Comparison of Topological Sort Implementations

Alex Shank

ECE60800 – Computational Models and Methods

November 17, 2022

Table of Contents

1. Introduction

A topological sort of graph G with vertex set V and edge set E is an ordered list $v_1, v_2, \ldots, v_{|V|}$ such that $v_i \in V$ and E does not contain any edge $e = (v_j, v_i)$ $1 \leq i \leq j \leq |V|$. Intuitively, a topological sort lists a graph's vertices in order of how many dependencies each vertex has (least to most dependencies). Topological sort is often used to identify which tasks (vertices) must be completed before other dependent tasks may start.

The following pages describe implementing two topological sort algorithms in Java and compare the implementations' execution times on various graphs. The programming environment and library dependencies of the implementations are explained first. Topological sort's requirement of Directed Acyclic Graphs is then explained in the graph biasing section along with the methodology for creating test graphs. The Depth First Search (DFS) implementation of topological sort is then discussed as the first implementation of topological sort. Afterwards, Khan's Algorithm is discussed as the second implementation of topological sort. Finally, the two implementations are compared by collating tabular results and observing execution time trends.

2. Implementation Language and Dependencies

The two topological sort implementations are written in Java 16 and leverage the popular graph library JGraphT. Java was chosen for its rich development environment; it has standard libraries for timing execution duration, collating test data, and exporting test data to files. JGraphT has abstract Graph, Vertex, and Edge classes that can expedite developing graph-related programs. These fundamental classes eliminate boilerplate code and allow developers to implement graph algorithms rather than the fundamental operations of a graph data structure. Under the hood, JGraphT represents graphs with one HashSet for vertices and one HashSet for edges (as opposed to with an adjacency matrix or adjacency lists).

After running tests and exporting data to CSV files, a Python script leveraging the Pandas data science library analyzes the data. Pandas has a streamlined API useful for calculating averages across the different algorithms, test runs, and graph sizes. Finally, Excel is used to create the table in Section 6 and compute the linear fit error of the two implementations.

3. Test Methodology and Graph Biases

As discussed in sections 4 and 5, the asymptotic runtime of both topological sorts is expected to be $O(|V| + |E|)$. This informs the choice of vertices and edges in each test case. Allowing $N = |V| + |E|$, the two implementations can arbitrarily be run with:

$$N = 10{,}000;\ 100{,}000;\ 500{,}000;\ 1{,}000{,}000;\ 5{,}000{,}000$$

Execution time is then dependent on only the relationship between $|V|$ and $|E|$, or how sparse the edges of the test graphs are. This relationship between $|V|$ and $|E|$ will be referred to as "edge sparsity" hereafter. Five cases of edge sparsity are tested at each N:

(1) $$|E| = \frac{|V|}{10}$$

(2) $$|E| = \frac{3 * |V|}{4}$$

(3) $$|E| = |V|$$

(4) $$|E| = \frac{1}{4} * \frac{|V| * (|V| - 1)}{2}$$

(5) $$|E| = \frac{1}{2} * \frac{|V| * (|V| - 1)}{2}$$

The max number of edges in each graph is limited to $|E| = \frac{|V|*(|V|-1)}{2}$, because topological sort is only defined on DAGs. Graphs with high edge density are therefore generated with $|E|$ close to this DAG edge limit. Graphs are also biased to be acyclic by excluding edges $e = (v_j, v_i)\ 1 \leq i \leq j \leq |V|$ as mentioned in Section 1.

A different test is run for every combination of N, edge sparsity, and topological sort implementation. Each test is run 10 times and each of the 10 runs has a newly generated random DAG. The execution time of each run is computed by calling Java's **System.nanoTime()** function and converting its result to milliseconds. The average execution time for the set of 10 runs is later computed with Pandas' **mean()** function.

4. DFS Implementation

The DFS-based topological sort implementation adds vertices to the sorted list based on their DFS "finishing" times. The given graph's vertex set is iterated over until each node is visited. A node may be visited by the initial iteration over all vertices, or a node may be visited recursively as the successor of another node.

A set with all visited nodes is maintained while DFS recursively visits the successors of each visited node. Once all successors of a node have been visited, the node itself "finishes" and is added to the output array. Once the outer loop over all vertices has completed, the sorted output is returned.

The asymptotic runtime of DFS-based topological sort is $O(|V| + |E|)$. Each node is visited once, which contributes $|V|$. Each of the parent node's successors is then visited through the parent node's edges, which contributes $|E|$.

5. Khan's Algorithm Implementation

The Khan topological sort implementation identifies vertices with no incoming edges and adds them to the sorted output. If a vertex has in-degree zero, then it cannot be dependent on other vertices and should therefore be earlier in the sorting.

Once a vertex with zero in-degree is added to the sorted list, its successor vertices have their in-degree decremented (essentially removing the parent node from the graph). This produces new vertices with in-degree zero. The entire process repeats until all vertices have in-degree zero and have been added to the sorted output.

