ECE 608 Topological Sort Assignment Report

Name: Andrew Gan

Abstract

Topological sort is an algorithm that sorts vertices such that all edges point from the left to the right in the list of sorted vertices. Vertices on the left are expected to have more outgoing edges whereas the ones on the right are expected to have more incoming edges. Two topological sorting algorithms are compared against different graph configurations to determine the advantages and drawbacks of both approaches. [1]

DFS algorithm

The DFS approach dives into the graph and inserts a vertex into the sorted array in reversed order after all adjacent vertices are visited. This algorithm uses recursion, thus vertices are added starting with the one at the end of a path. Finishing time was ignore since vertices are immediately added to the sorted array, negating its importance.

```
_dfsRecurse(G v, S):
    if v is finished:
        return;
    for a ← v.adjacencyList:
        _dfsRecurse(G, v, S)
    v.finished = true
    S.push_front(v)

Topological-DFS(G, S):
    for v ← G.vertices:
        _dfsRecurse(G, v, S)
```

Kahn's algorithm

Kahn's algorithm iterates across every root vertex. When a root vertex is visited, the vertex is added to the sorted array and outgoing edges are deleted. This may create new root vertices among the adjacent vertices which have to be added to the sorted array as well. Each iteration only checks the immediate adjacent vertices.

```
Topological-Kahn(G, S):
    q ← all root vertices in G
    while(q is not empty):
        v ← q.pop()
        S.push_back(v)
        for a ← v.adjacencyList:
            removeEdge(v, a)
            if a is root:
                S.push_back(a)
```

Hypothesis

The effectiveness of the DFS algorithm is negatively correlated to the average depth of a graph. The greater the depth of a graph, the longer the recursive chain, the longer the runtime.

Kahn's algorithm is expected to perform better when the graph is more interconnected, as that will allow more elimination of vertices in each iteration, reducing the number of actual visits.

Density-controlled DAG Generation

The DAG generation algorithm first instantiates an array of vertices and assigns an ID to each of them. Then, each vertex is assigned a random outgoing edge configuration. To ensure the graph is acyclic, edges may only connect a larger id vertex to a smaller id vertex. This also represents the worst case input possible for a sorting algorithm.

```
G ← graph
for id ← 1..size:
    v ← new vertex
    v.id ← id
    v.finished ← false
    v.adj ← random() * density
    G.addVertex(v)
    G.remove_all_cycles()
```

Graphs with varying degrees of graph density were generated. A graph with a 25% density would contain 25% of all containable edges.[2]

Depth-controlled DAG Generation

It was hypothesized that the DFS approach would have its runtime affected by the depth of the graph. This inspired the generation of graphs with varying depths to measure the correlation.

A single-path-graph with varying depth is used. The graph is generated by connecting *depth* number of elements into a path. For example, if the size is 9 and depth is 2, the resultant graph has paths ($2 \rightarrow 1 \rightarrow 0$), ($5 \rightarrow 4 \rightarrow 3$), ($8 \rightarrow 7 \rightarrow 6$).

```
G ← graph
for v ← 1..size..depth:
    for d ← 1..depth:
        v.id ← id
        v.finished = false
        v.addEdge(v-1)
```

Verification of Topological Sort

The verification algorithm iterates through each vertex in the sorted array and returns false when an edge connecting to a preceding vertex in the sorted array exists. This was done in polynomial time.

Benchmark Tests

Four graph density options were available: 25%, 50% and 75%. For each graph density, the number of vertices doubled from 2 until 32768.

