Assignment 1: Graphs

ECEN 625 – Winter 2021

Due Date: Tuesday, January 15, 2019 11:59pm

1 Learning Outcomes

The goals of this assignment are to:

- Familiarize yourself with graph algorithms that are most applicable to HLS techniques, namely topological sorting and identifying the critical path.
- Observe the structure of commercial HLS data flow graphs.
- Practice C++ skills.

2 Preliminary

This assignent is located in the `asst_graphs` folder. All commands shown assume you are located in that folder in your terminal.

2.1 Install Packages

2.2 Extract Graphs

Due to size contraints on Github, the graphs are in a zip files and need to be extracted. Run the following to extract the graphs:

```
cd lab_graphs
make unzip_graphs
```

2.3 Inspect the code

Inspect the contents:

- `niGraphs/` – This folder contains 2333 graphs of customer designs from National Instrument's HLS tool (LabVIEW Communications System Design Suite – `http://www.ni.com/labview-communications/`). Feel free to look at these files. You will see each file defines a set of nodes and edges with associated properties. As we progress through the class material you will learn more about these properties, but you can ignore most of them for now, aside from a few that are specifically mentioned in this assignment.
- `src/niGraphReader/` – This contains the NIGraphReader class, which will parse the graph files into NIGraph* data structures.
- `src/niGraph/` – This contains the NIGraph class, as well as NIGraphNode and NIGraphEdge classes, which are the data structures for the graphs.
- `src/main.cpp` – You will need to implement several functions in this file. You are free to split your code into additional files if you desire.

2.4 Build the code

The project uses `cmake` as a build manager. `cmake` is a tool that will automatically create necessary makefiles for use with the `make` tool. The code can be built using the following commands:

```
cd build
cmake ../src
```

```
make
```

This will produce an executable located in If you change contents of the `.cpp/.h` files, you will need to repeat from step 3. If you add new files, you will need to repeat from step 2. If you want to clean all temporary files, simply delete the `build` directory and start over from step 1.

If you are interested in how `cmake` works, you can look at the `CMakeLists.txt` files in `src/` and its subdirectories.

3 Requirements

4 Deliverables

4.1 Part 1: Visualizing Graphs

For this deliverable you must write code to output an `NIGraph` structure in DOT language. The code should be added to the following function:

```
void createDOT(NIGraph * graph, string outputPath) {
  // add code here
}
```

The graph should have the following properties:

- Show all nodes and edges.
- Nodes should be labeled with the node `id`.
- Edges should be labeled with the edge `delay`.
- Feedback edges should be colored in red (vs default of black).

See `http://www.graphviz.org/content/dot-language` for the specification of the DOT language. For example, a simple DAG with two nodes (`a` and `b`) and one edge (`delay` = 3) may have a DOT file like this:

```
strict digraph {
  a -> b [label="3"];
}
```

When you are building the DOT file, you can assume every node in the graph will have at least one connected edge. So to build the DOT file you only need to iterate through the edge list, not the node list.

You can use the `dot` program to create a graphic from a DOT file. For example, the following command will create a PDF file (many different file types are supported, see `http://www.graphviz.org/doc/info/output.html`):

```
dot graph0.dot -Tpdf -o graph0.pdf
```

You may need to install the graphviz package before using the `dot` command:

```
sudo apt install graphviz
```

Figure 1 provides an example of the visualized graph, for DelayGraph_0.

Deliverables: Choose any two graphs you like (aside from DelayGraph_0) and include an image such as Figure 1 in your report. Make sure the graph isn't too large, or it won't be readable. Anything much larger than the provided example is probably too large.

