41052 Assessment Task 1: Algorithm Implementation

Topological Sort

Oliver Salman (13881750)

October 7, 2022

1 Overview

Topological sort takes a directed acyclic graph (DAG) and returns its topological ordering. A topological ordering is a linear ordering of the vertices in the graph such that, for every directed edge, uv, vertex u comes before vertex v. Topological orderings are just a valid sequence for each task to be executed. For example, the vertices of a graph may represent tasks to be completed, and the edges may reflect restrictions that one tasks must be performed before another. More formally, a topological sort is a graph traversal in which each node v is visited only after all its dependencies have been visited. For this to be possible, the graph must not have any directed cycles. A graph with no directed cycles is called a Directed Acyclic Graph (DAG) with any DAG having at least one topological ordering. This report will further outline the applications of topological sorting, its theoretical complexity, the complexity of my implementation, and a discussion on the tests I've implemented.

2 Application Domains

The applications of topological sorting are broad and can be used for any kind of problem requiring an order of executing tasks, therefore it is not domain specific, but rather application specific. For example, a contrived application of topological sort is a library dependency build system. If you consider a library (say this one) which has dependencies to other libraries, there must exist a specific ordering of installing these libraries so the library of interest installs correctly. For example, let's say we have a library A which depends on libraries B and C. In this instance, the libraries can be represented as nodes, and each libraries dependencies can be represented as directed edges. The topological ordering of this example will be B -¿ C -¿ A, which specifies the orderings of each installation so that libraries will be installed with their dependencies. If, for example, library A has dependencies on libraries B and C and library B has a dependency on library A, it is clear that this is a cycle and will break the assumption that the input graph must be a graph containing no cycles. This contrived example illustrates how scheduling works and how it is used. One real-world application domain is in the software industry, where data pipelines depend heavily on the ordering of the executions of tasks. A pipeline should be intelligent to recognise that to load data into a target source, it must first be cleaned and transformed, and for it to be transformed, it must be available from clients to extract from.

3 Theoretical Complexity

There are two main algorithms that implement a topological sort of a DAG. Both algorithms, being a modified Depth-First Search, and Kahn's Algorithm have running time linear to the number of nodes plus the number of edges. Its worst-case running time is therefore $O(|V|+|E|)$. Both Kahn's algorithm and the modified DFS require an extra O(V) of auxiliary space to store a temporary data structure. In the modified DFS approach, sorted nodes are not returned immediately, rather, they are pushed to a stack and is only returned once all nodes have been topologically sorted and correctly pushed to the stack. Similarly, with Kahn's algorithm, a queue is used instead, adding nodes to the queue only when a node is found with an indegree (no incoming edges) equal to 0.

4 Implementation

My implementation of topological sort used a modified depth-first search using OCaML. The main structure of my implementation is comprised of two functions, one being the main entry function, and the other being the modified depth first search algorithm. The main entry function 'tsort' takes an input graph as an integer list of lists ((int * int list) list) and which calls the main 'dfs' function. The output of the 'dfs' function returns each topologically sorted node in order, where the 'tsort' function recursively creates the list of these sorted nodes. I have also re-implemented some helper OCaML native functions such as List.mem, List.assoc, List.map, and List.fold_left. Pseudocode for my implementation are as follows:

```
(* function dfs*)
let dfs g, s =
  let rec explore p, v, n =
    if n is in p then raise CycleFound
    else if n is in v then v else
      let p' = prepend n to p
      let n' = neighbours of n
      let v' = explore p', v, n'
      prepend n to v'
  explore [], v, s

(* function tsort *)
let tsort g =
  let nodes = all nodes of g fold_left dfs g, nodes
```

5 Assumptions

My implementation requires little assumptions for it to work as intended. The first of which being it is assumed that the input graph contains no cycles. This case is handled with an exception being raised, but the program will not produced a topological ordering with a graph containing a cycle. Though, this can be overcome by performing a topological sort on each strongly connected component of the graph instead.

6 Testing

The testing suite of my implementation was done using the OCaML ppx_expect library, a testing library created by Jane Street that differs from traditional testing libraries. The ppx_expect library differs by inserting print statements in source code and comparing these print statements against the expected printed output. Here is a great article written by a Jane Street engineer outlining testing with expectations. My testing suite (here) covers tests regarding all different test inputs that the user can give, including inputs that raise exceptions.

7 Insights

For anyone else implementing topological sort, it is advised to become very competent with depth-first search (though breadth first search *can* work too). Also fully understanding how and when nodes should be pushed to the output stack requires some knowledge of elementary graph theory. Completing this assignment also led me to realise that this algorithm would have been better implemented using an imperative programming language (but learning a new programming style was still nevertheless very fun).