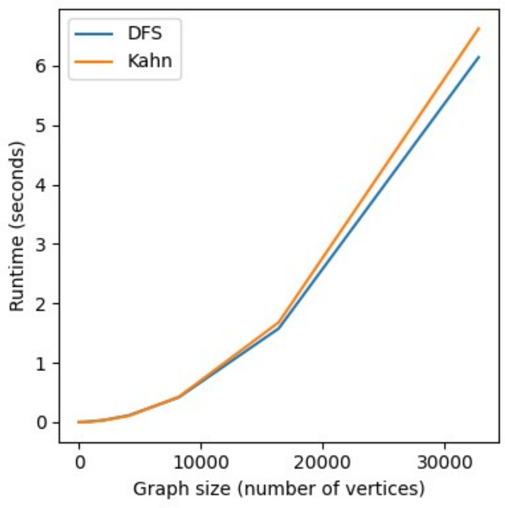
Single-path graphs of depth 100, 1000 and 10000 are used to test algorithm effectiveness against graph depth limits.

Each configuration was run three times and the average was retrieved. The tests were conducted on **eceprog.ecn.purdue.edu** for consistency, and in C for closer to baremetal performance.

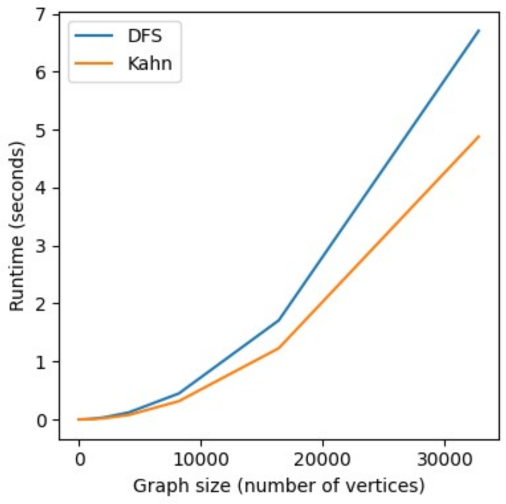
Results

The graphs were plotted for edge densities 25%, 50% and 75%. The DFS approach seem unaffected by graph density whereas the Kahn's algorithm benefitted significantly from a more interconnected graph. The runtime of Kahn's exceeded DFS when the graph density reached 50%.