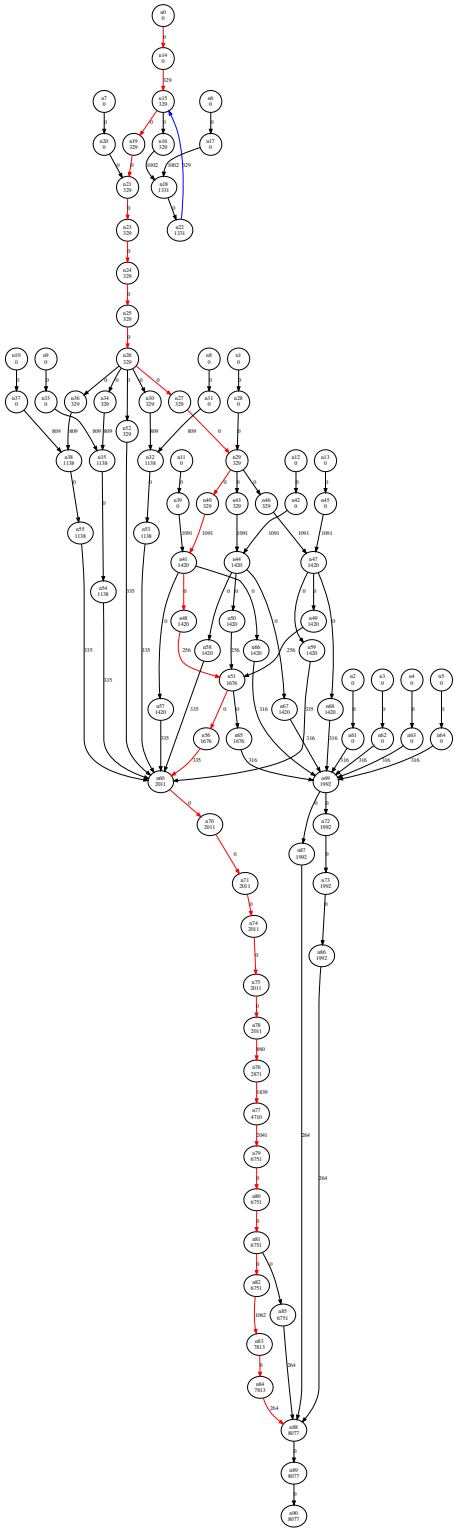

Figure 1: DelayGraph_0

4.2 Part 2: Topological Sorting

In this part of the assignment you will write code to perform a topological sort of a graph. See lecture slides, `https://en.wikipedia.org/wiki/Topological_sorting`, or search elsewhere online, for topological sorting algorithms.

The code should be added here:

```
deque<NIGraphNode*> topologicalSort(NIGraph * graph) {
  // add code here
}
```

The function has a single input, an `NIGraph`, and returns a topologically sorted list of nodes (`NIGraphNode*`). Topological order is such that for every directed edge uv from node u to node v, u comes before v in the ordering. Since a topological sort is only possible for directed acyclic graphs (DAGs), you will need to ignore the feedback edges.

Deliverables: Write a short paragraph about how your topological sorting algorithm works.

Include a scatter plot (`https://en.wikipedia.org/wiki/Scatter_plot`), which plots the run-time for your topological sort code for **ALL** 2333 of the provided graphs. The plot should be of the following format:

- The x-axis should show the size of the graph ($V + E$)
- The y-axis should show the runtime of the topological sorting.
- Both the x and y axis should be in logarithmic scale, with appropriate ranges to fit your data points.

There are many ways to do a topological sort. For full marks, your chart data should show that your algorithm complexity is approximately $O(V + E)$. Please don't spend extra time performing analysis to show this – just a visual inspection of the scatter plot is fine.

4.3 Part 3: Longest Path

In this deliverable you will write code to detect the longest delay path in the graph, using the topologically sorted nodes from Part 2. This code should be written in this function:

```
int longestDelayPath(deque<NIGraphNode*> & topolSortedNodes,
     deque<NIGraphNode*> & longestPath)
  // add code here
}
```

This function has two parameters. The first accepts a list of nodes sorted in topological order, and the second is a list which is populated with the nodes on the longest path. The return value of the function is the total delay along the longest delay path.

See lecture slides, `https://en.wikipedia.org/wiki/Longest_path_problem`, or search on the web for how to determine the longest path from a topological sort. For a DAG, the longest delay path is also known as the *critical path*. This term is likely familiar to you in the circuit domain, as combinational logic can be represented using a DAG, and the critical path restricts the maximum frequency of the circuit.

Again, to ensure the graph is a DAG, you will need to ignore feedback edges.

Deliverables: Include the following data using your longest path code:

- Include the following table in your report, with all values populated:
- Include the longest path for DelayGraph_3. A function is provided to print the list:

```
  void printNodeList(deque<NIGraphNode*> nodes);
```

Graph	**size** $(V+E)$	Delay
DelayGraph_0	197	8077
DelayGraph_1		
DelayGraph_2		
DelayGraph_3		
DelayGraph_4		
DelayGraph_5		
DelayGraph_6		
DelayGraph_7		
DelayGraph_8		
DelayGraph_9		
DelayGraph_10		

For example, the longest path for DelayGraph_0 is:

```
n0 -> n14 -> n15 -> n19 -> n21 -> n23 -> n24 -> n25 -> n26 -> n27 ->
n29 -> n43 -> n44 -> n50 -> n51 -> n56 -> n60 -> n70 -> n71 -> n74 ->
n75 -> n78 -> n76 -> n77 -> n79 -> n80 -> n81 -> n82 -> n83 -> n84 ->
n88
```

5 Coding Guidelines

- Your code should be added to the `src/522r_asst1.cpp` file. You can add additional files if you like, just don't forget to submit them. You shouldn't change the `niGraph*` files.
- You are free to add extra helper functions, but you should not change the definition of the provided functions.
- I have made `topologicalSort()` return a `std::deque` instead of a `std::vector`, as it supports $O(1)$ insertion at the front or back of the list (`std::vector` is $O(n)$ insertion at the front). Since different topological sorting algorithms require inserting at the front or back, I wanted to provide this functionality.
- It doesn't matter what code you submit in `main()`, you will only be marked on the code in the required functions, and any code that those functions call.

6 Submission Instructions

1. Submit a report containing the following items:
   - Your name :)
   - The two DOT graph images, described in Section 4.1.
   - The paragraph describing your algorithm, and scatter plot, from Section 4.2.
   - The table, and longest path, described in Section 4.3.
   - Feedback about the assignment
     - How many hours you spent on the assignment?
     - How challenging was the C++ coding?
     - Anything you liked?
     - Anything you didn't like? Or anything you would change?
     - Did you find the assignment worthwhile? Why or why not?

2. Submit your source code. If you only change the one .cpp file, then only submit that file.

Send your report and code to jgoeders@byu.edu with the subject: 625 Asst1

7 Evaluation Criteria

The three deliverables will be weighted equally. You will be marked based on completion, adherence to specification, correctness of code, efficiency of code, and readability of code.