40152 Advanced Algorithms Assignment 1: Topological Sort

Oliver Salman 13881750

40152 Advanced Algorithms Assignment 1: Topological Sort

# 

# 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.

**Application domains**

The applications of topological sorting are broad and can be used for any kind of problem requiring an order of executing tasks. For example, a contrived application of topological sort is a dependency build system. If you consider a codebase that contains lots of libraries that all require installation, they will most likely have dependencies referencing each another. 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. This contrived example illustrates how scheduling works and how it is used. Real-world applications and their domains include but are not limited to logistics planning, university course scheduling, and operating systems deadlock detection.

**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 . 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 pushed to the stack. Similarly, with Kahn’s algorithm, a queue is used instead, adding nodes to the queue with an indegree equal to 0.

**Implementation**

My implementation of topological sort used a modified depth-first search using OCaML programming language. Hence, the structure of the implementation differs from implementation via an imperative programming language. The main structure of my implementation 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 (otherwise known as a dictionary in other imperative programming languages) 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 implemented some helper functions `is\_in` and assoc, which are like OCaML base functions `List.mem` and `List.assoc`. 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