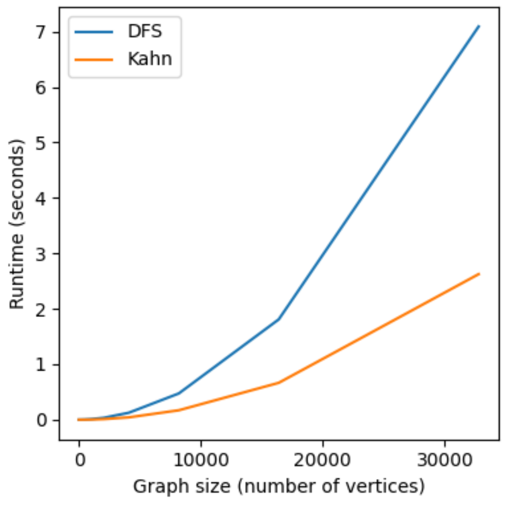
Graph density at 25%



Graph density at 50%



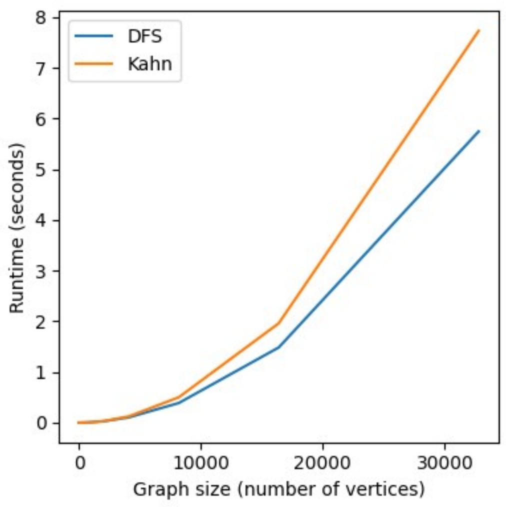
Graph density at 75%



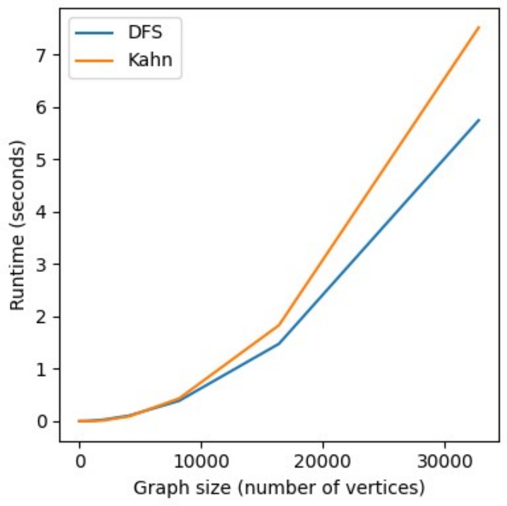
When the graph depth limit was scaled up, DFS was surpassed in performance by the Kahn's.

It was noted that unlike the graph density change which caused the graphs to diverge, the graph depth limit change kept both graphs in parallel.

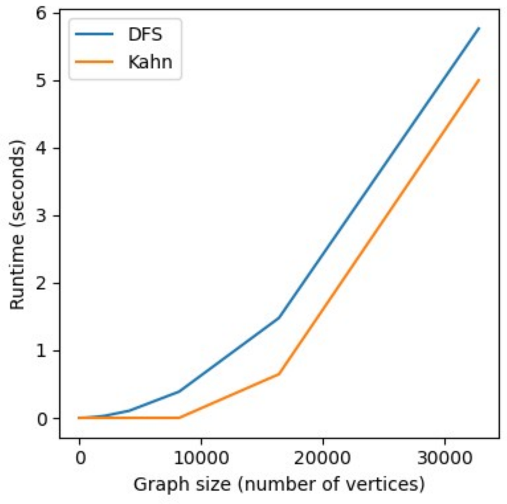
Graph depth limit at 100



Graph depth limit at 1k



Graph depth limit at 10k



Analysis

The graphs indicate that Kahn's algorithm performed better when the graph becomes densely populated with edges, which confirms the second hypothesis due to elimination of more adjancent vertices per iteration. Meanwhile, a less connected graph would require more iterations to cover the isolated subgraphs.

The DFS approach, on the other hand, was more resilient to graph connectivity. This can be explained by the fact that DFS is an algorithm that makes recursive calls for every depth of the graph, rather than for every adjacency vertex of the current vertex. The depth was randomly generated for the density-controlled graph and was therefore not impacting DFS significantly.

When controlled for depth, however, the DFS was restricted by the recursive depth limit of the stack, which can be circumvented by using an iterative approach. Meanwhile, the time taken for Kahn's algorithm decreases due to depth indirectly contributing to increased connectivity of the graph.

Eventually, the runtime for Kahn was slightly below DFS when the graph depth limit reached a critical point of 10k.

Conclusion

The algorithms can be summarized below:

	DFS	Kahn's
Function call	Recursive/ Iterative	Iterative
Information preprocessing	Reset all vertices to initial state	Add root vertices to set
Time complexity	O(V+E)	O(V+E)
Vertices visit pattern	Dive down one path before moving on	Explore graph layer by layer
Graph Density	Little to no effect	Performs better when dense
Depth limit	Has a limit on recursion depth limit	Performed better due to depth increasing connectivity
Function call chain length	Can reach number of vertices	Once, from vertex to adjacency

Both Kahn's and DFS have runtime O(V+E). The starting configuration of the graph, the information available such as the starting root vertex set, as well as the data structure used to access the graph are important factors when deciding which algorithm to implement. DFS favors graphs with limited depth and are somewhat dense, whereas Kahn's algorithm favors graphs that are very dense and have longer depths.

References

[1] C. al, *Introduction to algorithms*. Cambridge, MA: MIT Press, 2003. Chp. 22, pp. 549.

[2] S. Datta, "Graph density," *Baeldung on Computer Science*, 06-Nov-2022. [Online]. Available: https://www.baeldung.com/cs/graph-density. [Accessed: 18-Nov-2022].