly store all possible labels. There will be holes in the domain of input labels: ex (`1` , `10` , `100` , `1000`).

> ***Introduction to Combinatorics****: The domain of possible values quickly becomes extremely untenable: a seven character string of only* `A-Za-z0-9` *characters, such as* `CSCE221` *, has a space of* 62^7*, or about* 2^{41}*. That is approximately 2 Trillion labels, times however many bytes is needed per label, which is almost certainly more than* 1*. Your computer probably does not have the Multi-Terrabytes of RAM required for such an implementation; Mimir certainly does not.*

Graph output format

The output should be one line per node in the format: `label:adjacency_list`

Mimir is configured to accept the adjacency list in any order. Likewise the lines can be printed in any order.

An Example

Consider the graph:

```
  1 ----→ 2
 /  \   / | \
 |   | |  |  |
 ↓   ↓ ↓  |  ↓
 5    4 ←-+- 3
 ↑   / \  |
 |  |   | |
 \  ↓   ↓ ↓
   6 <-- 7
```

One valid input for the graph is:

```
1 2 4 5
2 3 4 7
3 4
4 6 7
5
6 5
7 6
```

One valid output (from `operator<<`):

```
6:5
2:4 7 3
1:2 4 5
4:7 6
7:6
3:4
5:
```

> *Note: The input and output representations are not unique for a graph. You should be able to handle permutations of the input. Mimir will accept any valid permutation of the output.*

The valid topological ordering for this graph:

```
1 2 3 4 7 6 5
```

> *Note: Most graphs will have multiple valid topological orderings. Mimir will accept any valid ordering*

Topological Sort

A topological sort is an ordering imposed on a directed graph. It ordered vertices such that every a vertex is visited before any of the vertexes to which it has an edge. Topological sorting is a common and powerful tool for organizing data represented in a graph.

For example, course scheduling can be modeled as a topological sort: students must take `CSCE121` before `CSCE221` . Students must take `CSCE221` and `CSCE222` before `CSCE313` . However students can take

`CSCE222` and `CSCE121` in either order or concurrently. The problem of organizing these classes is one of topological ordering.

Some other common uses are for compilation dependency (determining what needs to be recompiled if a file changes) and also for determining dependencies between test cases (is something wrong or did another failure cause this failure)

Psuedocode

Psuedocode from the textbook, section 9.2 pg 382-385

For calculating the indegree of the node:

```
for vertex in graph:
    vertex.indegree = 0

for vertex in graph:
    for adj_vertex in vertex.adj_list:
        adj_vertex.indegree += 1
```

For performing the topological sort

```
q = Queue()
counter = 0

q.makeEmpty()
q.extend([v in graph if v.indegree == 0])

while not q.isEmpty():
    v = q.pop()
    v.top_num = counter
    counter += 1

    for adj_v in v.adj_list:
        adj_v.indegree -= 1
        if adj_v.indegree == 0:
            q.push(adj_v)

if counter != graph.size():
    raise CycleDetectedException()
```

Templates inside templates (>>)

When templating templates (ex: `std::vector<Vertex<T>>`) you frequently end up with syntax `>>`. On *very* old compilers, this was sometimes confused with `operator>>`, which resulted in compilation errors or unexpected behavior. While modern compilers are much better at disambiguating (or at least erroring out when it is ambiguous), you may see warnings about this behavior. For this reason, most nested-templates are written with a space between closures (ex: `std::vector<Vertex<T> >`).

C++ Standard Library

Consider the following C++ standard library containers that are of note:

- `std::unordered_set`
- `std::unordered_map`
- `std::set`

- `std::map`

The former two use a hash table, the latter two use red-black trees. The key difference is `set` elements are immutable whereas `map` elements are mutable. The other key difference is that the ordered data structures require the elements to have an ordering (`operator<` is defined). This allows the in-order traversal of nodes based on this ordering. This would be great for `print_top_sort()` , however, as the topological ordering is not known at insertion time, this cannot be used for ordering until after the graph is read in; `std::unordered_map` is the preferable data structure.

