

# PA5 Report: Directed Acyclic Graph and Topological Sort

Minh Luu

CSCE 221-509

Date: April 27th, 2020

## Objectives:

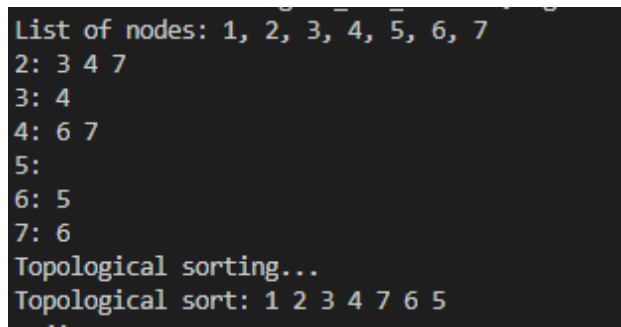
In this assignment, I am introduced to another data structure called graph. Specifically, I will use a specific type of graph called directed cyclic graph which is a graph that has direction and there is no cycle within it. I will implement the data structure in C++ with a vector of vertices and represent it with an adjacency list. Furthermore, I will extend the data structure with topological sort which is used in various applications such as student scheduling and software packing. By the end of this assignment, I will have a basic understanding of graph data structure and topological sort and hopefully use them on building software in the future.

## Procedure:

This programming assignment is divided into two parts: building the graph data structure and building topological sort.

Our graph can be implemented with two classes: Vertex for the vertices with label, indegree and topnum as its data members, and Graph for, well, building the graph. Graph contains a vector for storing Vertex objects and another vector of lists to represent the graph as an adjacency list. I followed the instruction and tested the structure using the given .data files in addition to debugging tools such as gdb and Valgrind.

The topological sort is built using the given example in the textbook with little to modifications needed to make sure it works well with the given graph data structure (the example is pseudocode, so I had to translate it to C++ code). Four cases are introduced to test the sorting algorithm and the results were verified by a TA.



```
List of nodes: 1, 2, 3, 4, 5, 6, 7
2: 3 4 7
3: 4
4: 6 7
5:
6: 5
7: 6
Topological sorting...
Topological sort: 1 2 3 4 7 6 5
```

Fig 1: My graph with adjacency list representation for input.data.

My program assume that the inputs are integers and that they are in the specific format given by the assignment. Although topological sort requires the input graph to have no cycle, my program would throw an exception if a cycle is detected instead of getting an unexpected result.

```
==41==  
==41== HEAP SUMMARY:  
==41==    in use at exit: 0 bytes in 0 blocks  
==41== total heap usage: 40 allocs, 40 frees, 87,104 bytes allocated  
==41==  
==41== All heap blocks were freed -- no leaks are possible  
==41==  
==41== For counts of detected and suppressed errors, rerun with: -v  
==41== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Fig 2: The program being tested with valgrind to prevent memory leaks.

## Results and Discussion:

Below are some more tests for the topological sorts.

```
List of nodes: 1, 2, 3, 4, 5, 6, 7, 8  
1: 3  
2: 3 8  
3: 4 5 6 8  
4: 7  
5: 7  
6:  
7:  
8:  
Topological sorting...  
Topological sort: 1 2 3 4 5 6 8 7
```

Fig 3: Output for input2.data

```
List of nodes: 1, 2, 3, 4, 5, 6, 7, 8, 9  
1:  
2: 1  
3:  
4: 1  
5: 2 4  
6: 3 5  
7: 1  
8: 3 1  
9: 5  
Topological sorting...  
Topological sort: 6 7 8 9 3 5 2 4 1
```

Fig 4: Output for input3.data

```

List of nodes: 1, 2, 3, 4, 5, 6, 7
1: 2 4 5
2: 3 4 7
3: 4
4: 6 7
5: 4
6: 5
7: 6
Topological sorting...
terminate called after throwing an instance of 'std::invalid_argument'
  what(): Cycle found!
Aborted (core dumped)

```

Fig 5: Output for input-cycle.data

The algorithm uses queue to put any vertex with its indegree value of zero into the queue. For each vertex getting dequeued, its adjacent vertices' indegree get decreased by one. A vertex is put on the queue as soon as its indegree falls to 0. The algorithm keeps going until the queue is empty. A stack can be used for this algorithm as well since the order does not matter, if it is put in the stack, it will get processed and the algorithm will keep putting in and processing vertices until the stack is completely empty. Additionally, the algorithm detects cycles because it would not work if there is one, since for two vertices  $v$  and  $w$  on the cycle,  $v$  precedes  $w$  and vice versa, therefore, there is no wrong order in that case.

The time it takes to build the graph is  $O(E + V)$  where  $V$  is the number of vertices and  $E$  is the number of edges, because for each vertex, I have to input in all of its edges as well (no repeated edge because there is no cycle). For topological sort, according to the book, it takes  $O(E + V)$ . This makes sense because the loop is executed once per edge and it will iterate through all vertices. Since this is just a very simple list, I sorted my nodes in topological order using bubble sort which is  $O(V)$ .

## Conclusion:

In conclusion, I have successfully implemented a graph and topological sort. Through the implementation, I can see how a graph can be modeled with an adjacency. Additionally, I was able to learn about topological sort and how it can be used many useful applications in real life.