The asymptotic runtime of Khan topological sort is $O(|V| + |E|)$. Each node is visited once to calculate its in-degree, which contributes $|V|$. Each of the parent node's successors is then decremented through the parent node's edges, which contributes $|E|$.

6. Comparison and Conclusions

Table 1 below shows a tabular comparison of the DFS-based topological sort and the Khan topological sort. The table breaks down each test case by $N = |V| + |E|$ and edge sparsity. The leftmost "DFS Avg" and "Khan Avg" columns are measured in milliseconds and are computed from the 10 test executions at each combination of N and edge sparsity. The difference between these two averages is then computed for the leftmost "DFS Faster?" column. This column is highlighted green if the DFS-based topological sort is faster than the Khan topological sort.

Similarly, the rightmost "DFS Avg" and "Khan Avg" columns aggregate all test cases at a given N and are measured in milliseconds. The rightmost "DFS Faster?" column is highlighted green if the DFS-based topological sort is faster than the Khan topological sort.

Finally, the rightmost "DFS Avg" and "Khan Avg" columns are linearly fit against $N = |V| + |E|$, since topological sorting is expected to be $O(|V| + |E|)$. The R^2 value of each fit is included at the bottom of these column and indicates how well the test results are explained by the linear fit. Both R^2 values are within 0.1% of a perfect linear fit.

E + V	E =	V	E	DFS Avg	Khan Avg	DFS Faster?	DFS Avg	Khan Avg	DFS Faster?
10,000	V / 10	9090	910	5.8	6.5	-0.7	2.24	3.16	-0.92
	3 * V / 4	5714	4286	1.6	4	-2.4			
	V	5000	5000	2.2	3.7	-1.5			
	(1/4) * (V*(V-1)) / 2	279	9721	1.1	1	0.1			
	(1/2) * (V*(V-1)) / 2	198	9802	0.5	0.6	-0.1			
100,000	V / 10	90909	9091	21.1	34.1	-13	17.12	22.94	-5.82
	3 * V / 4	57142	42858	20.6	28.7	-8.1			
	V	50000	50000	21	29.8	-8.8			
	(1/4) * (V*(V-1)) / 2	890	99110	11.2	9.2	2			
	(1/2) * (V*(V-1)) / 2	630	99370	11.7	12.9	-1.2			
500,000	V / 10	454545	45455	141.5	249.5	-108	118.48	148.06	-29.58
	3 * V / 4	285714	214286	144.2	193.8	-49.6			
	V	250000	250000	130.1	183.3	-53.2			
	(1/4) * (V*(V-1)) / 2	1996	498004	77.5	65.5	12			
	(1/2) * (V*(V-1)) / 2	1412	498588	99.1	48.2	50.9			
1,000,000	V / 10	909090	90910	264.6	522.9	-258.3	228.88	353.94	-125.06
	3 * V / 4	571428	428572	281.1	512	-230.9			
	V	500000	500000	357.9	540.4	-182.5			
	(1/4) * (V*(V-1)) / 2	2824	997176	118	105.4	12.6			
	(1/2) * (V*(V-1)) / 2	1998	998002	122.8	89	33.8			
5,000,000	V / 10	4545454	454546	1248.4	3175.4	-1927	1146.56	1920.82	-774.26
	3 * V / 4	2857142	2142858	1790.6	2857.7	-1067.1			
	V	2500000	2500000	1543.8	2686	-1142.2			
	(1/4) * (V*(V-1)) / 2	6321	4993679	587.3	435.2	152.1			
	(1/2) * (V*(V-1)) / 2	4470	4995530	562.7	449.8	112.9			
						Linear Fit R^2:	**0.99995**	**0.999541**	

Table 1: Tabular comparison data for two topological sort implementations.

The "Linear Fit R^2" values in the table show that both topological sort implementations run in $O(|V| + |E|)$ time. It is interesting to note that the DFS-based implementation is faster for graphs with sparse edges, while the Khan topological sort is faster for $|E| = \frac{1}{4} * \frac{|V|*(|V|-1)}{2}$ and $|E| = \frac{1}{2} * \frac{|V|*(|V|-1)}{2}$ cases. This can be seen in the green highlighting of the leftmost "DFS Faster?" column.

Overall, the DFS-based implementation is faster than the Khan implementation for every N. The difference between the two implementations' execution times is linear (as expected) and can be seen in the rightmost "DFS Faster?" column. Although both implementations have linear growth functions, big-O notation hides the constant factor which differentiates their execution times.

The space complexities of the implementations are equal, since both maintain an array or stack whose size is the number of vertices in the given test graph $O(|V|)$.

Appendix A: Source Code

Code for this project can be found at https://github.com/alexshank/algorithms-topological-sort.