A word of warning; many students struggle with incidental copying of elements. Be sure you understand if an operation is creating a copy and if you actually *want* to create a copy at that point. Copies are expensive in terms of runtime but also in terms of correctness. Modifying a copy does not modify the original elements; you may think you are updating things in the map, but in reality you are modifying the copy. This is a **very common mistake**. If you are running into problems, study the code below to understand the differences.

To aid in working with the standard library, the following code is provided:

```
// create the unordered_map object
// the two template types are for key and value type
//  notice the > > at the end of the template
std::unordered_map<T, Vertex <T> > node_set;

// create and insert a new object with key `label`
// if a key in the table with this item exists,
//      the new object is not inserted
// returns a std::pair<std::unordered_map <T, Vertex <T> >::iterator, bool>
//      where iterator is a reference to the object with `key` label in the hash table,
//      bool is true if the new element was inserted,
//          false if this key was already present
auto pairOut = node_set.insert(make_pair(label, Vertex<T>{label , 0}));
bool newItem = pairOut.second; // true if this is the first item with the given key

// use `auto` save keystrokes
unordered_map<T, Vertex<T> >::iterator iter = pairOut.first;

// the iterator can be dereferenced to get the object back
Vertex<T> v = *iter; // create a copy of the v object
Vertex<T>& v = *iter; // create a reference to v in the map
// WARNING : references (and some iterators) are transient (temporarily valid)
//   generally, only valid until the next insert is made,
//      but may vary with different data structures
// - pointers to references should never be made
// - be careful storing references in variables

// Working with STL data structures require considering when references and copies are used

// The trivial pattern is:
Vertex<T> v = node_set.at(label); // copy assignment for v
v.top_num = 0;                    // changes to v
node_set.at(label) = v;           // copy assignment modified object back into node_set

// Alternatively, using references can save some copies
//  top_num is 0 by default
cout << node_set.at(label).top_num << endl;  // outputs 0
Vertex<T>& vRef = node_set.at(label);        // by reference
```

```cpp
vRef.top_num += 1;                             // incrementing the object in the map
cout << node_set.at(label).top_num << endl;  // outputs 1
Vertex<T> vCopy = node_set.at(label);         // copy assignment
vCopy.top_num += 1;                            // increments the copy
cout << node_set.at(label).top_num << endl;  // outputs 1 again
node_set.at(label).top_num += 1;              // incrementing the object in the map
cout << node_set.at(label).top_num << endl;  // outputs 2


// much of the same applies to iterating the map object
//  in both the cases auto is pair<T, Vertex<T> > type object
//  in case this is a new syntax for you, these are for-each loops
//  they iterate over all objects in the map

// elements by reference, updates within the map
for(auto &v : node_set){
    v.second.indegree = 0;
}
// elements by copy, no updates to the item in the map
//  each step requires copying a vertex, which may be very slow
//      depending on the size of the adjacency list
for(auto v: node_set){
    v.second.indegree = 0;
}
```

Another standard library data structure you may be using for the first time is `std::priority_queue` , implemented via a binary heap. It is similar to the one you implemented in `PA4` except this one takes either one or three template parameters: `<Type, ContainerType, Functor>` . The `std::priority_queue` orders maximizing, meaning that the greatest priority element is returned first. Either the stored *Type* or the *Functor* should implement the `operator<()` . The *Container* type is unimportant, `std::vector<Vertex<T> >` can be used. A *Functor* is an object (so it has a class) that implements `operator()` . Basically, this is an object that can be called as a function. It is also an object, so it can have any of the other constructors, operators, inheritance, polymorphism, ect., that you have seen throughout the semester. The code for declaring a *Functor* class is show below. Recall, that the implementation of topological ordering assigns the first ordered elements a lower value and `std::priority_queue` is maximizing.

> *Note:* `Functor` *is sometime also referred to a* `Comparator` *. Neither of these words are a formal part of the C++ specification, so their definitions are semi-ambiguous.*

```cpp
// syntax for a custom comparator
template <class T>
class VertexCompare {
public:
    // will be called as operator<()
    bool operator ()(Vertex <T> v1 , Vertex<T> v2){
        // TODO - implement
        return false;
    }
};

// ...
```

```
std::priority_queue<Vertex<T>, vector<Vertex<T> >, VertexCompare<T> > pq;
```